

# Informatische Werkzeuge in den Geistes- und Sozialwissenschaften 1/2

Prof. Dr. Michael Kohlhase

Professur für Wissensrepräsentation und -verarbeitung  
Informatik, FAU Erlangen-Nürnberg  
`Michael.Kohlhase@FAU.de`

August 25, 2020

## Preface

### Course Concept

**Objective:** The course aims at giving students an overview over the variety of digital tools and methods at the disposal of practitioners of the humanities and social sciences, explaining their intuitions on how/why they work (the way they do). The main goal of the course is to empower students for their for the emerging discipline of “digital humanities and social sciences”. In contrast to a classical course in Computer Science which lays the mathematical and computational foundations which will become useful in the long run, we want to introduce methods and tools that can become *useful in the short term* and thus generate immediate success and gratification, thus alleviating the “programming shock” (the brain stops working when in contact with computer science tools or computer scientists) common in the humanities and social sciences.

**Original Context:** The course “Informatische Werkzeuge in den Geistes- und Sozialwissenschaften” is a first-year, two-semester course in the bachelor program “Digitale Geistes- und Sozialwissenschaften” (Digital Humanities and Social Sciences: DigiHumS) at FAU Erlangen-Nürnberg.

**Open to External Students:** Other Bachelor programs are increasingly co-opting the course as specialization option or a key skill. There is no inherent restriction to DHSS students in this course.

**Prerequisites:** There are no formal prerequisites – after all it starts in the first semester for DigiHumS – but a good deal of motivation, openness towards exploring the weird and wonderful world digital methods and tools, and a certain perseverance in the face of not understanding directly help tremendously and helps having fun in this course.

We do assume that students have a personal laptop, or access to a computer where they have admin rights, i.e. can install software. This is necessary for solving the homework. In particular, smartphones and most tablet computers will not suffice.

### Course Contents

The course comprises two parts that are given as two-hour/week lectures.

**IWGS 1 (the first semester):** begins with an introduction to programming in python which we will use as the main computational tool in the course; see Chapter 2 and Chapter 3. In particular we will cover

- systematics and culture of programming
- program and control structures
- basic data structures like numbers and strings, in particular character encodings, unicode, and regular expressions.

Building on this, we will cover

1. digital documents and document processing, in particular; text files, markup systems, HTML, and XML; see Chapter 4.
2. basic concepts of the World-Wide-Web; see Chapter 5
3. Web technologies for interactive documents and applications; in particular Internet infrastructure, web browsers and servers, PHP, dynamic HTML, Javascript, and CSS; see Chapter 6.

**IWGS 2 (the second semester):** covers selected topics and exemplary tools that will become useful in the DH. We will cover

1. Data bases; in particular Entity Relationship diagrams, CRUD operations, and DB querying; see Chapter 9.
2. large-scale collaborative development tools: revision control system and issue trackers, in particular Git and GitLab; see Chapter 10
3. Image processing tools, see Chapter 12
4. Copyright and Data Privacy as legal foundations of DH tools; see Chapter 13
5. Using the Ontologies and the Semantic Web for Cultural Heritage; see Chapter 14
6. The WissKI System: A Virtual Research Environment for Cultural Heritage; see Chapter 15

**Idea:** The first semester lays the foundations by introducing programming in `python` and work our way towards web applications, which form the base of most modern tools in the DH. In Chapter 11, we pull all parts together to build a first, simple web application with persistent storage that manages a set of books.

After an excursion into project management systems, we introduce images and tools for their management. Here, we extend our web application to deal with image fragments; actually building a simple replacement for a prominent DH web application.

Finally, after another excursion – this time into the legal foundations of intellectual property and data privacy the course culminates in an introduction of the WissKI system, a virtual research environment for documenting cultural heritage artefacts. Indeed the WissKI system combines all topics in the course so far.

## This Document

**Format:** The document mixes the slides presented in class with comments of the instructor to give students a more complete background reference.

**Caveat:** This document is primarily made available for the students of the IWGS course only. After two iterations of this course it is reasonably feature-complete, but will evolve and be polished in coming academic years.

**Licensing:** This document is licensed under a [Creative Commons license](#) that [requires attribution](#), [allows commercial use](#), and [allows derivative works](#) as long as [these are licensed under the same license](#).

**Knowledge Representation Experiment:** This document is also an experiment in knowledge representation. Under the hood, it uses the `SiTeX` package [Koh08; Koh20], a `TeX/LaTeX` extension for semantic markup, which allows to export the contents into [active documents](#) that adapt to the reader and can be instrumented with services based on the explicitly represented meaning of the documents.

**Other Resources:** The course notes will be complemented by a selection of problems (with and without solutions) that can be used for self-study; see <http://kwarc.info/teaching/IWGS>.

## Acknowledgments

**Materials:** The materials in this course are partially based on various lectures the author has given at Jacobs University Bremen in the years 2010-2016, these in turn have been partially based on materials and courses by Dr. Heinrich Stamerjohanns, PD Dr. Florian Rabe, and Prof. Dr. Peter Baumann. Chapter 12 have been provided by Philipp Kurth and Dr. Frank Bauer.

All course materials have been restructured and semantically annotated in the `SiTeX` format, so that we can base additional semantic services on them.

**IWGS Students:** The following students have submitted corrections and suggestions to this and earlier versions of the notes: Paul Moritz Wegener, Michael Gräwe.

## Recorded Syllabus

In this document, we record the progress of the course in the academic year 2019/20 in the form of a “recorded syllabus”, i.e. a syllabus that is created after the fact rather than before. For the topics planned for this course, see above.

Recorded Syllabus Winter Semester 2019/20:

#	date	until	slide	page
1	Oct 18.	admin, overview	16	9
2	Oct 24.	python intro	30	24
3	Oct 31.	python fundamentals	42	30
4	Nov. 7.	lists, dictionaries, input/output	55	37
5	Nov. 14.	number/character representation, unicode	84	55
6	Nov. 21.	strings, functions, regular expressions	88	57
7	Dec. 28.	plain/formatted text, HTML	104	68
8	Dec. 5.	HTML & XML	111	74
9	Dec. 12.	Documents as trees & XML	129	84
10	Dec. 19.	XML, XPath, URIs	139	92
10	Jan. 9.	WWW Architecture, URIs, HTTP	152	103
11	Jan. 16.	web applications, bottle	164	109
12	Jan. 23.	Cascading Style Sheets	182	121

Syllabus and Schedule for the online Summer Semester 2020:

Due to the Corona crisis in Spring/Summer 2020, at least the beginning of SS20 is likely to be completely online. We will post the recording at <https://fau.tv>.

#	date	until	slide	page
1.	April 23.	admin, overview, artefact lifecycles	240	169
2.	April 30.	revision control	246	173
3.	May 7.	distributed revision control, workflows issues	260	182
4.	May 14.	Databases, DDL, sqlite3, python exceptions	224	157
	May 21.	Public Holiday: Christi Himmelfahrt		
5.	May 28.	SQL Queries, Views	233	162
6.	June 4.	The Books App Project	293	203
	June 11.	Public Holiday: Fronleichnam		
7.	June 18.	Image Processing	333	235
8.	June 25.	Image Maps via SVG/CSS	360	256
9.	July 2.	Legal Foundations of IT	378	268
10.	July 9.	Information Privacy, Semantic Networks	405	285
11.	July 16.	Modeling Artefacts in CIDOC CRM	417	293
11.	July 23.	WissKI Architecture	458	319
12.	July 30.	Linked Open Data, What have we learned	470	327
	Aug. 6.	Exam		

Here the syllabus of the last academic year for reference, the current year should be similar; see the course notes of last year available for reference at <http://kwarc.info/teaching/IWGS/notes-2018-19.pdf>.

Recorded Syllabus Winter Semester 2018/19:

#	date	until	slide	page
1	Oct 18.	admin, overview		
2	Oct 25.	python intro		
	Nov. 1.	All Hallows Day (public holiday)		
3	Nov. 8.	python fundamentals		
4	Nov. 15.	review fundamentals, functions		
5	Nov. 22.	number/character representation, unicode		
6	Nov. 29.	regular expressions		
7	Dec. 6.	plain/formatted text, HTML		
8	Dec. 13.	HTML & CSS		
9	Dec. 20.	Review: HTML & CSS		
10	Jan. 10.	New Year recap; CSS		
11	Jan. 17.	Architecture of the WWW		
12	Jan. 24.	web applications, bottle		
13	Jan. 31.	client-side computation, JavaScript, JQuery		

Recorded Syllabus Summer Semester 2019:

#	date	until	slide	page
1.	April 25.	admin, overview, Revision Control		
2.	May 2.	distributed revision control, workflows		
3.	May 9.	GitLab, issues		
4.	May 16.	Databases, DDL, sqlite3		
5.	May 23.	SQL Queries, Views		
	May 30	Public Holiday: Christi Himmelfahrt		
6.	June 6.	Image Processing		
7.	June 13.	Image Maps via SVG/CSS		
	June 20.	Public Holiday: Fronleichnam		
8.	June 27.	Legal Foundations of IT		
9.	July 4.	Information Privacy, Semantic Web		
10.	July 11.	RDF, Linkd Open Data		
11.	July 18.	WissKI, What have we learned		
	July 25.	Exam		



# Contents

Preface . . . . .	i
Course Concept . . . . .	i
Course Contents . . . . .	i
This Document . . . . .	ii
Acknowledgments . . . . .	ii
Recorded Syllabus . . . . .	iv
<b>1 Preliminaries</b>	<b>1</b>
1.1 Administrative . . . . .	1
1.2 Goals, Culture, & Outline of IWGS . . . . .	5
1.3 About My Lecturing ... . . . .	6
<b>I IWGS-1: Programming, Documents, Web Applications</b>	<b>11</b>
<b>2 Introduction to Programming</b>	<b>13</b>
2.1 Programming in IWGS . . . . .	13
2.1.1 Introduction to Programming . . . . .	13
2.1.2 Programming in IWGS . . . . .	16
2.2 Programming in Python . . . . .	18
2.2.1 Hello IWGS . . . . .	18
2.2.2 Variables and Types . . . . .	25
2.2.3 Python Control Structures . . . . .	28
2.3 Some Thoughts about Computers and Programs . . . . .	32
2.4 More about Python . . . . .	34
2.4.1 Sequences and Iteration . . . . .	34
2.4.2 Input and Output . . . . .	37
2.4.3 Functions and Libraries in Python . . . . .	39
2.4.4 A Final word on Programming in IWGS . . . . .	41
<b>3 Numbers, Characters, and Strings</b>	<b>43</b>
3.1 Representing and Manipulating Numbers . . . . .	43
3.2 Characters and their Encodings: ASCII and UniCode . . . . .	47
3.3 More on Computing with Strings . . . . .	51
3.4 More on Functions in Python . . . . .	54
3.5 Regular Expressions: Patterns in Strings . . . . .	57
<b>4 Documents as Digital Objects</b>	<b>63</b>
4.1 Representing & Manipulating Documents on a Computer . . . . .	63
4.2 Measuring Sizes of Documents/Units of Information . . . . .	66
4.3 Hypertext Markup Language . . . . .	68
4.4 Documents as Trees . . . . .	75
4.5 An Overview over XML Technologies . . . . .	79

<b>5</b>	<b>Basic Concepts of the World Wide Web</b>	<b>91</b>
5.1	Preliminaries . . . . .	91
5.2	Addressing on the World Wide Web . . . . .	92
5.3	Running the World Wide Web . . . . .	95
<b>6</b>	<b>Web Applications</b>	<b>99</b>
6.1	Recap: HTML Forms Data Transmission . . . . .	100
6.2	Generating HTML on the Server . . . . .	103
6.2.1	Routing and Argument Passing in Bottle . . . . .	104
6.2.2	Templating in Python via STPL . . . . .	107
6.2.3	Completing the Contact Form . . . . .	110
6.3	Cascading Stylesheets . . . . .	112
6.3.1	Separating Content from Layout . . . . .	112
6.3.2	A small but useful Fragment of CSS . . . . .	114
6.3.3	CSS Tools . . . . .	119
6.3.4	Worked Example: The Contact Form . . . . .	121
6.4	Dynamic HTML: Client-side Manipulation of HTML Documents . . . . .	123
6.4.1	JavaScript in HTML . . . . .	124
6.4.2	JQuery: Write Less, Do More . . . . .	127
<b>7</b>	<b>What did we learn in IWGS-1?</b>	<b>131</b>
<b>II</b>	<b>IWGS-II: DH Project Tools</b>	<b>133</b>
<b>8</b>	<b>Semester Change-Over</b>	<b>135</b>
8.1	Administrativa . . . . .	135
<b>9</b>	<b>Databases</b>	<b>141</b>
9.1	Introduction . . . . .	141
9.2	Relational Databases . . . . .	143
9.3	SQL – A Standardized Interface to RDBMS . . . . .	145
9.4	ER-Diagrams and Complex Database Schemata . . . . .	148
9.5	RDBMS in Python . . . . .	152
9.6	Excursion: Programming with Exceptions in Python . . . . .	156
9.7	Querying and Views in SQL . . . . .	157
9.8	Querying via Python . . . . .	162
<b>10</b>	<b>Collaboration and Project Management</b>	<b>165</b>
10.1	Revision Control Systems . . . . .	165
10.1.1	Dealing with Large/Distributed Projects and Document Collections . . . . .	165
10.1.2	Centralized Version Control . . . . .	170
10.1.3	Distributed Revision Control . . . . .	174
10.1.4	Working with GIT in small Projects . . . . .	176
10.1.5	Working with GIT in large Projects . . . . .	179
10.2	Working with GIT and GitLab/GitHub . . . . .	181
10.3	Excursion: Authentication with SSH . . . . .	183
10.4	Bug/Issue Tracking Systems . . . . .	184
<b>11</b>	<b>Project: A Web GUI for a Books Database</b>	<b>191</b>
11.1	A Basic Web Application . . . . .	191
11.2	Access Control and Management . . . . .	199
11.3	Deploying the Books Application as a Program . . . . .	204
<b>12</b>	<b>Image Processing</b>	<b>207</b>

<b>13 Legal Foundations of Information Technology</b>	<b>257</b>
13.1 Intellectual Property . . . . .	257
13.2 Copyright . . . . .	260
13.3 Licensing . . . . .	263
13.4 Information Privacy . . . . .	267
<b>14 Ontologies, Semantic Web for Cultural Heritage</b>	<b>271</b>
14.1 Documenting our Cultural Heritage . . . . .	271
14.2 Systems for Documenting the Cultural Heritage . . . . .	274
14.3 The Semantic Web . . . . .	278
14.4 Semantic Networks and Ontologies . . . . .	283
14.5 CIDOC CRM: An Ontology for Cultural Heritage . . . . .	289
14.6 The Semantic Web Technology Stack . . . . .	295
14.7 Ontologies vs. Databases . . . . .	301
<b>15 The WissKI System</b>	<b>307</b>
15.1 WissKI extends Drupal . . . . .	308
15.2 Dealing with Ontology Paths: The WissKI Pathbuilder . . . . .	311
15.3 The WissKI Link Block . . . . .	315
15.4 Cultural Heritage Research: Querying WissKI Resources . . . . .	317
15.5 Application Ontologies in WissKI . . . . .	320
15.6 The Linked Open Data Cloud . . . . .	321
<b>16 What did we learn in IWGS?</b>	<b>327</b>



# Chapter 1

## Preliminaries

### 1.1 Administrativa

We will now go through the ground rules for the course. This is a kind of a social contract between the instructor and the students. Both have to keep their side of the deal to make learning as efficient and painless as possible.

#### Prerequisites

- ▷ **Prerequisites:** Motivation, interest, curiosity, hard work.
  - ▷ **nothing else!** We will teach you all you need to know.
  - ▷ You can do this course if you want!



©: Michael Kohlhasé

1



Now we come to a topic that is always interesting to the students: the grading scheme: The short story is that things are complicated. We have to strike a good balance between what is didactically useful and what is allowed by Bavarian law and the FAU rules.

#### Assessment, Grades

- ▷ **Grading Background/Theory:** only modules are graded (by the law)
  - ▷ module “DH-Einführung”  $\hat{=}$  courses IWGS1/2, DH-Einführung
  - ▷ DHE module grade  $\leadsto$  pass/fail determined by “portfolio”  $\hat{=}$  collection of contributions/assessments
- ▷ **Assessment Practice:** The IWGS assessments in the “portfolio” consist of
  - ▷ weekly homework assignments (practice IWGS concepts and tools)
  - ▷ 60 minutes exam directly after lectures end:  $\sim$  Feb.10. (to show you master them)
- ▷ **Retake Exam:** 60 min exam at the end of the semester ( $\sim$  Sep 30.)

- ▷ To help you succeed: we offer you
  - ▷ External motivation: points for homeworks and a grade for exam (even though only pass/fail relevant in the end)
  - ▷ Mid-semester mini-exam (online, optional, corrected but ungraded), (so you can predict the exam style)
  - ▷ weekly online quizzes that help you prepare for the course (ungraded  $\leadsto$  check understanding/preparation)



©: Michael Kohlhasse

2



Homework assignments, quizzes and end-semester exam may seem like a lot of work – and indeed they are – but you will need practice (getting your hands dirty) to master the concepts. We will go into the details next.

## IWGS Homework Assignments

- ▷ Homeworks: will be small individual problem/programming/system assignments (but take time to solve) group submission if and only if explicitly permitted
- ▷ Admin: To keep things running smoothly
  - ▷ Homeworks will be posted on StudOn (<https://studon.fau.de/studon/crs2287043.html>)
  - ▷ Homeworks are handed in electronically (plain text, program files, PDF)
  - ▷ go to the tutorials, discuss with your TA (they are there for you!)
- ▷ Homework Discipline:
  - ▷ start early! (many assignments need more than one evening's work)
  - ▷ Don't start by sitting at a blank screen
  - ▷ Humans will be trying to understand the text/code/math when grading it.



©: Michael Kohlhasse

3



It is very well-established experience that without doing the homework assignments (or something similar) on your own, you will not master the concepts, you will not even be able to ask sensible questions, and take nothing home from the course. Just sitting in the course and nodding is not enough!

If you have questions please make sure you discuss them with the instructor, the teaching assistants, or your fellow students. There are three sensible venues for such discussions: online in the lecture, in the tutorials, which we discuss now, or in the course forum – see below. Finally, it is always a very good idea to form study groups with your friends.

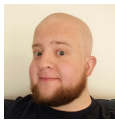

## IWGS Tutorials

- ▷ Weekly tutorials and homework assignments (first one in week two)



Teaching Assistants: (Doctoral Students in CS)

- ▷ Jonas Betzendahl: [jonas.betzendahl@fau.de](mailto:jonas.betzendahl@fau.de)
- ▷ Philipp Kurth: [philipp.kurth@fau.de](mailto:philipp.kurth@fau.de)

They know what they are doing and really want to help you learn! (dedicated to DH)

- ▷ Goal 1: Reinforce what was taught in class (important pillar of the IWGS concept)
- ▷ Goal 2: Let you experiment with python (think of them as Programming Labs)
- ▷ Life-saving Advice: go to your tutorial, and prepare it by having looked at the slides and the homework assignments
- ▷ Inverted Classroom: the latest craze in didactics (works well if done right)  
in CS: Lecture + Homework assignments + Tutorials  $\hat{=}$  Inverted Classroom



 ©: Michael Kohlhase 4 

Do use the opportunity to discuss the IWGS topics with others. After all, one of the non-trivial inter/transdisciplinary skills you want to learn in the course is how to talk about Computer Science topics – maybe even with real Computer Scientists. And that takes practice, practice, and practice.

But what if you are not in a lecture or tutorial and want to find out more about the IWGS topics?

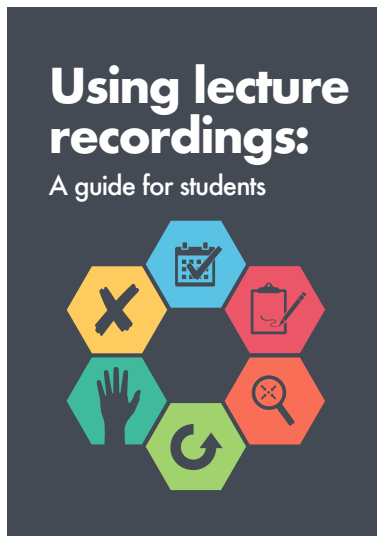
## Textbook, Handouts and Information, Forums







- ▷ No Textbook: but lots of online tutorials on the web
- ▷ Course notes will be posted at <http://kwarc.info/teaching/IWGS> (see references)
  - ▷ I mostly prepare/adapt/correct them as we go along.
  - ▷ please e-mail me any errors/shortcomings you notice. (improve for the group)
- ▷ Announcements will be posted on the StudOn course forum: [https://www.studon.fau.de/studon/goto.php?target=frm\\_2319978](https://www.studon.fau.de/studon/goto.php?target=frm_2319978)
- ▷ Check the forum frequently for
  - ▷ announcements, homework questions, ...
  - ▷ discussion among your fellow students
- ▷ If you become an active discussion group, the forum turns into a valuable resource!

 ©: Michael Kohlhase 5 

## Practical recommendations on Lecture Resources,

- ▷ Excellent Guide: [Nor+18a] (german Version at [Nor+18b])



-  Attend lectures.
-  Take notes.
-  Be specific.
-  Catch up.
-  Ask for help.
-  Don't cut corners.



©: Michael Kohlhasse

6



## Software/Hardware tools

- ▷ You will need computer access for this course
- ▷ we recommend the use of standard software tools
- ▷ find a **text editor** you are comfortable with (get good with it)  
A **text editor** is a program you can use to write **text files**. (not MS Word)
  - ▷ any **operating system** you like (I can only help with UNIX)
  - ▷ Any browser you like (I use Firefox: just a better browser (for Math))
  - ▷ **learn how to touch-type NOW** (reap the benefits earlier, not later)



©: Michael Kohlhasse

7



**Touch-typing:** You should not underestimate the amount of time you will spend typing during your studies. Even if you consider yourself fluent in two-finger typing, touch-typing will give you a factor two in speed. This ability will save you at least half an hour per day, once you master it. Which can make a crucial difference in your success.

Touch-typing is very easy to learn, if you practice about an hour a day for a week, you will re-gain your two-finger speed and from then on start saving time. There are various free typing tutors on the network. At [http://typingsoft.com/all\\_typing\\_tutors.htm](http://typingsoft.com/all_typing_tutors.htm) you can find about programs, most for windows, some for linux. I would probably try Ktouch or TuxType

Darko Pesikan (one of the previous TAs) recommends the TypingMaster program. You can download a demo version from <http://www.typingmaster.com/index.asp?go=tutordemo>

You can find more information by googling something like "learn to touch-type". (goto <http://www.google.com> and type these search terms).

## 1.2 Goals, Culture, & Outline of IWGS

### Goals of "IWGS"

- ▷ **Goal:** giving students an overview over the variety of digital tools and methods
- ▷ **Goal:** explaining their intuitions on how/why they work (the way they do).
- ▷ **Goal:** empower students for their for the emerging field "digital humanities and social sciences".
- ▷ **NON-Goal:** laying the mathematical and computational foundations which will become useful in the long run.
- ▷ **Method:** introduce methods and tools that can become *useful in the short term*
  - ▷ generate immediate success and gratification,
  - ▷ alleviate the "programming shock" (the brain stops working when in contact with computer science tools or computer scientists) common in the humanities and social sciences.



©: Michael Kohlhase

8



One of the most important tasks in an inter/trans-disciplinary enterprise – and that what “digital humanities” is, fundamentally – is to understand the disciplinary language, intuitions and foundational assumptions of the respective other side. Assuming that most students are more versed in the “humanities and social sciences” side we want to try to give an overview of the “Computer Science culture”.

### Academic Culture in Computer Science

- ▷ **Definition 1.2.1** The **academic culture** is the overall style of working, research, and discussion in an academic field.
- ▷ **Observation 1.2.2** *There are significant differences in the **academic culture** between **Computer Science** and the Humanities and Social Sciences.*
- ▷ Computer Science is an **Engineering Discipline** (we build things)
  - ▷ given a problem we look for a (mathematical) model, we can think with
  - ▷ once we have one, we try to re-express it with fewer “primitives” (concepts)
  - ▷ once we have, we generalize it (make it more widely applicable)
  - ▷ only then do we implement it in a program (ideally)

Design of versatile, usable, and elegant tools is a main concern

- ▷ almost all technical literature is in English (technical Vocabulary too)
- ▷ CSlings love shallow hierarchies (Kein Personenkult; alle per Du)



©: Michael Kohlhasse

9



Please keep in mind that – self-awareness is always difficult – the list below may be incomplete and clouded by mirror-gazing.

We now come to the concrete topics we want to cover in . The guiding intuition for the selection is to concentrate on techniques that may become useful in day-to-day DH work – not CS-completeness or teaching efficiency.

### Outline of IWGS 1:

- ▷ programming in python (main tool in IWGS)
  - ▷ systematics and culture of programming
  - ▷ program and control structures
  - ▷ basic data structures like numbers and strings, character encodings, unicode, and regular expressions
- ▷ digital documents and document processing
  - ▷ text files
  - ▷ markup systems, HTML, and CSS
  - ▷ XML: Documents are trees.
- ▷ Web technologies for interactive documents and web applications
  - ▷ Internet infrastructure: web browsers and servers
  - ▷ serverside computing: bottle routing and
  - ▷ clientside interaction: dynamic HTML, Javascript, HTML forms
- ▷ Web Application Project (fill in the blanks to obtain a working web app)



©: Michael Kohlhasse

10



## 1.3 About My Lecturing ...

First let me state the obvious – this is really still part of the admin – but there is an important point I want to make.

### Do I need to attend the lectures

- ▷ Attendance is not mandatory for the IWGS lecture
- ▷ There are two ways of learning IWGS: (both are OK, your mileage may vary)

- ▷ Approach **B**: Read a **Book**
- ▷ Approach **I**: come to the lectures, be **involved**, interrupt me whenever you have a question.

The only advantage of **I** over **B** is that books do not answer questions (**yet!** ↔ **we are working on this in AI research**)

- ▷ Approach **S**: come to the lectures and **sleep does not work!**
- ▷ **I really mean it**: If you come to class, be involved, ask questions, challenge me with comments, tell me about errors, ...
  - ▷ I would much rather have a lively discussion than get through all the slides
  - ▷ You learn more, I have more fun (**Approach B serves as a backup**)
  - ▷ You may have to change your habits, overcome shyness, ... (**please do!**)
- ▷ This is what I get paid for, and I am more expensive than most books (**get your money's worth**)



©: Michael Kohlhasse

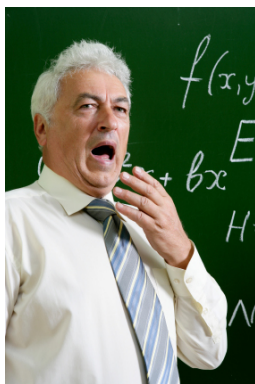
11



That being said – I know that it sounds quite idealistic – can I do something to help you along in this? Let me digress on lecturing styles → take the following with “cum kilo salis”<sup>1</sup>, I want to make a point here, not bad-mouth my colleagues.!

### Traditional Lectures (cum kilo salis)

- ▷ One person talks to 50+ students who just listen and take notes
- ▷ The **I have a book hat you do not have** style makes it hard to stay awake



- ▷ It is well-known that frontal teaching does not optimize learning
- ▷ But it scales very well (**especially when televised**)



©: Michael Kohlhasse

12



So there is a tension between

<sup>1</sup>with much more than the proverbial grain of salt.

- scalability of teaching – which is a legitimate concern for an institution like FAU, and
- effectiveness/efficiency of learning – which is a legitimate concern for students

### My Lectures? What can I do to keep you awake?

- ▷ We know how to keep large audiences engaged and motivated (even televised)
- ▷ But the topic is different (IWGS is arguably more complex than Sports/Media)



- ▷ We're not gonna be able to go all the way to TV entertainment ("IWGS total")
- ▷ But I am going to (try to) incorporate some elements ...



©: Michael Kohlhase

13



I will use interactive elements I call “questionnaires in my course. Here is one example to give you an idea of what is coming.

### The very first Questionnaire in IWGS

- ▷ **Question:** How many journal articles as “Digital Humanities” up to 2018
  - a) 7?
  - b) 1116?
  - c) 56.000?
- ▷ **Answer:**
  - a) 7 is much much too small (you could not study such a thin field at FAU)
  - b) 1116 this is the size of the DARIAH bibliography
  - c) 56.000 is the number of hits labeled “digital humanities” on google scholar (lots of duplicates likely)
- ▷ **Questionnaires:** are my attempt to get you to interact
  - ▷ At end of each logical unit (most, if I can get around to preparing them)

- ▷ You get 2 -5 minutes, feel free to make noise (e.g. discuss with your neighbors)



©: Michael Kohlhase

14



One of the reasons why I like the questionnaire format is that it is a small instance of a question-answer game that is much more effective in inducing learning – recall that learning happens in the head of the student, no matter what the instructor tries to do – than frontal lectures. In fact Sokrates – the grand old man of didactics – is said to have taught his students exclusively by asking leading questions. His style coined the name of the teaching style “Socratic Dialogue”, which unfortunately does not scale to a class of 100+ students.

### More Generally: My Questions to You

#### ▷ When will I ask them?

- ▷ In questionnaires.
- ▷ At various points during the lectures.
- ▷ We'll do examples together.

#### ▷ Why do I ask them?

- ▷ They give you the option to follow the lectures *actively*.
- ▷ They allow me to check whether or not you are able to follow.

#### ▷ How will I look for answers?

- ▷ “Streber syndrom”: 3 students answer all the questions,  $N - 3$  sleep.
- ▷ If this happens, I may resort to picking students randomly.

There is nothing to be ashamed of when giving a wrong answer! You wouldn't believe the number of times I got something wrong myself (I do hope all bugs are removed now, but ...)



©: Michael Kohlhase

15



Unfortunately, this idea of adding questionnaires is mitigated by a simple fact of life. Good questionnaires require good ideas, which are hard to come by; in particular for IWGS-2, I do not have many. But maybe you – the students – can help.

### Call for Help/Ideas with/for Questionnaires

- ▷ I have some questionnaires ..., but more would be good!
- ▷ I made some good ones ..., but better ones would be better
- ▷ Please help me with your ideas (I am not Stefan Raab)
  - ▷ You know something about IWGS by then.
  - ▷ You know when you would like to break the lecture by a questionnaire.
  - ▷ There must be a lot of hidden talent! (you are many, I am only one)

▷ I would be grateful just for the idea.

(I can work out the details)



©: Michael Kohlhasé

16



## Part I

# IWGS-1: Programming, Documents, Web Applications



## Chapter 2

# Introduction to Programming

### 2.1 Programming in IWGS

#### 2.1.1 Introduction to Programming

Programming is an important and distinctive part of “Informatische Werkzeuge in den Geistes- und Sozialwissenschaften” – the topic of this course. Before we delve into learning python, we will review some of the basics of computing to situate the discussion.

To understand programming, it is important to realize that that computers are universal machines. Unlike a conventional tool – e.g a spade – which has a limited number of purposes/behaviors – digging holes in case of a spade, maybe hitting someone over the head, a computer can be given arbitrary<sup>1</sup> purposes/behaviors by specifying them in form of a “program”.

This notion of a [program](#) as a behavior specification for an universal machine is so powerful, that the field of computer science is centered around studying it – and what we can do with [programs](#), this includes

- i)* storing and manipulating data about the world,
- ii)* encoding, generating, and interpreting images, audio, and video,
- iii)* transporting information for communication,
- iv)* representing knowledge and reasoning,
- v)* transforming, optimizing, and verifying other [programs](#),
- vi)* learning patterns in data and predicting the future from the past.

#### Computer Hardware/Software & Programming

- ▷ **Definition 2.1.1** [Computers](#) consist of [hardware](#) and [software](#).
- ▷ **Definition 2.1.2** [Hardware](#) consists of

---

<sup>1</sup>as long as they are “computable”, not all are.

- ▷ a **central processing unit (CPU)**
- ▷ **memory**: e.g. RAM, ROM, ...
- ▷ **storage devices**: e.g. Disks, SSD, tape, ...
- ▷ **input**: e.g. keyboard, mouse, touchscreen, ...
- ▷ **output**: e.g. screen, ear-phone, printer, ...

▷ **Definition 2.1.3** **Software** consists of

- ▷ **data** represents objects and their relationships in the world
- ▷ **programs** input, manipulate, output **data**

▷ **Remark 2.1.4** **Hardware** stores **data** and runs **programs**.

©: Michael Kohlhasse

17

A universal machine has to have – so experience in computer science shows – certain distinctive parts.

- A **CPU** that consists of a
  - **control unit** that interprets the **program** and controls the flow of instructions and
  - a **arithmetic/logic unit (ALU)** that does the actual computations internally.
- **Memory** that allows the system to store data during runtime (volatile storage; usually RAM) and between runs of the system (persistent storage; usually hard disks, solid state disks, magnetic tapes, or optical media).
- I/O devices for the communication with the user and other **computers**.

With these components we can build various kinds of universal machines; these range from thought experiments like Turing machines, to today's **general-purpose computers** like your laptop with various **embedded systems** (wristwatches, Internet routers, airbag controllers, ...) in-between.

Note that – given enough fantasy – the human brain has the same components. Indeed the human mind is a universal machine – we can think whatever we want, react to the environment, and are not limited to particular behaviors. There is a sub-field of Computer Science that studies this: **Artificial Intelligence (AI)**. In this analogy, the brain is the “hardware” –sometimes called “wetware” because it is not made of hard silicon or “meat machine”<sup>2</sup>. It is instructional to think about what the **program** and the data might be in this analogy.

## Programming Languages

- ▷ Programming  $\hat{=}$  writing **programs** (Telling the computer what to do)
- ▷ **Remark 2.1.5** The computer does exactly as told

<sup>2</sup>Marvin Minsky; one of the founders of AI

- ▷ extremely fast extremely reliable
- ▷ completely stupid: will not do what you mean unless you tell it exactly
- ▷ Programming can be extremely fun/frustrating/addictive (try it)
- ▷ **Definition 2.1.6** A **programming language** is the formal language in which we write **programs** (express an algorithm concretely)
  - ▷ formal, symbolic, precise meaning (a machine must understand it)
- ▷ There are lots of **programming languages**
  - ▷ design huge effort in computer science
  - ▷ all **programming languages** equally strong
  - ▷ each is more or less appropriate for a specific task depending on the circumstances
- ▷ Lots of paradigms: imperative, functional programming, logic programming, object oriented programming



AI studies human intelligence with the premise that the brain is a computational machine and that intelligence is a “**program**” running on it. In particular, the working hypothesis is that we can “program” intelligence. Even though AI has many successful applications, it has not succeeded in creating a machine that exhibits the equivalent to general human intelligence, so the jury is still out whether the AI hypothesis is true or not. In any case it is a fascinating area of scientific inquiry.

**Note:** This has an immediate consequence for the discussion in our course. Even though computers can execute **programs** very efficiently, you should not expect them to “think” like a human. In particular, they will execute **programs** exactly as you have written them. This has two consequences:

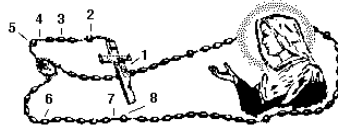
- the behavior of **programs** is – in principle – predictable
- all errors of **program** behavior are your own (the programmer’s)

In computer science, we distinguish two levels on which we can talk about **programs**. The more general is the level of **algorithms**, which is independent of the concrete **programming language**. **Algorithms** express the general ideas and flow of computation and can be realized in various languages, but are all equivalent – in terms of the **algorithms** they implement.

As they are not bound to **programming languages** **algorithms** transcend them, and we can find them in our daily lives, e.g. as sequences of instructions like recipes, game instructions, and the like. This should make algorithms quite familiar; the only difference of **programs** is that they are written down in an unambiguous syntax that a computer can understand.

## Program Execution

- ▷ **Algorithm:** informal description of what to do (good enough for humans)



**Program:** computer-processable version, e.g. in python

```
for x in range(0, 3):
    print ("we tell you",x,"time(s)")
```

▷

▷ **Interpreter:** reads a **program** and executes it directly

▷ special case: interactive interpretation (lets you experiment easily)

▷ **Compiler:** translates a **program** (the **source**) into another **program** (the **binary**) in a much simpler **programming language** for optimized execution on hardware directly.

▷ **Remark 2.1.7** **Compilers** are efficient, but more cumbersome for development.



We have two kinds of **programming languages**: one which the **CPU** can execute directly – these are very very difficult for humans to understand and maintain – and higher-level ones that are understandable by humans. If we want to use high-level languages – and we do, then we need to have some way bridging the language gap: this is what **compilers** and **interpreters** do.

## 2.1.2 Programming in IWGS

After the general introduction to “programming” in Chapter 2, we now instantiate the situation to the IWGS course, where we use **python** as the primary programming language.

### Programming in IWGS: python

▷ We will use python as the **programming language** in this course

▷ We cover just enough python, so that you

- ▷ understand the joy and principle of programming
- ▷ can play with objects we present in IWGS.

▷ After a general introduction we will introduce language features as we go along

▷ For more information on python (homework/preparation)

## RTFM ( $\hat{=}$ “read those fine manuals”)

**RTFM Resources:** There are also lots of good tutorials on the web,

- ▷ ▷ I like [LP; Sth; Swe13];
- ▷ but also see the language documentation [P3D].
- ▷ [Kar] is an introduction geared to the (digital) humanities



©: Michael Kohlhasse

20



**Note** that IWGS is not a programming course, which concentrates on teaching a programming language in all its gory detail. Instead we want to use the IWGS lecture to introduce the necessary concepts and use the tutorials to introduce additional language features based on these.

## But Seriously... Learning programming in IWGS

- ▷ The IWGS lecture teaches you
  - ▷ a general introduction to programming and python (next)
  - ▷ various useful concepts and how they can be done in python (in principle)
- ▷ The IWGS tutorials
  - ▷ teach the actual skill and joy of programming (hacking  $\neq$  security breach)
  - ▷ supply you with problems so you can practice that.

**Richard Stallman (MIT) on Hacking:** “What they had in common was mainly love of excellence and programming. They wanted to make their programs that they used be as good as they could. They also wanted to make them do neat things. They wanted to be able to do something in a more exciting way than anyone believed possible and show “Look how wonderful this is. I bet you didn’t believe this could be done.””

▷ So, ... : Let’s hack



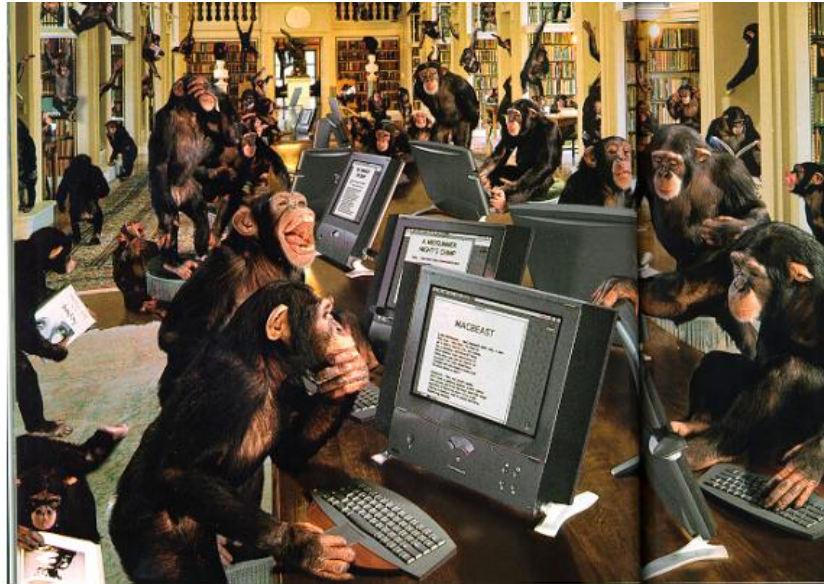
©: Michael Kohlhasse

21



However, the result would probably be the following:

⚠ 2am in the Kollegienhaus CIP Pool ⚠



©: Michael Kohlhasse

22



If we just start hacking before we fully understand the problem, chances are very good that we will waste time going down blind alleys, and garden paths, instead of attacking problems. So the main motto of this course is:

⚠ no, let's think ⚠

- ▷ We have to fully understand the problem, our tools, and the solution space first  
(That is what the IWGS lecture is for)
- ▷ read Richard Stallman's quote carefully  $\leadsto$  problem understanding is a crucial prerequisite for hacking.
- ▷ "The GIGO Principle: Garbage In, Garbage Out" (– ca. 1967)
- ▷ "Applets, Not Craplets<sup>tm</sup>" (– ca. 1997)



©: Michael Kohlhasse

23



## 2.2 Programming in Python

In this Section we will introduce the basics of the python language. python will be used as our means to express algorithms and to explore the computational properties of the objects we introduce in IWGS.

### 2.2.1 Hello IWGS

Before we get into the syntax and meaning of python, let us recap why we chose this particular language for IWGS.

## python in a Nutshell

### ▷ Why python?:

- ▷ general purpose **programming language**
- ▷ imperative, interactive **interpreter**



- ▷ syntax very easy to learn (spend more time on problem solving)
- ▷ scales well:
  - ▷ easy for beginners to write simple **programs**,
  - ▷ but advanced software can be written with it as well.

### ▷ Interactive mode: The python shell IDLE3

### ▷ For the eager (Optional): Establish a python **interpreter** (version 3.7) (not 2.?.?. that has different syntax)

- ▷ install python from <http://python.org> (for offline use)
- ▷ make sure (tick box) that the python executable is added to the path. (makes shell interaction much easier)



©: Michael Kohlhase

24



**Installing python:** python can be installed from <http://python.org> ~ “Downloads”, as a Windows installer or a Mac OS X disk image. For linux it is best installed via the package manager, e.g. using

```
sudo apt-get update
sudo apt-get install python3.7
```

The download will install the python **interpreter** and the python shell IDLE3 that can be used for interacting with the **interpreter** directly.

It is important that you make sure (tick the box in the Windows installer) that the python executable is added to the path. In the shell<sup>1</sup>, you can then use

EdN:1

```
python «filename»
```

to run the python file «filename». This is better than using the windows-specific

```
py «filename»
```

which does not need the python interpreter on the path as we will see later.

## Arithmetic Expressions in python

- ▷ Expressions are “**programs**” that compute values (here: numbers)

<sup>1</sup>EdNOTE: fully introduce the concept of a shell in the next round

▷ **Integers** (numbers without a decimal point)

- ▷ **operators**: addition +, subtraction −, multiplication \*, division /, integer division //, remainder/modulo %, ...
- ▷ Division yields a float

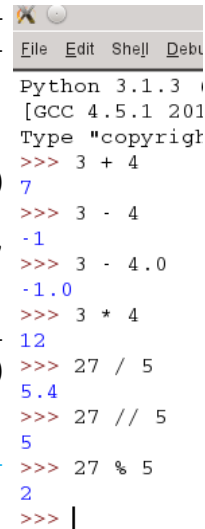
▷ **Floats** (numbers with a decimal point)

- ▷ **Operators**: integer below floor, integer above ceil, exponential exp, square root sqrt, ...

Numbers are **values**, i.e. data objects that can be computed with. (reference the last computed one with `_`)

- ▷ **Expressions** are created from **values** (and other **expressions**) via **operators**.

- ▷ **Observation**: The python **interpreter** simplifies **expressions** to **values** by computation.



```

Python 3.1.3
[GCC 4.5.1 201
Type "copyright
>>> 3 + 4
7
>>> 3 - 4
-1
>>> 3 - 4.0
-1.0
>>> 3 * 4
12
>>> 27 / 5
5.4
>>> 27 // 5
5
>>> 27 % 5
2
>>> |
  
```



In IWGS, we want to use the JupyterLab cloud service. This runs the python **interpreter** on a cloud server and gives you a browser window with a **web IDE**, which you can use for interacting with the **interpreter**. You will have to make an account there; details to follow.

## JupyterLab A Cloud IDE for python

- ▷ **For helping you** it would be good if the TAs could access to your code
- ▷ **Idea**: Use a **web IDE** (a web-based integrated development environment), which you can use for interacting with the **interpreter**.
- ▷ We will use JupyterLab for IWGS. (but you can also use python locally)
- ▷ **Homework**: Set up JupyterLab
  - ▷ make an account at <http://jupyter.kwarc.info>

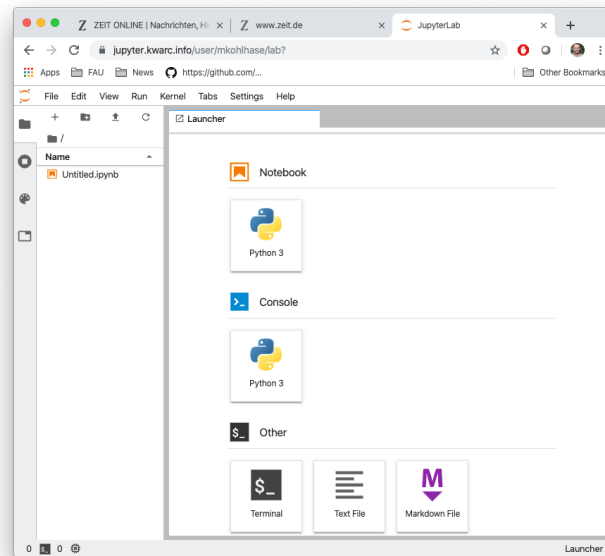


The advantage of a cloud IDE like JupyterLab for a course like IWGS is that you do not need any installation, cannot lose your files, and your teachers (the course instructor and the teaching assistants) can see (and even directly interact with) the your run time environment. This gives us a much more controlled setting and we can help you better.

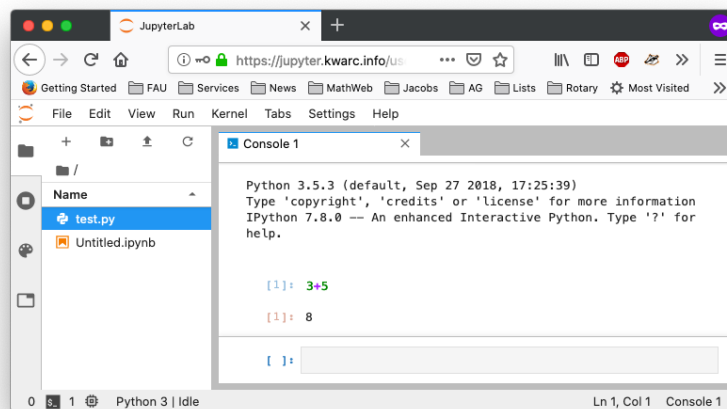
Both IDLE3 as well as JupyterLab come with an integrated editor for writing python programs. These editors gives you python syntax highlighting, and **interpreter** and debugger integration. In short, IDLE3 and JupyterLab are integrated development environments for python. Let us now go through the interface of the JupyterLab IDE.

## JupyterLab Components

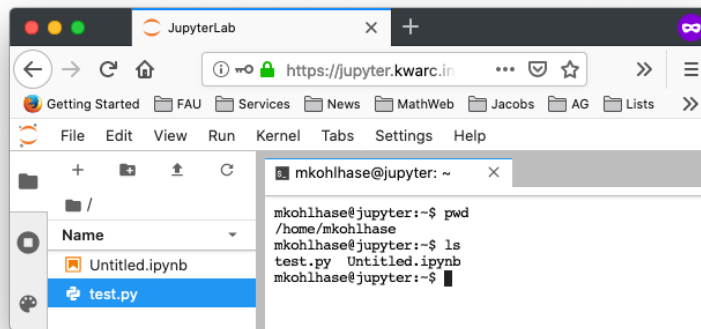
- ▷ The JupyterLab **dashboard** gives you access to all components



- ▷ The JupyterLab **python console**, i.e. a python **interpreter** in your browser. (use this for python interaction and testing)



- ▷ The JupyterLab **terminal**, i.e. a UNIX **shell** in your browser. (use this for managing files)



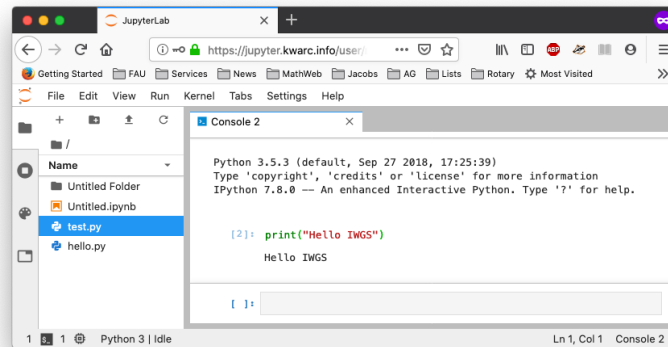
- ▷ **Definition 2.2.1** A **shell** is a **command-line interface** for accessing the **services** of a **computer's operating system**.
- ▷ **Useful shell commands:** See e.g. [All18] for a basic tutorial
- ▷ ls: “list” the files in this directory
  - ▷ mkdir: “make” folder (called “directory”)
  - ▷ pwd: “print working directory” (where am I)
  - ▷ cd  $\langle\langle\text{dirname}\rangle\rangle$ : “change directory”
    - ▷  $\langle\langle\text{dirname}\rangle\rangle = \dots$ : one up in the directory tree
    - ▷ empty  $\langle\langle\text{dirname}\rangle\rangle$ : go to your home directory.
  - ▷ rm  $\langle\langle\text{filename}\rangle\rangle$ , cp/mv  $\langle\langle\text{filename}\rangle\rangle \langle\langle\text{newname}\rangle\rangle / \langle\langle\text{dirname}\rangle\rangle$ : remove, copy, and move/rename
  - ▷ ...see [All18] for more ...



Now that we understand our tools, we can write our first program: Traditionally, this is a “hello-world program” (see [HWC] for a description and a list of hello world programs in hundreds of languages) which just prints the string “Hello World” to the console. For **python**, this is very simple as we can see below. We use this program to explain the concept of a program as a (text) file, which can be started from the console.

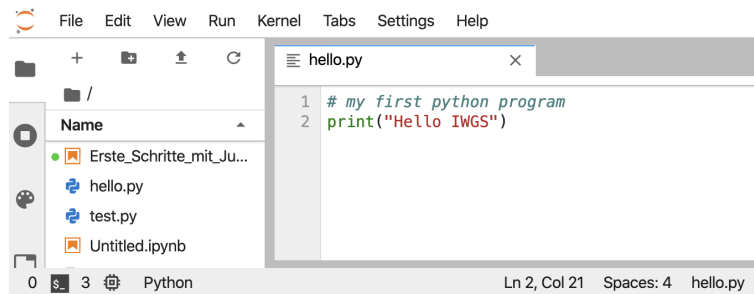
## A first program in python

- ▷ A classic “Hello World” program:  
start your **python console**, type **print("Hello IWGS")**. (print a string)



▷ Alternatively:

1. got to the JupyterLab [dashboard](#) select “Text File”,
2. Type your program,



3. Save the file as hello.py
4. Go to your [terminal](#) and type `python3 hello.py`
- 3' Alternatively: go to your [python console](#) and type `import hello` (in the same directory)



We have seen that we can just call a program from the [terminal](#), if we stored it in a file. In fact, we can do better: we can make our program behave like a native [shell](#) command.

1. The file extension `.py` is only used by convention, we can leave it out and simply call the file `hello`.
2. Then we can add a special python comment in the first [line](#)

```
#!/usr/bin/python3
```

which the [terminal](#) interprets as “call the program `python3` on me”.

3. Finally, we make the file `hello` executable, i.e. tell the [terminal](#) the file should behave like a shell command by issuing

```
chmod u+x hello
```

in the directory where the file `hello` is stored.

4. We add the [line](#)

```
export PATH="./:${PATH}"
```

to the file `.bashrc`. This tells the [terminal](#) where to look for programs (here the respective current directory called `.`)

With this simple recipe we could in principle extend the repertoire of instructions of the [terminal](#) and automate repetitive tasks.

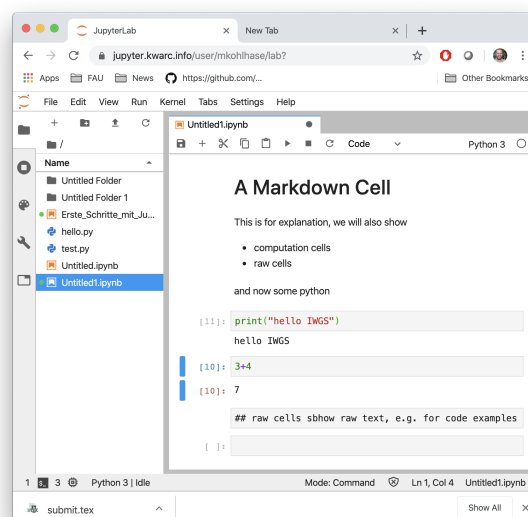
## jupyter Notebooks

- ▷ **Definition 2.2.2** [Jupyter notebooks](#) are documents that combine live runnable code with rich, narrative text (for comments and explanations).
- ▷ **Definition 2.2.3** [Jupyter notebooks](#) consist of [cells](#) which come in three forms
  - ▷ a [raw cell](#) shows text as is
  - ▷ a [markdown cell](#) interprets the contents as markdown text (later more)
  - ▷ a [code cell](#) interprets the contents as (e.g. `python`) code
- ▷ [Cells](#) can be executed by pressing “shift-enter” (Just “enter” gives a new line)
- ▷ **Idea:** [Jupyter notebooks](#) act as a [REPL](#), just as `IDLE3`, but allow
  - ▷ documentation in [raw](#) and [markdown cells](#)
  - ▷ changing and re-executing existing [cells](#).



## jupyter Notebooks

- ▷ **Example 2.2.4** (Showing off Cells in a Notebook)





Before we go on to learn more basic python operators and [instructions](#), we address an important general topic: comments in [program](#) code.

## Comments in python

- ▷ **Generally:** It is highly advisable to insert comments into your [programs](#),
  - ▷ especially, if others are going to read your code, (TAs/graders)
  - ▷ you may very well be one of the “others” yourself, (in a year’s time)
  - ▷ writing comments first helps you organize your thoughts.
- ▷ Comments are ignored by the python [interpreter](#) but are useful information for the programmer.
- ▷ **In python:** there are two kinds of comments
  - ▷ Single [line](#) comments start with a `#`
  - ▷ Multiline comments start and end with three quotes (single or double: `"""` or `'''`)
- ▷ **Idea:** Use comments to
  - ▷ specify what the intended input/output behavior of the [program](#) or fragment
  - ▷ give the idea of the algorithm achieves this behavior.
  - ▷ specify any assumptions about the context (do we need some file to exist)
  - ▷ document whether the [program](#) changes the context.
  - ▷ document any known limitations or errors in your code.



### 2.2.2 Variables and Types

And we start with a general feature of [programming languages](#): we can give names to [values](#) and use them multiple times. Conceptually, we are introducing shortcuts, and in reality, we are giving ourselves a way of storing [values](#) in [memory](#) so that we can reference them later.

## Variables in python

- ▷ **Idea:** [Values](#) (of [expressions](#)) can be given a name for later reference.
- ▷ **Definition 2.2.5** A [variable](#) is a [memory](#) location which contains a [value](#). It is referenced by an identifier – the [variable name](#).
- ▷ **note:** In python a [variable name](#)
  - ▷ must start with letter or `_`,
  - ▷ cannot be a python keyword

▷ is case-sensitive (foobar, FooBar, and fooBar are different variables)

▷ A **variable name** can be used in **expressions** everywhere its **value** could be.

▷ A **variable assignment** `⟨var⟩=⟨val⟩` assigns a **value**.

▷ **Example 2.2.6 (Playing with python Variables)**

```
>>> foot = 30.5
>>> inch = 2.54
>>> 6 * foot + 2 * inch
188.08
>>> 3 * Inch
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    3 * Inch
NameError: name 'Inch' is not defined
>>> |
```



Let us fortify our intuition about **variables** with some examples. The first shows that we sometimes need **variables** to store objects out of the way and the second one that we can use **variables** to assemble intermediate results.

## Variables in python: Extended Example

▷ **Example 2.2.7 (Swapping Variables)** To exchange the values of two **variables**, we have to cache the first in an auxiliary variable.

```
a = 45
b = 0
print("a =", a, "b =", b)
print("Swap the contents of a and b")
swap = a
a = b
b = swap
print("a =", a, "b =", b)
```

Here we see the first example of a **python** script, i.e. a series of **python** commands, that jointly perform an action (and communicates it to the user).

▷ **Example 2.2.8 (Variables for Storing Intermediate Variables)**

```
>>> x = "OhGott"
>>> y = x+x+x
>>> z = y+y+y
>>> z
'OhGottOhGottOhGottOhGottOhGottOhGottOhGottOhGottOhGott'
```



If we use **variables** to assemble intermediate results, we can use telling names to document what these intermediate objects are – something we did not do well in Example 2.2.8; but admittedly, the meaning of the objects in this contrived example is questionable.

The next phenomenon in **python** is also common to many (but not all) **programming languages**: **expressions** are classified by the kind of objects their **values** are. Objects can be simple (i.e. of a

basic **type**; python has five of these) or complex, i.e. composed of other objects; we will go into that below.

## Data Types in python

- ▷ **Recall**: python **programs** process data (**values**), which can be combined by **operators** and **variables** into **expressions**.
- ▷ **Data types** group data and tell the interpreter what to expect
  - ▷ 1, 2, 3, etc. are **data** of **type** "integer"
  - ▷ "hello" is **data** of **type** "string"
- ▷ **Data types** determine which operators can be applied
- ▷ In python, every **values** has a **type**, variables can have any **type**, but can only be assigned **values** of their **type**.
- ▷ **Definition 2.2.9** python has the following five basic **data types**

Data type	Keyword	contains	Examples
integers	<b>int</b>	bounded integers	1, -5, 0, ...
floats	<b>float</b>	floating point numbers	1.2, .125, -1.0, ...
strings	<b>str</b>	strings	"Hello", 'Hello', "123", 'a', ...
Booleans	<b>bool</b>	truth values	True, False
complexess	<b>complex</b>	complex numbers	2+3j, ...

- ▷ We will encounter more **types** later.



We will now see what we can – and cannot – do with **data types**, this becomes most noticeable in **variable assignments** which establishes a **type** for the variable (this cannot be change any more) and in the application of **operators** to **arguments** (which have to be of the correct **type**).

## Data Types in python (continued)

- ▷ The type of a **variable** is automatically determined in the first **variable assignment** (before that the variable is unbound)

```
>>> firstVariable = 23 # integer
>>> type(firstVariable)
<class 'int'>
weight = 3.45 # float
first = 'Hello' # str
```

**Hint**: The python function **type** to computes the **type** (don't worry about the **class** bit)



## ▷ Data Types in python (continued)

- ▷ **Observation 2.2.10** *python is strongly typed, i.e. types have to match*
- ▷ Use data type conversion functions `int()`, `float()`, `complex()`, `bool()`, and `str()` to adjust types

▷ **Example 2.2.11 (Type Errors and Type Coersion)**

```
>>> 3+"hello"
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    3+"hello"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> str(4)+"hello"
'4Hello'
```



### 2.2.3 Python Control Structures

So far, we only know how to make **programs** that are a simple sequence of **instructions** – no repetitions, no alternative pathways. Example 2.2.6 is a perfect example. We will now change that by introducing **control structures**, i.e. complex **program instructions** that change the **control flow** of the **program**.

#### Conditionals and Loops

- ▷ **Problem:** Up to now **programs** seem to execute all the **instructions** in sequence, from the first to the last (a **linear program**)
- ▷ **Definition 2.2.12** The **control flow** of a **program** is the sequence of execution of the **program instructions**. It is specified via special **program instructions** called **control structures**.
- ▷ **Definition 2.2.13** **Conditional execution** allows to execute (or not to execute) certain parts of a **program** (the **branches**) depending on a **condition**. We call a code block that enables **conditional execution** a **conditional statement**.
- ▷ **Definition 2.2.14** A **loop** is a **control structure** that allows to execute certain parts of a **program** (the **body**) multiple times depending on **conditions**.
- ▷ **Definition 2.2.15** A **condition** is a **Boolean expression** in a **control structure**.
- ▷ **Example 2.2.16** In python, **conditions** are constructed by applying a Boolean operator to arguments, e.g. `3>5`, `x==3`, `x!=3`, ...  
or by combining simpler conditions by Boolean connectives **or**, **and**, and **not** (using brackets if necessary), e.g. `x>5 or x<3`

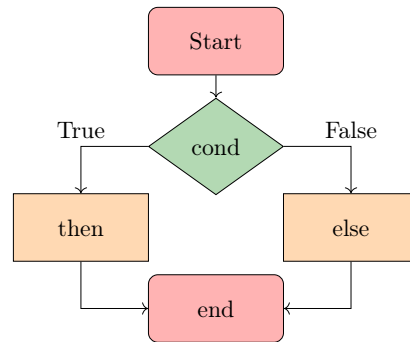
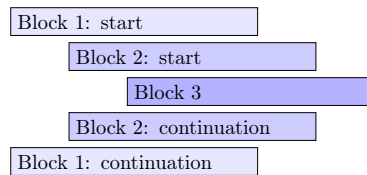


After this general introduction – **conditional execution** and **loops** are supported by all programming language in some form – we will see how this is realized in python

## Conditionals in python

▷ **Definition 2.2.17** Conditional execution via **if/else** statements

```
if <<condition>> :
    <<then-part>>
else :
    <<else-part>>
<<more code>>
```



- ▷ <<then-part>> and <<else-part>> have to be indented equally. (e.g. 4 blanks)
- ▷ If **control structures** are nested they need to be further indented consistently.



python uses indenting to signify nesting of body parts in control structures – and other structures as we will see later. This is a very un-typical syntactic choice in **programming languages**, which typically use brackets, braces, or other paired delimiters to indicate nesting and give the freedom of choice in indenting to programmers. This freedom is so ingrained in programming practice, that we emphasize the difference here. The following example shows **conditional execution** in action.

## Conditional Execution Example

▷ **Example 2.2.18 (Empathy in python)**

```
answer = input("Are you happy? ")
if answer == 'No' or answer == 'no':
    print("Have a chocolate!")
else:
    print("Good!")
print("Can I help you with something else?")
```

Note the indenting of the body parts.

- ▷ **BTW:** **input** is an operator that prints its argument string, waits for user input, and returns that.



But **conditional execution** in python has one more trick up its sleeve: what we can do with two branches, we can do with more as well.

## Variant: Multiple Branches

▷ making multiple **branches** is similar

```
if <<condition>> :
    <<then-part>>
```

```

elif ⟨⟨condition⟩⟩ :
    ⟨⟨other then-part⟩⟩
else :
    ⟨⟨else-part⟩⟩

```

- ▷ The there can be more than one **elif** clause.
- ▷ The ⟨⟨condition⟩⟩s are evaluated from top to bottom and the ⟨⟨then-part⟩⟩ of the first one that comes out true is executed. Then the whole **control structure** is exited.
- ▷ multiple **branches** could achieved by nested **if/else** structures.
- ▷ **Example 2.2.19 (Better Empathy in python)** In Example 2.2.18 we print Good! even if the input is e.g. I feel terrible, so extend **if/else** by
 

```

elif answer == 'Yes' or answer == 'yes' :
    print("Good!")
else :
    print("I do not understand your answer")

```



Note that the **elif** is just “syntactic sugar” that does not add anything new to the language: we could have expressed the same functionality as two nested if/else statements

```

if ⟨⟨condition⟩⟩ :
    ⟨⟨then-part⟩⟩
    if ⟨⟨condition⟩⟩ :
        ⟨⟨other then-part⟩⟩
    else :
        ⟨⟨else-part⟩⟩

```

But this would have introduced an additional layer of nesting (per **elif** clause in the original). The nested syntax also obscures the fact that all branches are essentially equal.

Now let us see the syntax for **loops** in python.

## Loops in python

- ▷ **Definition 2.2.20** python makes **loops** via **while**-blocks

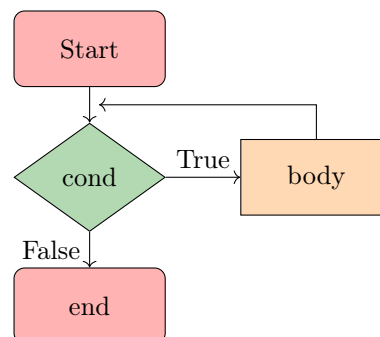
- ▷ syntax of the **while loop**

```

while ⟨⟨condition⟩⟩ :
    ⟨⟨body⟩⟩
    ⟨⟨more code⟩⟩

```

- ▷ breaking out of **loops** with **break**
- ▷ skipping the current **body** with **continue**
- ▷ ⟨⟨body⟩⟩ must be indented!



As always we will fortify our intuition with a couple of small examples.

## Examples of Loops

### ▷ Example 2.2.21 (Counting in python)

```
# Prints out 0,1,2,3,4
count = 0
while count < 5:
    print(count)
    count += 1 # This is the same as count = count + 1
```

This is the standard pattern for using **while**: using a loop variable (here `count`) and incrementing it in every pass through the loop.

### ▷ Example 2.2.22 (Breaking an unbounded Loop)

```
# Prints out 0,1,2,3,4 but uses break
count = 0
while True:
    print(count)
    count += 1
    if count >= 5:
        break
```



## Examples of Loops

### ▷ Example 2.2.23 (Exceptions in the Loop)

```
# Prints out only odd numbers – 1,3,5,7,9
count = 0
while count < 10
    count += 1
    # Check if x is even
    if count % 2 == 0:
        continue
    print(count)
```



Example 2.2.21 and Example 2.2.22 do the same thing: counting from zero to four, but using different mechanisms. This is normal in programming – there is not “one correct solution”. But the first solution is the “standard one”, and is preferred, since it is shorter and more readable. The **break** functionality shown off in the second one is still very useful. Take for instance the problem of computing the product of the numbers -10 to 1.000.000. The naive implementation of this is on the left below which does a lot of unnecessary work, because as soon as we passed 0, then the whole product must be zero. A more efficient implementation is on the right which breaks after seeing a zero.

Direct Implementation	More Efficient
<pre>count = -10 prod = 1 while count &lt; 1000000:     prod *= count     count += 1</pre>	<pre>count = -10 prod = 1 while count &lt;= 1000000:     prod *= count     if count = 0 :         break     count += 1</pre>

## 2.3 Some Thoughts about Computers and Programs

Finally, we want to go over a couple of general issues pertaining to **programs** and (universal) machines. We will just go over them to get the intuitions – which are central for understanding computer science – and let the lecture “Theoretical Computer Science” fill in the details and justifications.

### Computers as Universal Machines (a taste of theo. CS)

- ▷ **Observation:** **Computers** are **universal** tools: their behavior is determined by a **program**; they can do anything, the **program** specifies.
  - ▷ **Context:** Tools in most other disciplines are specific to particular tasks. (except in e.g. **ribosomes in cell biology**)
  - ▷ **Remark 2.3.1 (Deep Fundamental Result)** There are things no **computer** can compute.
  - ▷ **Example 2.3.2** whether another **program** will terminate in finite time.
  - ▷ **Remark 2.3.3 (Church-Turing Hypothesis)** There are two classes of languages
    - ▷ **Turing complete** (or **computationally universal**) ones that can compute what is theoretically possible.
    - ▷ **data languages** that cannot. (but describe data sets)
  - ▷ **Observation 2.3.4 (Turing Equivalence)** All **programming languages** are (made to be) **universal**, so they can compute exactly the same. (compilers/interpreters exist)
- ...in particular ...: Everybody who tells you that one **programming languages** is the best has no idea what they’re talking about (though differences in efficiency, convenience, and beauty exist)

- ▷ **Another Universal Tool:** The human mind. (We can understand/learn anything.)
- ▷ **Strong Artificial Intelligence:** claims that the brain is just another computer.
- ▷ **If that is true** then
  - ▷ the human mind underlies the same restrictions as computational machines
  - ▷ we may be able to find the “mind-program”.



We now come to one of the most important, but maybe least acknowledged principles of **programming languages**: The Principle of Compositionality. To fully understand it, we need to fix some fundamental vocabulary.

### Top Principle of Programming: Compositionality

- ▷ **Observation 2.3.5** Modern *programming languages* compose various *primitives* and give them a pleasing, concise, and uniform *syntax*.
- ▷ **Question:** What does all of this even mean?
- ▷ **Definition 2.3.6** In a *programming language*, a *primitive* is a “basic unit of processing”, i.e. the simplest element that can be given a procedural meaning (its *semantics*) of its own.
- ▷ **Definition 2.3.7 (Compositionality)** All *programming languages* provide *composition principles* that allow to *compose* smaller program fragments into larger ones in such a way, that the *semantics* of the larger is determined by the *semantics* of the smaller ones and that of the *composition principle* employed.
- ▷ **Observation 2.3.8** The *semantics* of a *programming language*, is determined by the meaning of its *primitives* and *composition principles*.
- ▷ **Definition 2.3.9** *Programming language syntax* describes the surface form of the program: the admissible character sequences. It is also a composition of the *syntax* for the *primitives*.



All of this is very abstract – it has to be as we have not fixed a programming language yet – and you will only understand the true impact of the compositionality principle over time and with programming experience. Let us now see what this means concretely for our course.

### Consequences of Compositionality

- ▷ **Observation 2.3.10** To understand a *programming language*, we (only) have to understand its *primitives*, *composition principles*, and their *syntax*.
- ▷ **Definition 2.3.11** The “art of *programming*” consists of *composing* the *primitives* of a *programming language*.
- ▷ **Observation 2.3.12** We only need very few – about half a dozen – *primitives* to obtain a *Turing complete programming language*.

- ▷ **Observation 2.3.13** *The space of program behaviors we can achieve by **programming** is infinitely large nonetheless.*
- ▷ **Remark 2.3.14** More **primitives** make **programming** more convenient.
- ▷ **Remark 2.3.15** **Primitives** in one language can be composed in others.



©: Michael Kohlhase

47



## A note on Programming: Little vs. Large Languages

- ▷ **Observation 2.3.16** *Most such concepts can be studied in isolations, and some can be given a syntax on their own. (standardization)*
- ▷ **Consequence:** If we understand the concepts and syntax of the sublanguages, then learning another **programming language** is relatively easy.



©: Michael Kohlhase

48



## 2.4 More about Python

After we have had some general thoughts about programming in general, we can get back to concrete python facilities and idioms. We will concentrate on those – there are lots and lots more – that are useful in IWGS.

### 2.4.1 Sequences and Iteration

We now come to a commonly used class of objects in **python**: sequences, such as **lists**, sets, tuples, **ranges**, and **dictionaries**.

They are used for storing, accumulating, and accessing objects in various ways in programs. They all have in common, that they can be used for **iteration**, thus creating a uniform interface to similar functionality.

### Lists in python

- ▷ **Definition 2.4.1** A **list** is a **finite sequence** of objects, its **elements**.
- ▷ In **programming languages**, **lists** are used for locally storing and passing around collections of objects.
- ▷ In python **lists** can be written as a sequence of comma-separated expressions between square brackets.
- ▷ **Definition 2.4.2** We call `[⟨seq⟩]` the **list constructor**.
- ▷ **Example 2.4.3 (Three lists)** elements can be of different **types** in python

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5];
```

```
list3 = ["a", "b", "c", "d"];
```

▷ **Example 2.4.4** List elements can be accessed by specifying ranges

```
>>> list1[0]    >>> list1[-2]    >>> list2[1:4]
'physics'       1997              [2, 3, 4]
```



©: Michael Kohlhase

49



As Example 2.4.4 shows, python treats counting in lists accessors somewhat peculiarly. It starts counting with zero when counting from the front and with one when counting from the back.

But lists are not the only things in python that can be accessed in the way shown in Example 2.4.4. python introduces a special class of types the sequence types.

### Sequences in python

▷ **Definition 2.4.5** python has more types that behave just like lists, they are called sequence types.

▷ The most important sequence types for IWGS are lists, strings and ranges.

▷ **Definition 2.4.6** A range is a finite sequence of numbers it can conveniently be constructed by the range function: range(⟨start⟩,⟨stop⟩,⟨step⟩) constructs a range from ⟨start⟩ to ⟨stop⟩ with step size ⟨step⟩.

▷ **Example 2.4.7** Lists can be constructed from ranges:

```
>>> list(range(1,6,2))
[1,3,5]
```

range(1,6,2) makes a “range” from 1 to 6 with step 2, list makes it a list.



©: Michael Kohlhase

50



Ranges are useful, because they are easily and flexibly constructed for iteration (up next).

### Iterating over Sequences in python

▷ **Definition 2.4.8** A for loop iterates a program fragment over a sequence; we call the process iteration. python uses the following general syntax

```
for ⟨var⟩ in ⟨range⟩:
    ⟨body⟩
⟨other code⟩
```

▷ **Example 2.4.9**

```
for x in range(0, 3):
    print ("we tell you",x,"time(s)")
```

▷ **Example 2.4.10** Lists and strings can also act as sequences. (try it)

```
print("Let me reverse something for you!")
```

```
x = input("please type something!")
for i in reversed(list(x)):
    print(i)
```



But [lists](#) are not the only data structure for collections of objects. python provides others that are organized slightly differently for different applications. We give a particularly useful example here: [dictionaries](#).

## python Dictionaries

- ▷ **Definition 2.4.11** A **dictionary** is an unordered, indexed collection of **ordered pairs**  $(k, v)$ , where we call  $k$  the **key** and  $v$  the **value**.
- ▷ In python **dictionaries** are written with curly brackets, pairs are separated by commas, and the **value** is separated from the **key** by a colon.
- ▷ **Example 2.4.12** **Dictionaries** can be used for various purposes,

painting = {	dict_de_en = {	enum = {
"artist": "Rembrandt",	"Maus": "mouse",	1: "copy",
"title": "The Night Watch",	"Ast": "branch",	2: "paste",
"year": 1642	"Klavier": "piano"	3: "adapt"
}	}	}

- ▷ **dictionaries** and **sequences** can be nested, e.g. for a **list** of paintings.



**Dictionaries** give “keyed access” to collections of data: we can access a **value** via its key. In particular, we do not have to remember the position of a value in the collection.

## Interacting with Dictionaries

- ▷ Dictionary commands by example
  - ▷ painting["title"] returns the **value** for the **key** "title" in the dictionary painting.
  - ▷ painting["title"]="De Nachtwacht" changes the **value** for the **key** "title" to its original Dutch  
(or adds item "title": "De Nachtwacht")

- ▷ **Example 2.4.13 (Printing Keys and Values)**

keys	values	items
<b>for x in thisdict:</b>	<b>for x in thisdict:</b>	<b>for x, y in thisdict.items():</b>
<b>print(x)</b>	<b>print(thisdict[x])</b>	<b>print(x, y)</b>

- ▷ more dictionary commands
  - ▷ **if <<key>> in <<dict>>** checks whether <<key>> is a **key** in <<dict>>.
  - ▷ painting.pop("title") removes the "title" item from painting.



### 2.4.2 Input and Output

The next topic of our stroll through python is one that is more practically useful than intrinsically interesting: file input/output. Together with the [regular expressions](#) this allows us to write programs that transform files.

#### Input/Output in python

- ▷ **Recall:** The [CPU](#) communicates with the user through [input](#) devices like keyboards and [output](#) devices like the screen.
- ▷ [programming languages](#) provide special [instructions](#) for this.
- ▷ In python we have already seen
  - ▷ `input(⟨⟨prompt⟩⟩)` for [input](#) from the keyboard, it returns a [string](#).
  - ▷ `print(⟨⟨objects⟩⟩, sep=⟨⟨separator⟩⟩, end=⟨⟨endchar⟩⟩)` for [output](#) to the screen.
- ▷ But computers also supply another object to [input](#) from and [output](#) to (up next)



We now fix some of the nomenclature surrounding [files](#) and [file systems](#) provided by most computer operating systems. Most programming languages provide their own bindings that allow to manipulate [files](#).

#### Secondary (Disk) Storage; Files, Folders, etc.

- ▷ **Definition 2.4.14** A [file](#) is a resource for recording data in a [storage device](#).
- ▷ **Definition 2.4.15** [Files](#) are identified by a [file name](#) are managed by a [file system](#) which organize them hierarchically into named [folders](#) and locate them by a [path](#); a sequence of [folder names](#). The [file name](#) and the [path](#) together fully identify a [file](#).  
 A [file name](#) usually consists of a [base name](#) and an [extension](#) separated by a dot character.
- ▷ Some [file systems](#) restrict the characters allowed in the [file name](#) and/or lengths of the [base name](#) or [extension](#).
- ▷ **Definition 2.4.16** Once a [file](#) has been [opened](#), the [CPU](#) can [write](#) to it and [read](#) from it. After use a file should be [closed](#) to protect it from accidental [reads](#) and [writes](#).



Many operating systems use files as a primary computational metaphor, also treating other resources like [files](#). This leads to an abstraction of files called [streams](#), which encompass [files](#) as

well as e.g. keyboards, printers, and the screen, which are seen as objects that can be read from (keyboards) and written to (e.g. screens). This practice allows flexible use of [programs](#), e.g. re-directing a the (screen) output of a [program](#) to a [file](#) by simply changing the output [stream](#).

Now we can come to the python bindings for the [file](#) input/output operations. They are rather straightforward.

## Disk Input/Output in python

- ▷ In python we have special [instructions](#) for dealing with files:
  - ▷ `open(⟨⟨path⟩⟩,⟨⟨iospec⟩⟩)` returns a file object `f`; `⟨⟨iospec⟩⟩` is one of `r` ([read](#) only; the default), `a` ([append](#)  $\hat{=}$  [write](#) to the end), and `r+` ([read/write](#)).
  - ▷ `f.read()` [reads](#) the file `f` into a [string](#).
  - ▷ `f.readline()` reads a single [line](#) from the file (including the newline character `\n`) otherwise returns the empty string `''`.
  - ▷ `f.write(⟨⟨str⟩⟩)` appends the [string](#) `⟨⟨str⟩⟩` to the end of `f`, returns the number of characters written.
  - ▷ `f.close()` closes `f` to protect it from accidental [reads](#) and [writes](#).

### ▷ Example 2.4.17 (Duplicating the contents of a file)

```
f = open('workfile','r+')
filecontents = f.read()
f.write(filecontents)
```



## Disk Input/Output in python (continued)

### ▷ Example 2.4.18 (Reading a file linewise)

<pre>&gt;&gt;&gt; f.readline() 'This is the first line of the file.\n' &gt;&gt;&gt; f.readline() 'Second line of the file\n' &gt;&gt;&gt; f.readline() ''</pre>	<pre>&gt;&gt;&gt; for line in f: ...     print(line, end='') ... This is the first line of the file. Second line of the file</pre>
---	--

- ▷ If you want to read all the lines of a file in a list you can also use `list(f)` or `f.readlines()`.
- ▷ For [reading](#) a python file we use the `import(⟨⟨basename⟩⟩)` [instruction](#)
  - ▷ it searches for the file `⟨⟨basename⟩⟩.py`, loads it, interprets it as python code, and directly executes it.
  - ▷ primarily used for loading python modules (additional functionality)
  - ▷ useful for loading python-encoded data (e.g. dictionaries)



### 2.4.3 Functions and Libraries in Python

We now come to a general device for organizing and modularizing code provided by most [programming languages](#), including python. Like [variables](#), [functions](#) give names to python objects – here fragments of code – and thus make them reusable in other contexts.

#### Functions in python (Introduction)

- ▷ **Observation:** sometimes programming tasks are repetitive

```
print("Hello Peter, how are you today? How about some IWGS?")
print("Hello Roxana, how are you today? How about some IWGS?")
print("Hello Frodo, how are you today? How about some IWGS?")
...
```

- ▷ **Idea:** We can automate the repetitive part by functions

- ▷ **Example 2.4.19**

```
def greet (who):
    print("Hello ",who," how are you today? How about some IWGS?")
greet("Peter")
greet("Roxana")
greet("Frodo")
greet(input ("Who are you?"))
...
```

- ▷ Functions can be a very powerful tool for structuring and documenting [programs](#) (if used correctly)



#### Functions in python (Example)

- ▷ **Example 2.4.20 (Multilingual Greeting)** Given a value for lang

```
def greet (who):
    if lang == 'en' :
        print("Hello ",who," how are you today? How about some IWGS?")
    elif lang == 'de' :
        print("Sehr geehrter ",who," , wie geht's heute? Wie waere es mit IWGS?")
```

we can even localize (i.e. adapt to the language specified in lang) the greeting.



We can now make the intuitions above formal and give the exact python syntax of [functions](#).

#### Functions in python (Definition)


- ▷ **Definition 2.4.21** A python [function](#) is defined by a code snippet of the form

```
def f (p1, ..., pn):
    """docstring, what does this function do on parameters
```

```

    :param  $p_i$ : document arguments}
    """
    «body» # it can contain  $p_1, \dots, p_n$ , and even  $f$ 
    return «value» # value of the function call (e.g text or number)
«more code»

```

- ▷ the indented part is called the **body** of  $f$ , (: whitespace matters in python)
- ▷ the  $p_i$  are called **parameters**, and  $n$  the **arity** of  $f$ .

A function  $f$  can be **called** on **arguments**  $a_1, \dots, a_n$  by writing the expression  $f(a_1, \dots, a_n)$ . This executes the body of  $f$  where the (formal) parameters  $p_i$  are replaced by the arguments  $a_i$ .



We now come to a peculiarity of an object-oriented language like **python**: it treats types as first-class entities, which can be defined by the user – they are called **classes** then. We will not go into that here, since we will not need **classes** in IWGS, but have to briefly talk about **methods**, which are essentially functions, but have a special notation.

**python** provides two kinds of function-like facilities: regular **functions** as discussed above and **methods**, which come with **python** classes. We will not attempt a presentation of object-oriented programming and its particular implementation in **python** – this would be beyond the mandate of the IWGS course – but give a brief introduction that is sufficient to use **methods**.

## Functions vs. Methods in python

- ▷ There is another mechanism that is similar to **functions** in python. (we briefly introduce it here to delineate)
- ▷ **Background**: Actually, the **types** from Definition 2.2.9 are **classes**, ...
- ▷ **Definition 2.4.22** In python all **values** belong to a **class**, which provide special **functions** we call **methods**. **Values** are also called **objects**, to emphasise **class** aspects. **Method** application is written with **dot notation**:  $\langle\text{obj}\rangle.\langle\text{meth}\rangle(\langle\text{args}\rangle)$  corresponds to  $\langle\text{meth}\rangle(\langle\text{obj}\rangle, \langle\text{args}\rangle)$ .
- ▷ **Example 2.4.23** Finding the position of a substring

```

>>> s = 'This is a Python string' # s is an object of class 'str'
>>> type(s)
<class 'str'> # see, I told you so
>>> s.index('Python') # dot notation (index is a string method)
10

```



## Functions vs. Methods in python

- ▷ **Example 2.4.24** (Functions vs. Methods)

```
>>> sorted('1376254') # no dots!
['1', '2', '3', '4', '5', '6', '7']

>>> mylist = [3, 1, 2]
>>> mylist.sort() # dot notation
>>> mylist
[1, 2, 3]
```

**Intuition:** only **methods** can change objects, functions return changed copies



©: Michael Kohlhasse

62



For the purposes of IWGS, it is sufficient to remember that **methods** are a special kind of **functions** that employ the **dot notation**. They are provided by the **class** of an **object**.

It is very natural to want to share successful and useful code with others, be it collaborators in a larger project or company, or the respective community at large. Given what we have learned so far this is easy to do: we write up the code in a (collection of) **python** files, and make them available for download. Then others can simply load them via the **import** command.

### python Libraries

- ▷ **Idea:** **Functions**, **classes**, and **methods** are re-usable, so why not package them up for others to use.
- ▷ **Definition 2.4.25** A python **library** is a python file with a collection of **functions**, **classes**, and **methods**. It can be loaded via the **import** command.
- ▷ There are  $\geq 150.000$  libraries for python ( $\hat{=}$  **packages on <http://pypi.org>**)
  - ▷ search for them at <http://pypi.org> (e.g. 815 packages for “music”)
  - ▷ install them with `pip install «package-name»`
  - ▷ look at how they were done (all have links to source code)
  - ▷ maybe even contribute back (report issues, improve code, ...) (**open source**)



©: Michael Kohlhasse

63



The **python** community is an **open source** community, therefore many developers organize their code into libraries and license them under **open source licenses**.

Software repositories like PyPI (the **python** Package Index) collect (references to) and make them for the package manager **pip**, a **program** that downloads **python** libraries and installs them on the local machine where the **import** command can find them.

#### 2.4.4 A Final word on Programming in IWGS

This leaves us with a final word on the way we will handle programming in this course: IWGS is not a programming course, and we expect you to pick up **python** from the IWGS and web/book resources.

In this Subsection we will introduce the basics of the **python** language. **python** will be used as our means to express algorithms and to explore the computational properties of the objects we introduce in IWGS.

For more information on python

RTFM ( $\hat{=}$  “read the fine manuals”)



©: Michael Kohlhase

64



Our very quick introduction to python is intended to present the very basics of programming and get students off the ground, so that they can start using programs as tools for the humanities and social sciences.

But there is a lot more to the core functionality python than our very quick introduction showed, and on top of that there is a wealth of specialized packages and libraries for almost all computational and practical needs.

## Chapter 3

# Numbers, Characters, and Strings

In our basic introduction to programming above we have convinced ourselves that we need some basic objects to compute with, e.g. Boolean values for conditionals, numbers to calculate with, and characters to form strings for input and output. In this section we will look at how these are represented in the computer, which in principle can only store binary digits – voltage or no voltage on a wire – which we think of as 1 and 0.

In this Chapter we look at the representation of the basic data types of programming languages (numbers and characters) in the computer; Boolean values (“True” and “False”) can directly be encoded as binary digits.

### Documents as Digital Objects

- ▷ **Question:** how do texts get onto the computer? (after all, computers can only do 0/1)
- ▷ **Hint:** At the most basic level, texts are just sequences of characters.
- ▷ **Answer:** We have to encode characters as sequences of bits.
- ▷ **We will go into how:**
  - ▷ documents are represented as sequences of characters
  - ▷ characters are represented as numbers
  - ▷ numbers are represented as bits (0/1)



©: Michael Kohlhase

65



### 3.1 Representing and Manipulating Numbers

We start with the representation of numbers. There are multiple number systems, as we are interested in the principles only, we restrict ourselves to the natural numbers – all other number systems can be built on top of these. But even there we have choices about representation, which influence the space we need and how we compute with natural numbers.

The first system for number representations is very simple; so simple in fact that it has been discovered and used a long time ago.



**Problem:** For realistic arithmetics we need better number representations than the unary natural numbers (e.g. for representing the number of EU citizens  $\hat{=} 100\,000$  pages of /)



The unary natural numbers are very simple and direct, but they are neither space-efficient, nor easy to manipulate. Therefore we will use different ways of representing numbers in practice.

## ▷ Positional Number Systems

▷ **Problem:** Find a better representation system for natural numbers.

▷ **Idea:** build a clever code on the unary numbers, use position information and addition, multiplication, and exponentiation.

▷ **Definition 3.1.3** A **positional number system**  $\mathcal{N}$  is a pair  $\mathcal{N} = \langle D_b, \varphi_b \rangle$  with

▷  $D_b$  is a finite alphabet of  $b$  **digits**.  $b$  is called the **base** or **radix** of  $\mathcal{N}$

▷ assign each digit  $d \in D_b$  a number  $\varphi_b(d)$  between 0 and  $b - 1$ .

▷ Extend  $\varphi_b$  to **sequences of digits** by  $\varphi_b(\langle n_k, \dots, n_1 \rangle) := \sum_{i=1}^k \varphi_b(n_i) \cdot b^{i-1}$

▷ **Example 3.1.4**  $\langle \{a, b, c\}, \varphi \rangle$  with  $\varphi(a) := 0$ ,  $\varphi(b) := 1$ , and  $\varphi(c) := 2$  is a **positional number system** for base three. We have

$$\varphi(\langle c, a, b \rangle) = 2 \cdot 3^2 + 0 \cdot 3^1 + 1 \cdot 3^0 = 18 + 0 + 1 = 19$$

▷ **Observation 3.1.5** To convert a number  $n$  to base  $b$ , use successive integer division (division with remainder) by  $b$ :

$i := n$ ; **repeat** (record  $i \bmod b$ ,  $i := i \operatorname{div} b$ ) **until**  $i = 0$ .

▷ **Example 3.1.6 (Convert 456 to base 8)** Result:  $710_8$

$$\begin{array}{ll} 456 \operatorname{div} 8 = 57 & 456 \bmod 8 = 0 \\ 57 \operatorname{div} 8 = 7 & 57 \bmod 8 = 1 \\ 7 \operatorname{div} 8 = 0 & 7 \bmod 8 = 7 \end{array}$$



The problem with the unary number system is that it uses enormous amounts of space, when writing down large numbers. We obviously need a better encoding.

If we look at the unary number system from a greater distance, we see that we are not using a very important feature of strings here: position. As we only have one letter in our alphabet (/), we cannot, so we should use a larger alphabet. The main idea behind a positional number system  $\mathcal{N} = \langle D_b, \varphi_b \rangle$  is that we encode numbers as strings of digit in  $D_b$ , such that the position matters, and to give these encoding a meaning by mapping them into the unary natural numbers via a mapping  $\varphi_b$ . This is the same process we did for the logics; we are now doing it for number systems. However, here, we also want to ensure that the meaning mapping  $\varphi_b$  is a bijection, since we want to define the arithmetics on the encodings by reference to The arithmetical operators on the unary natural numbers.

## Commonly Used Positional Number Systems

▷ **Definition 3.1.7** The following positional number systems are in common use.

name	set	base	digits	example
unary	$\mathbb{N}_1$	1	/	////// <sub>1</sub>
binary	$\mathbb{N}_2$	2	0,1	0101000111 <sub>2</sub>
octal	$\mathbb{N}_8$	8	0,1,...,7	63027 <sub>8</sub>
decimal	$\mathbb{N}_{10}$	10	0,1,...,9	162098 <sub>10</sub> or 162098
hexadecimal	$\mathbb{N}_{16}$	16	0,1,...,9,A,...,F	FF3A12 <sub>16</sub>

▷ **Notation 3.1.8** attach the base of  $\mathcal{N}$  to every number from  $\mathcal{N}$ . (default: decimal)

**Trick:** Group triples or quadruples of binary digits into recognizable chunks (add leading zeros as needed)

$$\begin{aligned}
 \triangleright \triangleright 110001101011100_2 &= \underbrace{0110_2}_{6_{16}} \underbrace{0011_2}_{3_{16}} \underbrace{0101_2}_{5_{16}} \underbrace{1100_2}_{C_{16}} = 635C_{16} \\
 \triangleright 110001101011100_2 &= \underbrace{110_2}_{6_8} \underbrace{001_2}_{1_8} \underbrace{101_2}_{5_8} \underbrace{011_2}_{3_8} \underbrace{100_2}_{4_8} = 61534_8 \\
 \triangleright F3A_{16} &= \underbrace{F_{16}}_{1111_2} \underbrace{3_{16}}_{0011_2} \underbrace{A_{16}}_{1010_2} = 111100111010_2, \quad 4721_8 = \underbrace{4_8}_{100_2} \underbrace{7_8}_{111_2} \underbrace{2_8}_{010_2} \underbrace{1_8}_{001_2} = 100111010001_2
 \end{aligned}$$



We have all seen positional number systems: our decimal system is one (for the base 10). Other systems that important for us are the binary system (it is the smallest non-degenerate one) and the octal- (base 8) and hexadecimal- (base 16) systems. These come from the fact that binary numbers are very hard for humans to scan. Therefore it became customary to group three or four digits together and introduce we (compound) digits for them. The octal system is mostly relevant for historic reasons, the hexadecimal system is in widespread use as syntactic sugar for binary numbers, which form the basis for circuits, since binary digits can be represented physically by current/no current.

## Arithmetics in Positional Number Systems

▷ For arithmetics just follow elementary school rules (for the right base)

▷ Tom Lehrer's "New Math"

▷ **Example 3.1.9**

Addition base 4

$$\begin{array}{r}
 \phantom{+} \phantom{1} \phantom{2} \phantom{3} \\
 \phantom{+} 1 \phantom{2} \phantom{3} \\
 + 1_1 \phantom{2}_1 3 \\
 \hline
 3 \phantom{1} 1 \phantom{2}
 \end{array}$$

binary multiplication

$$\begin{array}{r}
 \phantom{*} \phantom{1} \phantom{0} \phantom{1} \phantom{0} \\
 \phantom{*} 1 \phantom{0} \phantom{1} \phantom{0} \\
 * \phantom{1} \phantom{0} \phantom{1} \phantom{0} \\
 \hline
 \phantom{*} 0 \phantom{0} \phantom{0} \phantom{0} \\
 \phantom{*} 1 \phantom{0} \phantom{1} \phantom{0} \\
 \phantom{*} 1 \phantom{0} \phantom{1} \phantom{0} \\
 \hline
 1 \phantom{0} \phantom{1} \phantom{0} \\
 1 \phantom{0} \phantom{1} \phantom{0} \\
 \hline
 1 \phantom{0} \phantom{1} \phantom{0} \phantom{0}
 \end{array}$$



## 3.2 Characters and their Encodings: ASCII and UniCode

IT systems need to encode characters from our alphabets as bit strings (sequences of binary digits (bits) 0 and 1) for representation in computers. To understand the current state – the unicode standard – we will take a historical perspective.

It is important to understand that encoding and decoding of characters is an activity that requires standardization in multi-device settings – be it sending a file to the printer or sending an e-mail to a friend on another continent. Concretely, the recipient wants to use the same character mapping for decoding the sequence of bits as the sender used for encoding them – otherwise the message is garbled.

We observe that we cannot just specify the encoding table in the transmitted document itself, (that information would have to be en/decoded with the other content), so we need to rely document-external external methods like standardization or encoding negotiation at the meta-level. In this Section we will focus on the former.

The ASCII code we will introduce here is one of the first standardized and widely used character encodings for a complete alphabet. It is still widely used today. The code tries to strike a balance between a being able to encode a large set of characters and the representational capabilities in the time of punch cards (see below).

### The ASCII Character Code

- ▷ **Definition 3.2.1** The **American Standard Code for Information Interchange** (ASCII) is a **character code** that assigns characters to numbers 0-127

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1...	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2...		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

The first 32 characters are control characters for ASCII devices like printers

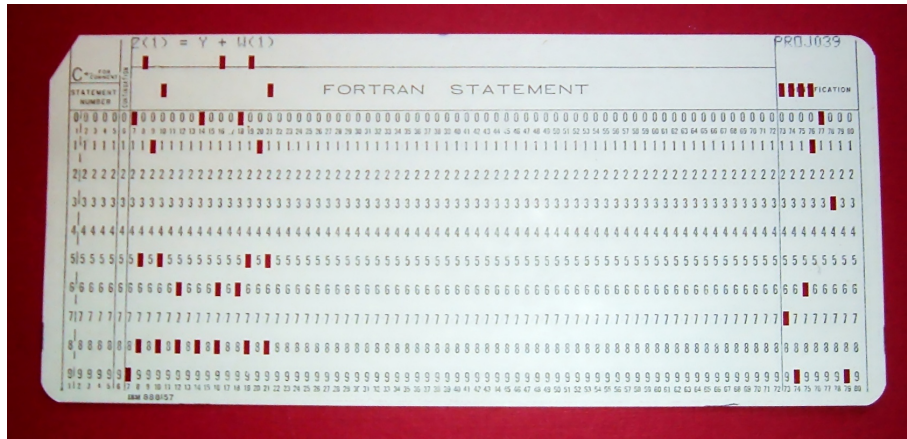
- ▷ **Motivated by punchcards:** The character 0 (binary 0000000) carries no information (used as dividers)  
 NUL,  
 Character 127 (binary 1111111) can be used for deleting (overwriting) last value (cannot delete holes)
- ▷ The ASCII code was standardized in 1963 and is still prevalent in computers today (but seen as US-centric)



Punch cards were the preferred medium for long-term storage of programs up to the late 1970s, since they could directly be produced by card punchers and automatically read by computers.

## A Punchcard

- ▷ A **punch card** is a piece of stiff paper that contains digital information represented by the presence or absence of holes in predefined positions.
- ▷ **Example 3.2.2** This punch card encoded the FORTRAN statement  $Z(1) = Y + W(1)$



Up to the 1970s, computers were batch machines, where the programmer delivered the program to the operator (a person behind a counter who fed the programs to the computer) and collected the printouts the next morning. Essentially, each punch card represented a single **line** (80 characters) of program code. Direct interaction with a computer is a relatively young mode of operation.

The ASCII code as above has a variety of problems, for instance that the control characters are mostly no longer in use, the code is lacking many characters of languages other than the English language it was developed for, and finally, it only uses seven bits, where a byte (eight bits) is the preferred unit in information technology. Therefore there have been a whole zoo of extensions, which — due to the fact that there were so many of them — never quite solved the encoding problem.

## Problems with ASCII encoding

- ▷ **Problem:** Many of the control characters are obsolete by now (e.g. NUL, BEL, or DEL)
- ▷ **Problem:** Many European characters are not represented (e.g. è, ñ, ü, ß, ...)
- ▷ **European ASCII Variants:** Exchange less-used characters for national ones
- ▷ **Example 3.2.3 (German ASCII)** remap e.g. [  $\mapsto$  Ä, ]  $\mapsto$  Ü in German ASCII  
("Apple "] comes out as "Apple Ü")
- ▷ **Definition 3.2.4 (ISO-Latin (ISO/IEC 8859))** 16 Extensions of ASCII to 8-bit (256 characters) ISO-Latin 1  $\hat{=}$  "Western European", ISO-Latin 6  $\hat{=}$  "Arabic", ISO-Latin 7  $\hat{=}$  "Greek"...

- ▷ **Problem:** No cursive Arabic, Asian, African, Old Icelandic Runes, Math,...
- ▷ **Idea:** Do something totally different to include all the world's scripts: For a scalable architecture, separate
  - ▷ what characters are available from the (character set)
  - ▷ bit string-to-character mapping (character encoding)



©: Michael Kohlhase

73



The goal of the UniCode standard is to cover all the worlds scripts (past, present, and future) and provide efficient encodings for them. The only scripts in regular use that are currently excluded are fictional scripts like the elvish scripts from the Lord of the Rings or Klingon scripts from the Star Trek series.

An important idea behind UniCode is to separate concerns between standardizing the character set — i.e. the set of encodable characters and the encoding itself.

### Unicode and the Universal Character Set

- ▷ **Definition 3.2.5 (Twin Standards)** A scalable architecture for representing all the worlds scripts
  - ▷ The **universal character set (UCS)** defined by the ISO/IEC 10646 International Standard, is a standard set of **characters** upon which many character encodings are based.
  - ▷ The **unicode Standard** defines a set of standard character encodings, rules for normalization, decomposition, collation, rendering and bidirectional display order
- ▷ **Definition 3.2.6** Each **UCS character** is identified by an unambiguous name and an integer number called its **code point**.
- ▷ The **UCS** has 1.1 million code points and nearly 100 000 characters.
- ▷ **Definition 3.2.7** Most (non-Chinese) characters have code points in [1, 65536] (the **basic multilingual plane**).
- ▷ **Notation 3.2.8** For code points in the Basic Multilingual Plane (BMP), four **hexadecimal** digits are used, e.g. U+ 0058 for the character LATIN CAPITAL LETTER X;



©: Michael Kohlhase

74



Note that there is indeed an issue with space-efficient encoding here. UniCode reserves space for  $2^{32}$  (more than a million) characters to be able to handle future scripts. But just simply using 32 bits for every UniCode character would be extremely wasteful: UniCode-encoded versions of ASCII files would be four times as large.

Therefore UniCode allows multiple encodings. UTF-32 is a simple 32-bit code that directly uses the code points in binary form. UTF-8 is optimized for western languages and coincides with the ASCII where they overlap. As a consequence, ASCII encoded texts can be decoded in UTF-8 without changes — but in the UTF-8 encoding, we can also address all other UniCode characters (using multi-byte characters).

## Character Encodings in Unicode

▷ **Definition 3.2.9** A **character encoding** is a mapping from bit strings to UCS code points.

▷ **Idea:** Unicode supports multiple encodings (but not character sets) for efficiency

▷ **Definition 3.2.10 (Unicode Transformation Format)**

▷ UTF-8, 8-bit, variable-width encoding, which maximizes compatibility with ASCII.

▷ UTF-16, 16-bit, variable-width encoding (popular in Asia)

▷ UTF-32, a 32-bit, fixed-width encoding (for safety)

▷ **Definition 3.2.11** The UTF-8 encoding follows the following encoding scheme

Unicode	byte 1	byte 2	byte 3	byte 4
U+000000 – U+00007F	0xxxxxxx			
U+000080 – U+0007FF	110xxxxx	10xxxxxx		
U+000800 – U+00FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+010000 – U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

▷ **Example 3.2.12** \$ = U+0024 is encoded as 00100100 (1 byte)

ç = U+00A2 is encoded as 11000010,10100010 (two bytes)

€ = U+20AC is encoded as 11100010,10000010,10101100 (three bytes)



Note how the fixed bit prefixes in the encoding are engineered to determine which of the four cases apply, so that UTF-8 encoded documents can be safely decoded.

## XKCD's Take on Recent Unicode Extensions

▷ Unicode 6.0 adopted hundreds of emoji characters in 2010 (2666 in July 2017)

▷ Modifying Characters (<https://xkcd.com/1813/>)

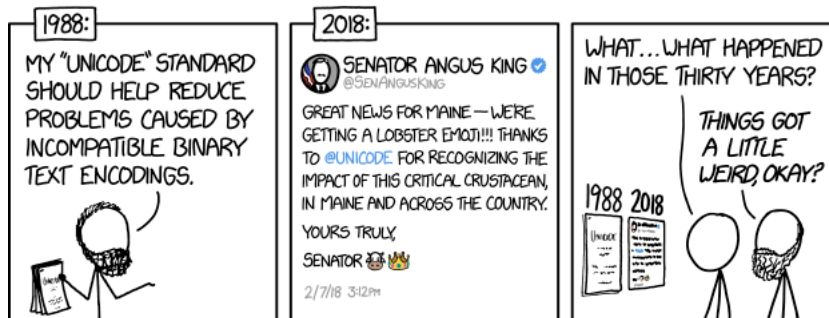




## XKCD's Take on Recent Unicode Extensions (cont.)

▷ Recent Unicode Extensions

(<https://xkcd.com/1953/>)



## 3.3 More on Computing with Strings

We now extend our repertoire on handling and formatting strings in `python`: we will introduce [string literals](#), which allow writing complex strings.

### Playing with Strings and Characters in python

▷ **Definition 3.3.1** python `strings` are sequences of UniCode `characters`.

▷ `△`: in python, characters are just strings of length 1.

▷ `ord` gives the `UCS code point` of the character, `chr character` for a number.

▷ **Example 3.3.2 (Playing with Characters)**

```
def lc(c) :
    return chr(ord(c) + 32)
def uc(c) :
    return chr(ord(c) - 32)
>>> uc('d')
'D'
>>> lc('D')
'd'
```

▷ strings can be accessed by ranges `[i:j]` (`[i] ≡ [i:i]`)

▷ **Example 3.3.3** taking strings apart and re-assembling them.

```
def cap(s) :
    return uc(s[0]) + cap(s[1:len(s)])
```





Now that we understand the “theory” of encodings, let us work out how to program with them in python:

Programming with UniCode strings is particularly simple, strings in `python` are UTF-8-encoded UniCode strings and all operations on them are UniCode-based<sup>1</sup>. This makes the introduction to UniCode in `python` very short, we only have to know how to produce non-ASCII characters, i.e. the characters that are not on regular keyboards.

If we know the code point, this is very simple: we just use UniCode [escape sequences](#).

### Unicode in python

▷ **Remark 3.3.6** The python [string data type](#) is UniCode encoded as UTF-8.

▷ **How to write UniCode characters?:** there are five ways

- ▷ write them in your editor (make sure that it uses UTF-8)
- ▷ otherwise use python escape sequences (try it!)

```
>>> "\xa3" # Using 8-bit hex value
'\u00A3'
>>> "\u00A3" # Using a 16-bit hex value
'\u00A3'
>>> "\U000000A3" # Using a 32-bit hex value
'\u00A3'
>>> "\N{Pound Sign}" # character name
'\u00A3'
```



[String literals](#) are convenient for creating simple strings. For more complex ones, we usually want to build them from pieces, usually using the values of variables or the results of functions. This is what [f-strings](#) are for in python; we will cover that now.

### Formatted String Literals (aka. f-strings)

▷ **Definition 3.3.7** [Formatted string literals](#) (aka. [f-strings](#)) are [string literals](#) can contain python expressions that will be replaced with their values at runtime.

[F-strings](#) are prefixed by a prefix `f` or `F`, the expressions are delimited by curly braces, and `{` and `}` are represented by `{{` and `}}`.

▷ **Example 3.3.8 (An f-String for IWGS)**

```
>>> course="IWGS"
>>> f"The {course} course has {6*11} students"
'The IWGS course has 66 students'
```

▷ **Example 3.3.9 (An f-String with Dictionary)**

```
>>> course = {'name':"IWGS",'students':'66'}
>>> f"The {course['name']}" course has {course['students']} students."
'The IWGS course has 66 students.'
```

<sup>1</sup>Older [programming languages](#) have ASCII strings only, and UniCode strings are supplied by external modules.

Note that we alternated the quotes here to avoid the following problems:

```
>>> f'The course {course['name']} has {course['students']} students.'
File "<stdin>", line 1
    f'The course {course['name']} has {course['students']} students.'
                        ^
SyntaxError: invalid syntax
```



## 3.4 More on Functions in Python

We now extend our repertoire of dealing with functions in python.

In a sense, we now know all we have to about python function: we can define them and apply them to arguments. But python offers us much more: python

- treats functions as “first-class objects”, i.e. entities that can be given to other functions as arguments, and can be returned as results.
- provides more ways of passing arguments to a function than the rather rigid way we have seen above. This can be very convenient and make code more readable.

We will cover these features now. The main motivation for this is that they are widely used in programming and being able to read them is important for collaborating with experienced programmers and reading existing code.

We digress to the internals of functions that make them even more powerful. It turns out that we do not have to give a function a name at all.

### Anonymous Functions (lambda)

▷ **Observation 3.4.1** A python function definition combines making a function object with giving it a name.

▷ **Definition 3.4.2** python also allows to make **anonymous functions** via the lambda constructor for **function objects**:

```
lambda (p1, ..., pn): <<expr>>
```

▷ **Example 3.4.3** The following two python fragments are equivalent:

```
def cube (x):      cube = lambda (x): x*x*x
    x*x*x
```

The right one is just a **variable assignment** that assigns a **function object** to the **variable** cube. (In fact python uses the right one internally)

**Question:** Why use **anonymous functions**?

▷ **Answer:** We may not want to invent (i.e. waste) a name if the function is only used once (examples on the next slide)



Anonymous functions do not seem like a big deal at first, but having a way to construct a function that can be used in any expression, is very powerful as we will see now.

### Higher-Order Functions in python

▷ **Definition 3.4.4** We call a **function** a **higher-order function**, iff it takes a **function** as **argument**.

▷ **Definition 3.4.5** `map` and `filter` are built-in **higher-order functions** in python. They take a **function** and a **list** as arguments.

▷ `map( $f, L$ )` returns the list of  $f$ -values of the members of  $L$ .

▷ `filter( $p, L$ )` returns the sub-list  $L'$  of those  $l$  in  $L$ , such that  $p(l)=\text{True}$ .

▷ **Example 3.4.6** Mapping over and filtering a list

```
>>> li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
>>> map(lambda x: x*2, li)
[10, 14, 44, 194, 108, 124, 154, 46, 146, 122]
>>> filter(lambda x: (x%2 != 0), li)
[5, 7, 97, 77, 23, 73, 61]
```



Admittedly, in our example, we could also have defined a named function twice and then mapped that over `li`. But the code from Example 3.4.6 is more compact. Once we get used to the programming idiom and understand it, it becomes quite readable.

Another important feature of python **functions** is flexible argument passing. This allows to define **functions** that supply complex behaviors – for which we need to set many **parameters** – but simple calling patterns – which is good to hide complexity from the programmer.

The first **argument** passing feature we want to discuss is the use of **keyword arguments**, which gets around the problem of having to remember the position of an argument of a multi-argument function.

### Argument Passing in python: Keyword Arguments

▷ **Definition 3.4.7** The last  $k \leq n$  of  $n$  parameters of a **function** can be **keyword arguments** of the form  $p_i = \langle\langle \text{val} \rangle\rangle_i$ : If no argument  $a_i$  is given in the function call, the **default value**  $\langle\langle \text{val} \rangle\rangle_i$  is taken.

▷ **Example 3.4.8** The head of the `open` function is

```
def open(file, mode='r', buffering=-1, encoding=None, errors=None,
         newline=None, closefd=True, opener=None)
```

Even if we only call it with `open("foo")`, we can use **parameters** like `mode` or `opener` in the **body**; they have the corresponding **default value**.

We can also give more arguments via keywords, even out of order

```
open("foo", buffering=1, mode="+a")
```



**BTW:** The opener argument of `open` is a [function](#), and often an [anonymous function](#) is used if it is specified.

The next feature is dual to the last: instead of letting the caller leave out some arguments, we allow the caller more, which is then bound to a [list parameter](#).

### Argument Passing in python: Flexible Arity

▷ **Definition 3.4.9** python [functions](#) can take a variable number of [arguments](#):  
`def f(p1, ..., pk, *r)` allows  $n \geq k$  [arguments](#), e. g.  $f(a_1, \dots, a_k, a_{k+1}, \dots, a_n)$  and binds the [parameter](#) `r` to the [list](#)  $[a_{k+1}, \dots, a_n]$ .

▷ **Example 3.4.10** A somewhat construed function that reports the number of extra arguments

```
def flexary(a,b,*c)
    return len(c)
>>> flexary(1,2,3,4,5)
>>> 3
```

▷ **Definition 3.4.11** The [star operator](#) unpacks a [list](#) into an [argument](#) sequence.

▷ **Example 3.4.12 (Passing a starred list)**

```
def test_var_args_call(arg1, arg2, arg3):
    ...
args = ["two", 3]
test_var_args_call(1, *args)
```



Actually the [star operator](#) can be used in other situations as well, consider for instance

```
>>> numbers = [2, 1, 3, 4, 7]
>>> more_numbers = [*numbers, 11, 18]
>>> print(*more_numbers, sep=', ')
2, 1, 3, 4, 7, 11, 18
```

Here we have used the [star operator](#) twice: First to pass the list `numbers` as arguments to the [list constructor](#) and a second time to pass the extended list `more_numbers` to the `print` function.

Finally, we can combine the ideas from the last two to make [keyword arguments](#) flexary.

### Argument Passing in python: Flexible Keyword Arguments

▷ **Definition 3.4.13** python [functions](#) can take [keyword arguments](#):  
 if  $k$  is a sequence of key/value pairs then `def f(p1, ..., pn, **k)`, binds the keys to values in the body of  $f$ .

▷ **Example 3.4.14**

```
def kw_args(farg, **kwargs):
    print "formal arg:", farg
    for key in kwargs:
        print "another keyword arg: ",key," : ",kwargs[key]
```

```
>>> kw_args(1, myarg2="two", myarg3=3)
formal arg: 1
another keyword arg: myarg2 : two
another keyword arg: myarg3 : 3
```



## Argument Passing in python: Flexible Keyword Arguments (cont.)

▷ **Definition 3.4.15** The **double star operator** unpacks a **dictionary** into a sequence of **keyword arguments**.

▷ **Example 3.4.16 (Passing around dates as dictionaries)**

```
date_info = {'day': '01', 'month': '01', 'year': '2020'}
def filename (year='2019', month=1, day=1)
    year + "-" + month + "-" + day + ".txt"
>>> filename(**date_info)
'2020-01-01.txt'
```

▷ **Example 3.4.17 (Mixing formal and keyword arguments)**

```
def pdict(a1, a2, a3):
    print('a1: ', a1, ', a2: ', a2, ', a3: ', a3)
dict = {"a3": 3, "a2": "two"}
>>> pdict(1, **dict)
>>> a1: 1, a2: two, a3: 3
```



## 3.5 Regular Expressions: Patterns in Strings

Now we can come to the main topic of this Section: **regular expressions**. A domain-specific language for describing string patterns. **Regular expressions** are extremely useful, but also quite cryptical at first. They should be understood as a powerful tool, that relies on a language with a very limited vocabulary. It is more important to understand what this tool can do and how it works in principle than memorizing the vocabulary – that can be looked up on demand.

There are several dialects of regular expression languages that differ in details, but share the general setup and syntax. Here we introduce the **python** variant and recommend [PyRegex] for a cheat-sheet on **python** regular expressions (and an integrated **regex** tester).

### Regular Expressions, see [Pyt]

▷ **Definition 3.5.1** A **regular expression** (also called **regex**) is a formal expression that specifies a set of strings.

▷ **Definition 3.5.2 (Meta-Characters for Regexp)**

char	denotes
.	any single character (except a newline)
^	beginning of a string
\$	end of a string
[...]	any single character in the brackets
[^...]	any single character not in the brackets
(...)	marks a group
\n	the $n^{\text{th}}$ group
	disjunction
*	matches the preceding element zero or more times
+	matches the preceding element one or more times
?	matches the preceding element zero or one times
{n, m}	matches the preceding element between $n$ and $m$ times
\s	whitespace character
\S	non-whitespace character

All other characters match themselves, to match e.g. a `?`, escape with a `\`: `\?`.



Let us now fortify our intuition with some (simple) examples and a more complex one.

## Regular Expression Examples

### ▷ Example 3.5.3 (Regular Expressions and their Values)

regexp	values
car	car
.at	cat, hat, mat, ...
[hc]at	cat, hat
[^c]at	hat, mat, ... (but not cat)
^[hc]at	hat, cat, but only at the beginning of the line
[0–9]	Digits
[1–9][0–9]*	natural numbers
(.*)\1	mama, papa, wakawaka
cat dog	cat, dog

A regular expression can be interpreted by a regular expression processor (a program that identifies parts that match the provided specification) or a compiled by a parser generator.

⇒ **Example 3.5.4 (A more complex example)** The following **regexp** times in a variety of formats, such as 10:22am, 21:10, 08h55, and 7.15 pm.

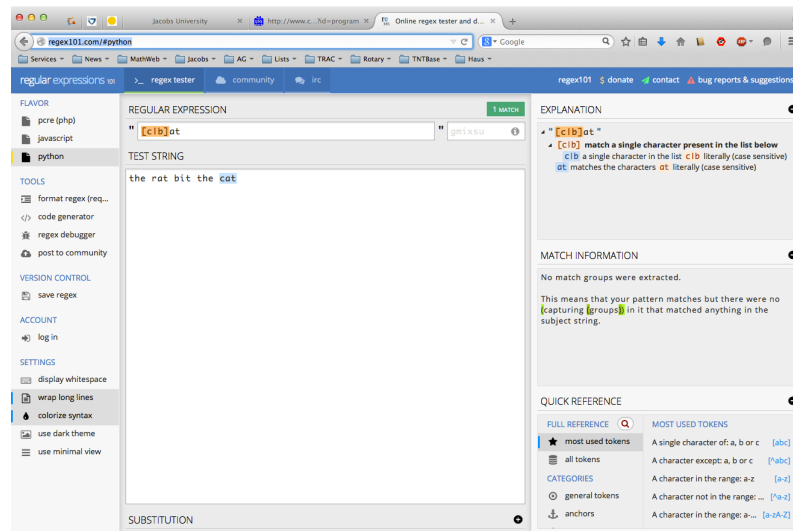
```
^(?:([0]?[d1[012]]|(?1[3–9]|2[0–3]))[.h]?[0–5]\d(?:\s?(?1(am|AM|pm|PM)))?$
```



As we have seen **regular expressions** can become quite cryptic and long (cf. e.g. Example 3.5.4), so we need help in developing them. One way is to use one of the many regexp testers online

## Playing with Regular Expressions

- ▷ If you want to play with [regexps](http://regex101.com), go e.g. to <http://regex101.com>



©: Michael Kohlhasse

90



## Regular Expressions in python

- ▷ We can use **regular expressions** directly in python by importing the re module (just add **import re** at the beginning)
- ▷ As python has UniCode strings, **regular expressions** support UniCode as well.
- ▷ Useful python functions that use **regular expressions**.

- ▷ `re.findall(⟨pat⟩,⟨str⟩)`: Return a list of non-overlapping matches of `⟨pat⟩` in `⟨str⟩`.

```
>>> re.findall(r"[h|c|r]at','the cat ate the rat on the mat')
['cat','rat']
```

- ▷ `re.sub(⟨pat⟩,⟨sub⟩,⟨str⟩)`: Replace substrings that match `⟨pat⟩` in `⟨str⟩` by `⟨sub⟩`.

```
>>> re.sub(r'\sAND|and\s', ' & ', 'Baked Beans and Spam')
'Baked Beans & Spam'
```

- ▷ `re.split(⟨pat⟩,⟨str⟩)`: Split `⟨str⟩` into substrings that match *metavarpat*.

```
>>> re.split(r'\s+', 'When shall we three meet again?'))
['When', 'shall', 'we', 'three', 'meet', 'again?']
>>> re.split(r'\s+|[?|.!,|:|;]', 'When shall we three me
['When', 'shall', 'we', 'three', 'meet', 'again']
```



©: Michael Kohlhasse

91



We will now see what we can do with **regular expressions** in a practical example. You should consider it as a “code reading/understanding” exercise, not think of it as something you should

(easily) be able to do yourself. But Example 3.5.5 could serve as a quarry of ideas for things you can do to texts with [regular expressions](#).

### Example: Correcting and Anonymizing Documents

▷ **Example 3.5.5 (Document Cleanup)** We write a [function](#) that makes simple corrections on documents and also crosses out all names to anonymize.

- ▷ *The worst president of the US, arguably was George W. Bush, right?*
- ▷ *However, are you famILlar with Paul Erdős or Henri Poincaré?* (Unicode)

Here is the function

- ▷ we import the regular expressions package and start the function

```
import re
def corranon (s)
```

- ▷ we first add blanks after commata

```
s = re.sub(r"(\S)", r" ", s)
```

- ▷ capitalize the first letter of a new sentence,

```
s = re.sub(r"([\.\?!])\w*(\S)",
           lambda (m):m.group(1)+m.group(2).upper()+m.group(3),
           s)
```



### Example: Correcting and Anonymizing Documents (cont.)

▷ **Example 3.5.6 (Document Cleanup (continued))**

- ▷ next we make abbreviations for regular expressions to save space

```
c = "[A-Z]"
l = "[a-z]"
```

- ▷ remove capital letters in the middle of words

```
s = re.sub(f"({l})({c}+)(\S)",
           lambda (m):f"{m.group(1)}{m.group(2).lower()}{m.group(3)}",
           s)
```

- ▷ and we cross-out for official public versions of government documents,

```
s = re.sub(f"({c}{l}+ ({c}{l}*(\.\.?) )?{c}{l}+)",
           lambda (m):re.sub("\S", "X", m.group(1)),
           s)
```

- ▷ finally, we return the result

```
s
```

*The worst president of the US, arguably was George W. Bush, right?*

becomes

*The worst president of the US, arguably was XXXXXX XX XXXX, right?*



## Example: Correcting and Anonymizing Documents (all)

### ▷ Example 3.5.7 (Document Cleanup (overview))

```
import re
def corrnanon (s)
    s = re.sub(r"(\S)", r" \1", s)
    s = re.sub(r"([\.\?!])\w*(\S)",
               lambda (m):m.group(1),r" ".upper()+m.group(2),
               s)
    c = "[A-Z]"
    l = "[a-z]"
    s = re.sub(f"({l})({c}+)(\S)",
               lambda (m):f"{m.group(1)}{m.group(2).lower()}{m.group(3)}",
               s)
    s = re.sub(f"({c}{l}+ ({c}{l}*(\S) )?{c}{l}+)",
               lambda (m):re.sub("\S", "X", m.group(1)),
               s)
    s
```





## Chapter 4

# Documents as Digital Objects

In this Chapter we take a first look at documents and how they are represented on the computer.

### 4.1 Representing & Manipulating Documents on a Computer

Now that we can represent characters as bit sequences, we can represent text documents. In principle text documents are just sequences of characters; they can be represented by just concatenating them.

#### Electronic Documents

- ▷ **Definition 4.1.1** An **electronic document** is any **media content** that is intended to be used via a **document renderer**, i.e. a **program** or **computing device** that transforms it into a form that can be directly perceived by the **end user**.
- ▷ **Definition 4.1.2** An **electronic document** that contains a digital encoding of textual material that can be read by the **end user** by simply presenting the encoded characters is called **digital text**.
- ▷ **Definition 4.1.3** **Digital text** is subdivided into **plain text**, where all characters carry the textual information and **formatted text**, which also contains instructions to the **document renderer**.
- ▷ **Example 4.1.4** python **programs** are **plain text**.



©: Michael Kohlhase

95

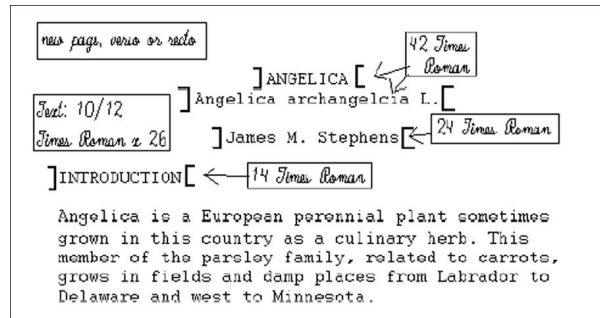


We will now establish a nomenclature for giving instructions to a **document renderer**. This has originated from movable (lead) type based typesetting but carries over well to **electronic documents**.

#### Document Markup

- ▷ **Definition 4.1.5** **Document markup** (or just **markup**) is the process of adding **control words** (special character sequences also called **markup codes**) to a **plain text** to control the structure, formatting, or the relationship among its parts.

▷ **Example 4.1.6** A text with **markup codes** (for printing)



▷ **Definition 4.1.7** The vocabulary and composition rules for a particular kind of **markup system** determine a **markup format**. The **markup format** used in a document is called its **document type**. All characters that are not **control words** constitute its **textual content**.



There are many systems for document markup, ranging from informal ones as in Example 4.1.6 that specify the intended document appearance to humans – in this case the printer – to technical ones which can be understood by machines but serving the same purpose.

**Markup** is by no means limited to **visual markup** for documents intended for printing as Example 4.1.6 may suggest. There are **aural markup** formats that instruct **document renderers** that transform documents to audio streams of e.g. reading speeds, intonation, and stress.

We now come to another aspect of **electronic documents**: We mostly interact with them in the form of **files**. Again, we fix our nomenclature.

## File Types

▷ **Observation 4.1.8** We mostly encounter **electronic documents** in the form of **file** on some **storage medium**.

▷ **Definition 4.1.9** A **text file** is a **file** that is structured as a **sequence** of **encoded characters**. Computer files that are not **text files** are called **binary files**.

▷ **Remark 4.1.10** **Text files** are usually encoded with ASCII, **ISO-Latin**, or – increasingly – **Unicode** encodings like UTF-8.

▷ **Example 4.1.11** python programs are stored in **text files**.

▷ In practice, **text files** are often processed as a **sequence** of **text lines** (or just **lines**), i.e. sub-strings separated by the **line feed character** U+000A; LINE FEED (LF). The **line number** is just the position in the sequence.



**Plain text** is different from **formatted text**, which includes **markup codes**, and **binary files** in which some portions must be interpreted as binary objects (encoded integers, real numbers, images, etc.)

As we have seen above, it does not take much to **render** a **text file**: we only need to guess the right **encoding scheme** so we can decode the file and show the character sequence to the user.

Indeed the UNIX cat just prints the contents of a [text file](#) to a [shell](#). But we need much more, we need tools with which we can compose and edit [text files](#); we do this with [text editors](#), which we will discuss now.

## Text Editors

- ▷ **Definition 4.1.12** A [text editor](#) is a program used for [rendering](#) and manipulating [text files](#).
- ▷ **Example 4.1.13** Popular [text editors](#) include
  - ▷ Notepad is a simple [editor](#) distributed with Windows.
  - ▷ emacs and vi are powerful [editors](#) originating from UNIX and optimized for programming.
  - ▷ sublime is a sophisticated programming [editor](#) for multiple [operating systems](#).
  - ▷ EtherPad is a browser-based real-time collaborative editor.
- ▷ **Example 4.1.14** Even though it can save documents as [text files](#), MS Word is not usually considered a [text editor](#), since it is optimized towards [formatted text](#); such “editors” are called [word processors](#).



What [text editors](#) do for [text files](#), [word processors](#) do for other [electronic documents](#).

## Word Processors and Formatted Text

- ▷ **Definition 4.1.15** A [word processor](#) is a software application, that – apart from being a [document renderer](#) – also supports the tasks of composition, editing, formatting, printing of [electronic documents](#).
- ▷ **Example 4.1.16** Popular [word processors](#) include
  - ▷ MS Word is an elaborated [word processor](#) for Windows, whose native format is [Office Open XML](#) (OOXML; file extension .docx).
  - ▷ OpenOffice and LibreOffice are similar [word processors](#) using the [ODF](#) format ([Open Office Format](#); file extension .odf) natively, but can also import other formats..
  - ▷ Pages is a [word processors](#) for Mac OS X it uses a proprietary format.
  - ▷ Office Online and GoogleDocs are browser-based real-time collaborative [word processors](#).
- ▷ **Example 4.1.17** [Text editor](#) are usually not considered to be [word processors](#), even though they can sometimes be used to edit [markup-based formatted text](#).



Before we go on, let us first get into some basics: how do we measure information, and how does this relate to units of information we know.

## 4.2 Measuring Sizes of Documents/Units of Information

Having represented documents are sequences of characters, we can use that to measure the sizes of documents. In this Section we will have a look at the underlying units of information and try to get an intuition about what we can store in files.

△: We will take a very generous stance towards what a document is, in particular, we will include pictures, audio files, spreadsheets, computer aided designs, . . . .

### Units for Information

- ▷ **Observation:** The smallest **unit** of information is knowing the state of a system with only two states.
- ▷ **Definition 4.2.1** A **bit** (a contraction of “binary digit”) is the basic **unit** of capacity of a data storage device or communication channel. The capacity of a system which can exist in only two states, is one **bit** (written as 1 b)
- ▷ **Note:** In the **ASCII encoding**, one character is encoded as 8 b, so we introduce another basic **unit**:
- ▷ **Definition 4.2.2** The **byte** is a derived **unit** for information capacity: 1 B = 8 b.



©: Michael Kohlhase

100



From the basic units of information, we can make prefixed units for prefixed units for larger chunks of information. But note that the usual **SI unit prefixes** are inconvenient for application to information measures, since powers of two are much more natural to realize.

### Larger Units of Information via Binary Prefixes

- ▷ We will see that memory comes naturally in powers to 2, as we address memory cells by binary numbers, therefore the derived information units are prefixed by special prefixes that are based on powers of 2.
- ▷ **Definition 4.2.3 (Binary Prefixes)** The following **binary unit prefix** es are used for information units because they are similar to the **SI unit prefixes**.

prefix	symbol	$2^n$	decimal	~SI prefix	Symbol
kibi	Ki	$2^{10}$	1024	kilo	k
mebi	Mi	$2^{20}$	1048576	mega	M
gibi	Gi	$2^{30}$	$1.074 \times 10^9$	giga	G
tebi	Ti	$2^{40}$	$1.1 \times 10^{12}$	tera	T
pebi	Pi	$2^{50}$	$1.125 \times 10^{15}$	peta	P
exbi	Ei	$2^{60}$	$1.153 \times 10^{18}$	exa	E
zebi	Zi	$2^{70}$	$1.181 \times 10^{21}$	zetta	Z
yobi	Yi	$2^{80}$	$1.209 \times 10^{24}$	yotta	Y

**Note:** The correspondence works better on the smaller prefixes; for **yobi** vs. **yotta** there is a 20% difference in magnitude.

▷ The **SI unit prefixes** (and their operators) are often used instead of the correct binary ones defined here.

▷ **Example 4.2.4** You can buy hard-disks that say that their capacity is “one tera-byte”, but they actually have a capacity of one tebibyte.



©: Michael Kohlhase

101



Let us now look at some information quantities and their real-world counterparts to get an intuition for the information content.

### How much Information?

<b>Bit (b)</b>	<i>binary digit 0/1</i>
<b>Byte (B)</b>	<i>8 bit</i>
2 Bytes	A Unicode character in UTF.
10 Bytes	your name.
<b>Kilobyte (kB)</b>	<i>1,000 bytes OR <math>10^3</math> bytes</i>
2 Kilobytes	A Typewritten page.
100 Kilobytes	A low-resolution photograph.
<b>Megabyte (MB)</b>	<i>1,000,000 bytes OR <math>10^6</math> bytes</i>
1 Megabyte	A small novel or a 3.5 inch floppy disk.
2 Megabytes	A high-resolution photograph.
5 Megabytes	The complete works of Shakespeare.
10 Megabytes	A minute of high-fidelity sound.
100 Megabytes	1 meter of shelved books.
500 Megabytes	A CD-ROM.
<b>Gigabyte (GB)</b>	<i>1,000,000,000 bytes or <math>10^9</math> bytes</i>
1 Gigabyte	a pickup truck filled with books.
20 Gigabytes	A good collection of the works of Beethoven.
100 Gigabytes	A library floor of academic journals.





©: Michael Kohlhase

102



### How much Information?

<b>Terabyte (T B)</b>	<i>1,000,000,000,000 bytes or <math>10^{12}</math> bytes</i>
1 Terabyte	50000 trees made into paper and printed.
2 Terabytes	An academic research library.
10 Terabytes	The print collections of the U.S. Library of Congress.
400 Terabytes	National Climate Data Center (NOAA) database.
<b>Petabyte (P B)</b>	<i>1,000,000,000,000,000 bytes or <math>10^{15}</math> bytes</i>
1 Petabyte	3 years of EOS data (2001).
2 Petabytes	All U.S. academic research libraries.
20 Petabytes	Production of hard-disk drives in 1995.
200 Petabytes	All printed material (ever).
<b>Exabyte (E B)</b>	<i>1,000,000,000,000,000,000 bytes or <math>10^{18}</math> bytes</i>
2 Exabytes	Total volume of information generated in 1999.
5 Exabytes	All words ever spoken by human beings ever.
300 Exabytes	All data stored digitally in 2007.
<b>Zettabyte (Z B)</b>	<i>1,000,000,000,000,000,000,000 bytes or <math>10^{21}</math> bytes</i>
2 Zettabytes	Total volume digital data transmitted in 2011
100 Zettabytes	Data equivalent to the human Genome in one body.


©: Michael Kohlhasse
103


The information in this table is compiled from various studies, most recently [HL11].

**Note:** Information content of real-world artifacts can be assessed differently, depending on the view. Consider for instance a text typewritten on a single page. According to our definition, this has ca. 2kB, but if we fax it, the image of the page has 2MB or more, and a recording of a text read out loud is ca. 50MB. Whether this is a terrible waste of bandwidth depends on the application. On a fax, we can use the shape of the signature for identification (here we actually care more about the shape of the ink mark than the letters it encodes) or can see the shape of a coffee stain. In the audio recording we can hear the inflections and sentence melodies to gain an impression on the emotions that come with text.

### 4.3 Hypertext Markup Language

**WWW** documents have a specialized **markup format** that mixes markup for document structure with layout markup, hyper-references, and interaction. The HTML markup elements always concern text fragments, they can be nested but may not otherwise overlap. This essentially turns a text into a document tree.

In IWGS, we discuss HTML mostly as a way to build interfaces of web applications. Therefore we will prioritize those aspects of HTML that have to do with “programming documents” over the creation of nice-looking web pages. Therefore we will pick up the notion of nested text fragments marked up by well-bracketed tags and elements in Section 4.4 and generalize these ideas to XML as a general representation paradigm for semi-structured data in Section 4.5.

We will also postpone the discussion of cascading style sheets, which have evolved as the dominant technology for the specification of presentation (layout, colors, and fonts) for marked-up documents, to Chapter 6.

HTML was created in 1990 and standardized in version 4 in 1997 [RHJ98]. Since then the **WWW** has evolved considerably from a web of static **web pages** to a Web in which highly dynamic **web pages** become user interfaces for web-based applications and even mobile applets. HTML5 standardized the necessary infrastructure in 2014 [Hic+14].

## HTML: Hypertext Markup Language

▷ **Definition 4.3.1** The **HyperText Markup Language** (HTML), is a representation format for **web pages** [Hic+14].

▷ **Definition 4.3.2 (Main markup elements of HTML)** HTML marks up the structure and appearance of text with **tags** of the form `<el>` (**begin tag**), `</el>` (**end tag**), and `<el/>` (**empty tag**), where `el` is one of the following

structure	html, head, body	metadata	title, link, meta
headings	h1, h2, ..., h6	paragraphs	p, br
lists	ul, ol, dl, ..., li	hyperlinks	a
multimedia	img, video, audio	tables	table, th, tr, td, ...
styling	style, div, span	old style	b, u, tt, i, ...
interaction	script	forms	form, input, button
Math	MathML (formulae)	interactive graphics	vector graphics (SVG) and canvas (2D bitmapped)

▷ **Example 4.3.3** A (very simple) HTML file with a single paragraph.

```
<html>
<body>
  <p>Hello IWGS students!</p>
</body>
</html>
```



The thing to understand here is that HTML uses the characters `<`, `>`, and `/` to delimit the markup. All markup is in the form of **tags**, so anything that is not between `<` and `>` is the **textual content**.

We will not give a complete introduction to the various tags and elements of the HTML language here, but refer the reader to the HTML recommendation [Hic+14] and the plethora of excellent web tutorials. Instead we will introduce the concepts of HTML markup by way of examples.

The best way to understand HTML is via an example. Here we have prepared a simple file that shows off some of the basic functionality of HTML.

## A very first HTML Example (Source)

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>A first HTML Web Page</title>
</head>
<body>
  <h1>Anatomy of a HTML Web Page</h1>
  <h3>Michael Kohlhase<br/>FAU Erlangen Nuernberg</h3>
  <h2 id="intro">1. Introduction</h2>
  <p>This is really easy, just start writing.</p>
  <h2>3. Main Part: show off features</h2>
  <p>We can can markup <b>text</b> <em>styles</em> inline.</p>
  <p> And we can make itemizations:
  <ul>
    <li> with a list item</li>
```

```

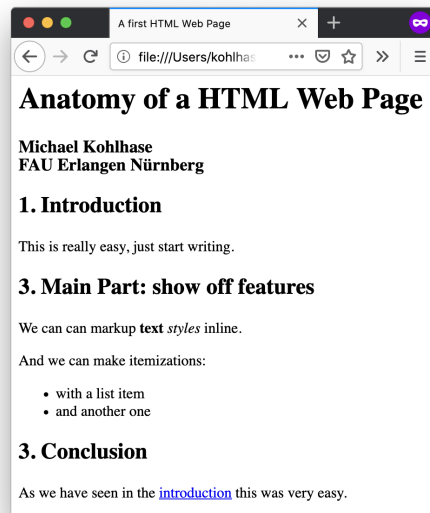
</li> and another one</li>
</ul>
</p>
<h2>4. Conclusion</h2>
<p> As we have seen in the <a href="#intro">introduction</a> this
was very easy.</p>
</body>
</html>

```



The thing to understand here is that HTML markup is itself a well-balanced structure of [begin](#) and [end tags](#). That wrap other balanced HTML structures and – eventually – [textual content](#). The HTML recommendation [RHJ98] specifies the visual appearance expectation and interactions afforded by the respective [tags](#), which HTML-aware software systems – e.g. a [web browser](#) – then execute. In the next slide we see how **Firefox** displays the HTML document from the previous.

## A very first HTML Example (Result)



In the last slide, we have seen **Firefox** as a [document renderer](#) for HTML. We will now introduce this class of [programs](#) in general and point out a few others.

## Web Browsers

- ▷ **Definition 4.3.4** A [web browser](#) is a software application for retrieving (via [HTTP](#)), presenting, and traversing information resources on the [WWW](#), enabling users to view [web pages](#) and to jump from one page to another.
- ▷ [Practical Browser Tools](#):

- ▷ Status Bar: security info, page load progress
- ▷ Favorites (bookmarks)
- ▷ View Source: view the code of a [web page](#)
- ▷ Tools/Internet Options, history, temporary Internet files, home page, auto complete, security settings, programs, etc.
- ▷ **Example 4.3.5 (Common Browsers)**
  - ▷ Edge is provided by Microsoft for Windows (replaces MS Internet Explorer)
  - ▷ Firefox is an open source browser for all platforms, it is known for its standards compliance.
  - ▷ Safari is provided by Apple for Mac OS X and Windows
  - ▷ Chrome is a lean and mean browser provided by Google (very common)
  - ▷ WebKit is a library that forms the open source basis for Safari and Chrome.



Let us now look at a couple of more advanced tools available in most [web browsers](#) for dealing with the underlying HTML document.

## Browser Tools for dealing with HTML, e.g. in Firefox

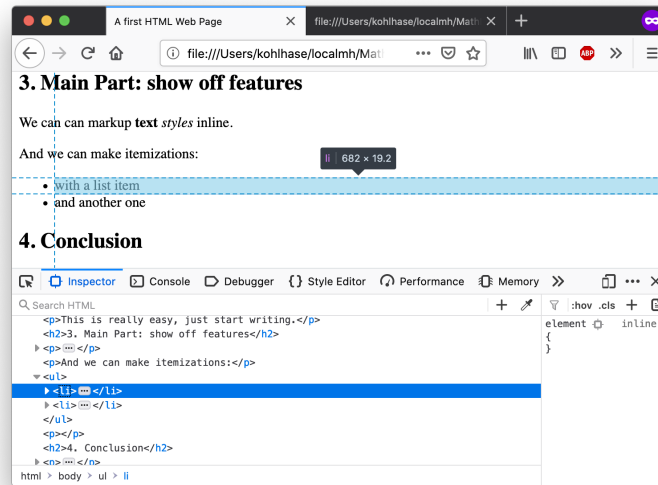
- ▷ Hit Control-U to see the page source in the browser

```

1 <html xmlns="http://www.w3.org/1999/xhtml">
2 <head>
3   <title>A first HTML Web Page</title>
4 </head>
5 <body>
6   <h1>Anatomy of a HTML Web Page</h1>
7   <h3>Michael Kohlhase<br/>FAU Erlangen Nürnberg</h3>
8   <h2 id="intro">1. Introduction</h2>
9   <p>This is really easy, just start writing.</p>
10  <h2>3. Main Part: show off features</h2>
11  <p>We can can markup <b>text</b> <em>styles</em> inline.</p>
12  <p>And we can make itemizations:
13    <ul>
14      <li> with a list item</li>
15      <li> and another one</li>
16    </ul>
17  </p>
18  <h2>3. Conclusion</h2>
19  <p>As we have seen in the <a href="#intro">introduction</a> this
20  was very easy.</p>
21 </body>
22 </html>
23

```

- ▷ go to an element and right-click ~ "Inspect element"

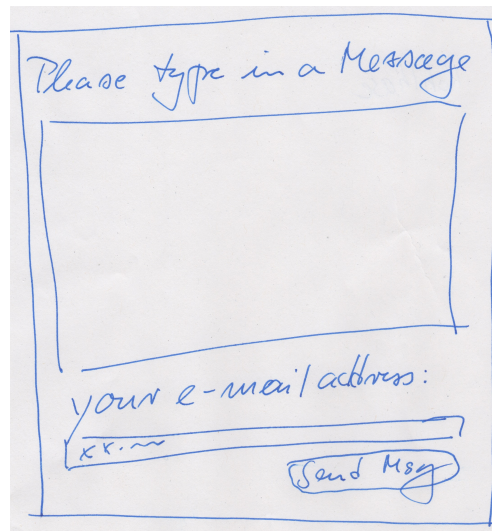


We have used **FireFox** as an example here, but these tools are available in some form in all major browsers – the browser vendors want to make their offerings attractive to web developers, so that web pages and web applications get tested and debugged in them and therefore work as expected.

After this simple example, we will come to a more complex one: a little “contact form” as we find on many web sites that can be used for sending a message to the owner of the site. Let us only look at the design of the form document before we go into the interaction facilities afforded it.

## HTML in Practice: Worked Example

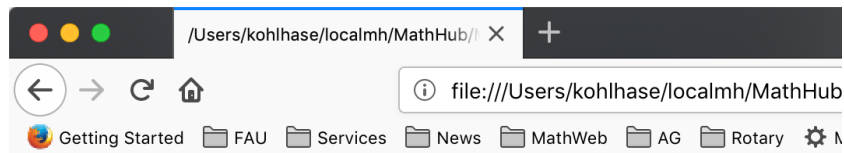
- ▷ Make a design and “paper prototype” of the page



- ▷ put the intended text into a file: `contact.html`

Contact  
 Please enter a message:  
 Your e-mail address: xx @ xx.de  
 Send message

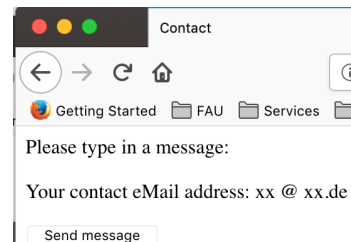
▷ load into your browser to check the state



Contact Please type in a message: Your e-mail address: xx @ xx.de Send message

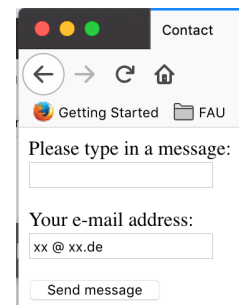
▷ add title, paragraph and button markup:

```
<title>Contact</title>
<h2>Please enter a message:</h2>
<h3>Your e-mail address: xx @ xx.de</h3>
<button>Send message</button>
```



▷ add input fields and breaks:

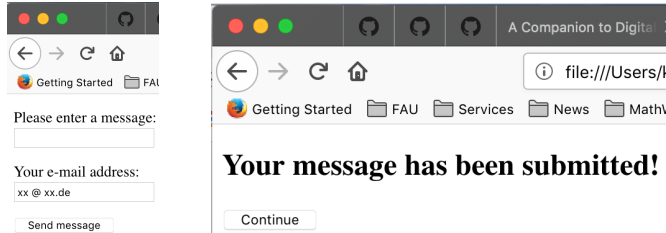
```
<title>Contact</title>
<h2>Please enter a message:</h2>
<input name="msg" type="text"/>
<h3> Your e-mail address:</h3>
<input name="addr" type="text"
      value="xx @ xx.de"/>
<br/>
<button>Send message</button>
```



▷ convert into a HTML form with action (message receipt):

```
<title>Contact</title>
<form action="contact-after.html">
  <h2>Please enter a message:</h2>
  <input name="msg" type="text"/>
  <h3>Your e-mail address:</h3>
  <input name="addr" type="text"
        value="xx @ xx.de"/>
  <br/>
  <input type="submit"
        value="Send message"/>
</form>
```

```
<title>
  Contact — Message Confirmed
</title>
<form action="contact4.html">
  <h2>
    Your message has been submitted!
  </h2>
  <input type="submit"
        value="Continue"/>
</form>
```



▷ That's as far as we will go, the rest is page layout and interaction. (up next)

©: Michael Kohlhase 109

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

After designing the functional (what are the text blocks) structure of the contact form, we will need to understand the interaction with the contact form.

## HTML Forms

- ▷ **Question:** But how does the interaction with the contact form really work?
- ▷ **Definition 4.3.6** The HTML form element groups the layout and **input elements**:
  - ▷ `<form action="⟨URI⟩"...>` specifies the **form action** (as a web page address)
  - ▷ `<input type="submit"../>` triggers the **form action**: it sends the **form data** to web page specified there.
- ▷ **Example 4.3.7 (In the Contact Form)** We send the request  
`contact-after.html?msg=Hi&addr=foo@bar.de`  
 We current ignore the **form data** (the part after the ?)
- ▷ We will come to the full story of processing actions later.



©: Michael Kohlhase

110



Unfortunately, we can only see what the browser sends to the server at the current state of play, not what the server does with the information. But we will get to this when we take up the example again.

For the moment, we made use of the fact that we can just specify the page `contact-after.html`, which the browser displays next. That ignores the query part and – via a **form** element of its own gets the user back to the original contact form.

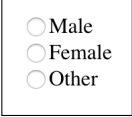
## More useful types of Input fields

- ▷ radio buttons: `type="radio"` (grouped by name attribute)

```

<input type="radio" name="gender" value="male"/>Male<br/>
<input type="radio" name="gender" value="female"/>Female<br/>
<input type="radio" name="gender" value="other"/>Other

```



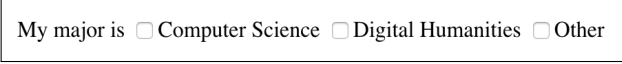
▷ check boxes: type="checkbox"

My major is

```

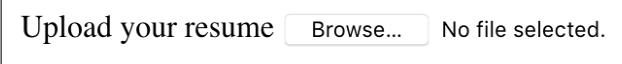
<input type="checkbox" name="major" value="cs"/>Computer Science
<input type="checkbox" name="major" value="dh"/>Digital Humanities
<input type="checkbox" name="major" value="other"/>Other

```



▷ file selector dialogs (interaction is system-specific – here for MacOS Mojave)

<p> Upload your resume <input type="file" name="resume"/></p>




▷ drop down menus: select and option


Which animal do you like?<br/>

```

<select name="animals">
  <option value="bird">Bird</option>
  <option value="hamster">Hamster</option>
  <option value="cat">Cat</option>
  <option value="dog">Dog</option>
</select>


```





©: Michael Kohlhase

111



## 4.4 Documents as Trees

We have concentrated on HTML as a document [markup format](#) for interactive multimedia documents. Before we progress, we want to discuss an important feature: all practical [markup formats](#) that [control words](#) are in some sense well-bracketed. Well-bracketed structures are well-understood in CS and Mathematics: they are called [trees](#) and come with a rich and useful collection of descriptive concepts and tools. We will present the concepts in this Section and the tools they enable in Section 4.5.

### Well-Bracketed Structures in Computer Science

▷ **Observation 4.4.1** *We often deal with well-bracketed structures in CS, e.g.*

▷ *Expressions: e.g.  $\frac{3 \cdot (a + 5)}{2x + 7}$  (numerator and denominator in fractions implicitly bracketed)*

▷ *Markup Languages like HTML:*

```

<html>
  <head><script>.emph {color:red}</script></head>
  <body><p>Hello IWGS</p></body>
</html>

```

- ▷ *Programming languages like python:*

```
answer = input("Are you happy? ")
if answer == 'No' or answer == 'no':
    print("Have a chocolate!")
else:
    print("Good!")
print("Can I help you with something else?")
```

**Idea:** Come up with a common data structure that allows to program the same algorithms for all of them. (common approach to scaling in Computer Science)



## ▷ A Common Data Structure for Well-Bracketed Structures

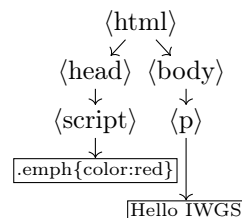
- ▷ **Observation 4.4.2** *In well-bracketed structures, brackets contain two kinds of objects*

- ▷ *bracket-less objects*
- ▷ *well-bracketed structures themselves*

**Idea:** Write bracket pairs and bracket-less objects as nodes, connect when contained

- ▷ **Example 4.4.3** Let's try this for HTML – creating nodes top to bottom

```
<html>
<head>
  <script>.emph {color:red}</script>
</head>
<body>
  <p>Hello IWGS</p>
</body>
</html>
```



- ▷ We call such structures **trees** (more on trees next)



**Trees** are well-understood mathematical objects and **tree data structures** are very commonly used in Computer Science and **programming**. As such they have a well-developed nomenclature, which we will introduce now.

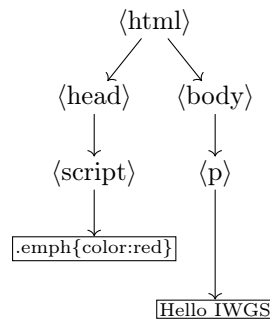
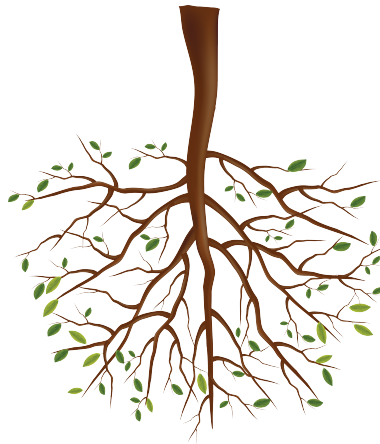
## Well-Bracketed Structures: Tree Nomenclature

- ▷ In Math and CS, such well-bracketed structures are called **trees** (with **root**, **branches**,

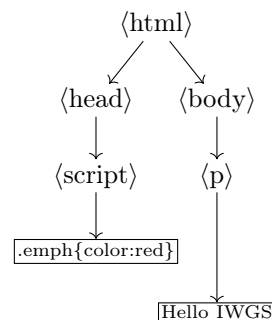
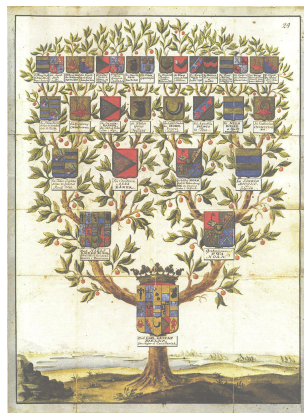
leaves, And height).

(but written upside-down)

▷ **Example 4.4.4** In a tree, there is only one path from the root to the leaves



▷ **Definition 4.4.5** We speak of parent, child, ancestor, and descendant nodes (genealogy nomenclature)



**Why are trees written upside-down?:** The main answer is that we want to draw tree diagrams in text. And we naturally start drawing a tree at the root. So, if a tree grows from the root and we do not exactly know the tree height, then we do not know how much space to leave. When we

write trees upside down, we can directly start from the **root** and grow the **tree** downward as long as we need. We will keep to this tradition in the IWGS course.

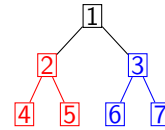
We will now make use of the **tree** structure for computation. Even if the computing tasks we pursue here may seem a bit abstract, they show very nicely how tree algorithms typically work.

### Computing with Trees in python

▷ **Observation 4.4.6** All connected substructures of **trees** are **trees** themselves.

▷ **Idea:** operate on the tree by “Divide and Conquer”

- ▷ operate on the two subtrees
- ▷ combine results, taking root into account



This approach lends itself very well to **recursive programming** (functions that call themselves)

▷ **Idea:** represent **trees** as **lists** of tree labels and **lists** (of **subtrees**).

▷ **Example 4.4.7** (The tree above) represented as `[1,[2,[[4],[5]]],[2,[[6],[7]]]]`  
compute the **tree height** by the following python functions:

```
def height (tree):
    return maxh(tree[1:]) + 1
height([1,[2,[[4],[5]]],[2,[[6],[7]]]])
>>> 3
```

```
def maxh (l):
    if l == []:
        return 0
    else
        return max(height(l[0]),maxh(l[1:]))
```



Let us have a closer look at Example 4.4.7. The algorithm consists of two **functions**:

1. **height**, which computes the **height** of an input **tree** by delegating the computation of the maximal **height** of its **children** to **maxh** and then incrementing the value by 1.
2. **maxh**, which takes a list of **trees** and computes the maximum of their **heights** by calling **height** on the first input **tree** and then comparing with the maximal **height** of the remaining **trees**.

Note that **maxh** and **height** each **call** the other. We call such **functions mutually recursive**. Here this behavior poses no problem, since the arguments in the recursive calls are smaller than the inputs: for **maxh** it is the rest list, and for **height** the “list of children” of the input tree.

Example 4.4.7 was complex for two reasons: **mutual recursion** and the somewhat cryptic encoding of trees as lists of lists of integers. We claim that recursive programming is “not a bug, but a feature”, as it allows to succinctly capture the “divide-and-conquer” approach afforded by trees. For the cryptic encoding of trees we can do better.

### Computing with Trees in python (Dictionaries)

▷ **That was a bit cryptic:** i.e. very difficult to read/debug

▷ **Idea:** why not use **dictionaries**? (they are more explicit) compute the tree weight (sum of all labels) by

```

t =
{
  "label": 1,
  "children": [
    {
      "label": 2,
      "children": [
        {
          "label": 4,
          "children": []
        },
        {
          "label": 5,
          "children": []
        }
      ]
    },
    {
      "label": 3,
      "children": [
        {
          "label": 6,
          "children": []
        },
        {
          "label": 7,
          "children": []
        }
      ]
    }
  ]
}

def wsum (tl):
    if tl == []:
        return 0;
    else:
        return weight(tl[0]) + wsum(tl[1:])

def weight (tree):
    return tree["label"] + wsum(tree["children"]);

weight(t);
>>> 28

```



Again, we have two **mutually recursive functions**: `weight` that takes a tree, and `wsum` that takes a list and the recursion goes analogously. Only that this time, the list of children is a dictionary value and the calls are clearer. The only real difference, is that in `wsum` we have to add up the weight of the head of the list and the joint sum of the rest list.

## The Document Object Model

- ▷ **Definition 4.4.8** The **document object model (DOM)** is a **data structure** for storing **marked-up electronic documents** as **trees** together with a standardized set of access methods for manipulating them.
- ▷ **Idea**: When a **web browser** loads a HTML page, it directly parses it into a **DOM** and then works exclusively on that. In particular, the HTML document is immediately discarded; documents are rendered from the **DOM**.



## 4.5 An Overview over XML Technologies

We have seen that many of the technologies that deal with marked-up documents utilize the tree-like structure of (the **DOM**) of HTML documents. Indeed, it is possible to abstract from the concrete vocabulary of HTML that the intended layout of hypertexts and the function of its fragments, and build a generic framework for document trees. This is what we will study in this Section.

## XML (EXtensible Markup Language)

- ▷ XML is framework for **markup formats** for documents and structured **data**.
  - ▷ tree representation language (begin/end brackets)
  - ▷ restrict instances by *Doc. Type Def. (DTD)* or *Schema* (Grammar)
  - ▷ Presentation markup by *style files* (XSL: XML Style Language)

### Intuition: XML is extensible HTML

- ▷ logic annotation (*markup*) instead of presentation!
- ▷ many tools available: parsers, compression, data bases, ...
- ▷ conceptually: transfer of *trees* instead of *strings*.
- ▷ details at <http://w3c.org> (XML is standardize by the WWWWeb Consortium)



©: Michael Kohlhase

118



The idea of XML being an “extensible” markup language may be a bit of a misnomer. It is made “extensible” by giving language designers ways of specifying their own vocabularies. As such XML does not have a vocabulary of its own, so we could have also it an “empty” markup language that can be filled with a vocabulary.

### XML is Everywhere (E.g. Web Pages)

- ▷ **Example 4.5.1** Open web page file in FireFox, then click on *View* ↘ *PageSource*, you get the following text: (showing only a small part and reformatting)

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Michael Kohlhase</title>
    <meta name="generator"
          content="Page generated from XML sources with the WSML package"/>
  </head>
  <body>...
  <p>
    <i>Professor of Computer Science</i><br/>
    Jacobs University<br/><br/>
    <strong>Mailing address - Jacobs (except Thursdays)</strong><br/>
    <a href="http://www.jacobs-university.de/schools/ses">
      School of Engineering & Science
    </a><br/>...
  </p>...
</body>
</html>
```

- ▷ **Definition 4.5.2** XHTML is the XML version of HTML (just make it valid XML)



©: Michael Kohlhase

119



Now we see an example of an XML file that is used for communicating data in a machine-readable, but human-understandable way.

### XML is Everywhere (E.g. Catalogs)

- ▷ **Example 4.5.3 (The NYC Galleries catalog)** A public XML file at <https://data.cityofnewyork.us/download/kcrm-j9hh/application/xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<museums>
  <museum>
    <name>American Folk Art Museum</name>
    <phone>212-265-1040</phone>
    <address>45 W. 53rd St. (at Fifth Ave.)</address>
```

```

<closing>Closed: Monday</closing>
<rates>admission: $9; seniors/students, $7; under 12, free</rates>
<specials>
  Pay-what-you-wish: Friday after 5:30pm;
  refreshments and music available
</specials>
</museum>
<museum>
  <name>American Museum of Natural History</name>
  <phone>212-769-5200</phone>
  <address>Central Park West (at W. 79th St.)</address>
</closing>Closed: Thanksgiving Day and Christmas Day</closing>

```



This XML uses an ad-hoc markup language: Every `<museum>` **element** represents one museum in New York City (NYC). Its **children** convey the detailed information as “key value pairs”.

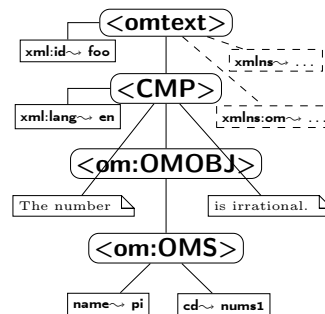
## XML Documents as Trees

▷ **Idea:** An XML Document is a Tree

```

<omtext xml:id="foo"
  xmlns="..."
  xmlns:om="...">
  <CMP xml:lang='en'>
    The number
    <om:OMOBJ>
      <om:OMS cd="nums1"
        name="pi"/>
    </om:OMOBJ>
    is irrational.
  </CMP>
</omtext>

```



▷ **Definition 4.5.4** The **XML document tree** is made up of **element nodes**, **attribute nodes**, **text nodes** (and **namespace declarations**, **comments**,...)



## XML Documents as Trees (continued)

▷ **Definition 4.5.5** For communication this tree is serialized into a balanced bracketing structure, where

- ▷ an inner **element node** is represented by the brackets `<el>` (called the **opening tag**) and `</el>` (called the **closing tag**).
- ▷ The **leaves** of the XML **tree** are represented by **empty element tags** (serialized as `<el></el>`, which can be abbreviated as `<el/>`)
- ▷ and text nodes (serialized as a sequence of Unicode characters).

- ▷ An **element node** can be annotated by further information using **attribute nodes** — serialized as an **attribute** in its **opening tag**.

**Note:** As a document is a **tree**, the XML specification mandates that there must be a unique **document root**.



We have claimed above that the **tree** nature of XML documents is one of the main advantages. Let us now see how **python** makes good on this promise.

**Acknowledgements:** Many of the examples and the flow of exposition in the next slides has been adapted from the **lxml** tutorial [LXMLc].

### ▷ Computing with XML in python (Elements)

- ▷ The **lxml** library [LXMLa] provides python bindings for the (low-level) LibXML2 library. (install it with `pip3 install lxml`)

- ▷ The **ElementTree** API is the main way to programmatically interact with XML. Activate it by importing **etree** from **lxml**:

```
>>> from lxml import etree
```

- ▷ Elements are easily created, their properties are accessed with special accessor methods

```
>>> root = etree.Element("root")
>>> print(root.tag)
root
```

- ▷ Elements are organised in an XML **tree** structure. To create **child element nodes** and add them to a **parent element node**, you can use the **append()** method:

```
>>> root.append( etree.Element("child1") )
```

- ▷ **Abbreviation:** create a **child element node** and add it to a **parent**.

```
>>> child2 = etree.SubElement(root, "child2")
>>> child3 = etree.SubElement(root, "child3")
```



### Computing with XML in python (Result)

- ▷ Here is the resulting XML tree so far; we serialize it via **etree.tostring**

```
>>> print(etree.tostring(root, pretty_print=True))
<root>
  <child1/>
  <child2/>
  <child3/>
</root>
```

- ▷ BTW, the **etree.tostring** is highly configurable via default arguments.

```
tostring(element_or_tree,
          encoding=None, method="xml", xml_declaration=None, doctype=None,
          pretty_print=False, with_tail=True, standalone=None, exclusive=False,
```

```
inclusive_ns_prefixes=None, with_comments=True, strip_text=False)
```

The lxml API documentation [LXMLb] has the details.



©: Michael Kohlhase

124



This method of “manually” producing XML [trees](#) in memory by applying `etree` methods may seem very clumsy and tedious. But the power of `lxml` lies in the fact that these can be embedded in python programs. And as always, programming gives us the power to do things very efficiently.

### Computing with XML in python (Automation)

- ▷ This may seem trivial and/or tedious, but we have python power now:

```
def nchildren (n):
    root = etree.Element("root")
    for i in range(1,n):
        root.append(f"child{i}")
```

produces a tree with 1000 children without much effort.

```
>>> t = nchildren(1000)
>>> print(len(t))
>>> 1000
```

We abstain from printing the XML tree (too large) and only check the length.



©: Michael Kohlhase

125



But XML documents that only have documents, are boring; let's do attributes next. Recall that attributes are essentially string-valued key/value pairs. So what could be more natural than treating them like [dictionaries](#).

### Computing with XML in python (Attributes)

- ▷ Attributes can directly be added in the Element function

```
>>> root = etree.Element("root", interesting="totally")
>>> etree.tostring(root)
b'<root interesting="totally"/>'
```

- ▷ The `.get` method returns attributes in a dictionary-like object.

```
>>> print(root.get("interesting"))
totally
```

we can set them with the `.set` method

```
>>> root.set("hello", "Huhu")
>>> print(root.get("hello"))
Huhu
```

this results in a changed element:

```
>>> etree.tostring(root)
b'<root interesting="totally" hello="Huhu"/>'
```



©: Michael Kohlhase

126



Recall that we could use python [dictionaries](#) for iterating over in a `for` loop. We can do the same for attributes:

### Computing with XML in python (Attributes; continued)

▷ We can access attributes by the keys, values, and items methods, known from [dictionaries](#):

```
>>> sorted(root.keys())
['hello', 'interesting']

>>> for name, value in sorted(root.items()):
...     print(f'{name} = {value}')
hello = 'Huhu'
interesting = 'totally'
```

⚠: To get a ‘real’ dictionary, use the attrib method (e.g. to pass around)

```
>>> attributes = root.attrib
```

Note that attributes participates in any changes to root and vice versa.

▷ ⚠: To get an independent snapshot of the attributes that does not depend on the XML tree, copy it into a [dict](#):

```
>>> d = dict(root.attrib)
>>> sorted(d.items())
[('hello', 'Guten Tag'), ('interesting', 'totally')]
```



The last two items touch a somewhat delicate subject in programming. [Mutable](#) and [immutable data structures](#): the former can be changed in-place – as we have above with the `.set` method, and the latter cannot. Both have their justification and respective advantages. [Immutable data structures](#) are “safe” in the sense that they cannot be changed unexpectedly by another part of the [program](#), they have the disadvantage that every time we want to have a variant, we have to copy the whole object. [Mutable](#) ones do not – we can change in place – but we have to be very careful about who accesses them when.

This is also the reason why we spoke of “dictionary-like interface” to XML trees in lxml: [dictionaries](#) are [immutable](#), while XML trees are not.

The main remaining functionality in XML is the treatment of text. XML treats text as special kinds of [node](#) in the [tree](#): [text nodes](#). They can be treated just like any other [node](#) in the XML [tree](#) in the [etree](#) library.

### Computing with XML in python (Text nodes)

▷ Elements can contain text: we use the `.text` property to access and set it.

```
>>> root = etree.Element("root")
>>> root.text = "TEXT"
>>> print(root.text)
TEXT
>>> etree.tostring(root)
b'<root>TEXT</root>'
```



To get a real intuition about what is happening, let us see how we can use all the functionality so far: we programmatically construct an HTML [tree](#).

### Case Study: Creating an HTML document

- ▷ We create nested html and body elements
 

```
>>> html = etree.Element("html")
>>> body = etree.SubElement(html, "body")
```
- ▷ Then we inject a text node into the latter using the `.text` property.
 

```
>>> body.text = "TEXT"
```
- ▷ Let's check the result
 

```
>>> etree.tostring(html)
b'<html><body>TEXT</body></html>'
```
- ▷ We add another element: a line break and check the result
 

```
>>> br = etree.SubElement(body, "br")
>>> etree.tostring(html)
b'<html><body>TEXT<br/></body></html>'
```
- ▷ Finally, we can add trailing text via the `.tail` property
 

```
>>> br.tail = "TAIL"
>>> etree.tostring(html)
b'<html><body>TEXT<br/>TAIL</body></html>'
```



Note the use of the `.tail` property here? While the `.text` property can be used to set “all” the text in an XML element, we have to use the `.tail` property to add trailing text (e.g. after the `<br/>` element).

Notwithstanding the “python power” argument from above, there are situations, where we just want to write down XML fragments and insert them into (programmatically created) XML [trees](#). lxml as functionality for this: [XML literals](#), which we introduce now.

### Computing with XML in python (XML Literals)

- ▷ **Definition 4.5.6** We call any [string](#) that is well-formed XML an [XML literal](#).
- ▷ We can use the XML [function](#) to read [XML literals](#).
 

```
>>> root = etree.XML("<root>data</root>")
```

The result is a first-class element tree, which we can use as above

```
>>> print(root.tag)
root
>>> etree.tostring(root)
b'<root>data</root>'
```

BTW, the `fromstring` [function](#) does the same
- ▷ There is a variant `html` that also supplies the necessary HTML decoration.
 

```
>>> root = etree.HTML("<p>data<br/>more</p>")
>>> etree.tostring(root)
b'<html><body><p>data<br/>more</p></body></html>'
```

**BTW:** If you want to read only the text content of an XML element, i.e. without any intermediate tags, use the method keyword in `tostring`:

```
>>> etree.tostring(root, method="text")
b'datamore'
```

▷



©: Michael Kohlhasse

130



## XML is Everywhere (E.g. document metadata)

- ▷ **Example 4.5.7** Open a PDF file in Acrobat Reader, then click on *File \ Document Properties \ Document Metadata* you get the following text: (showing only a small part)

```
<rdf:RDF xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:iX='http://ns.adobe.com/iX/1.0/'>
  <rdf:Description xmlns:pdf='http://ns.adobe.com/pdf/1.3/'>
    <pdf:CreationDate>2004-09-08T16:14:07Z</pdf:CreationDate>
    <pdf:ModDate>2004-09-08T16:14:07Z</pdf:ModDate>
    <pdf:Producer>Acrobat Distiller 5.0 (Windows)</pdf:Producer>
    <pdf:Author>Herbert Jaeger</pdf:Author>
    <pdf:Creator>Acrobat PDFMaker 5.0 for Word</pdf:Creator>
    <pdf:Title>Exercises for ACS 1, Fall 2003</pdf:Title>
  </rdf:Description>
  ...
  <rdf:Description xmlns:dc='http://purl.org/dc/elements/1.1/'>
    <dc:creator>Herbert Jaeger</dc:creator>
    <dc:title>Exercises for ACS 1, Fall 2003</dc:title>
  </rdf:Description>
</rdf:RDF>
```

- ▷ Example 4.5.7 mixes elements from three different vocabularies:
- ▷ RDF: xmlns:rdf for the “Resource Description Format”,
  - ▷ PDF: xmlns:pdf for the “Portable Document Format”, and
  - ▷ DC: xmlns:dc for the “Dublin Core” vocabulary



©: Michael Kohlhasse

131



This is an excerpt from the document metadata which Acrobat Distiller saves along with each PDF document it creates. It contains various kinds of information about the creator of the document, its title, the software version used in creating it and much more. Document metadata is useful for libraries, bookselling companies, all kind of text databases, book search engines, and generally all institutions or persons or programs that wish to get an overview of some set of books, documents, texts. The important thing about this document metadata text is that it is not written in an arbitrary, PDF-proprietary format. Document metadata only make sense if these metadata are independent of the specific format of the text. The metadata that MS Word saves with each Word document should be in the same format as the metadata that Amazon saves with each of its book records, and again the same that the British library uses, etc.

We will now reflect what we have seen in Example 4.5.7 and fully define the namespacing mechanisms involved. Note that these definitions are technically involved, but conceptually quite natural. As a consequence they should be read more with an eye towards “what are we trying to achieve” than the technical details.

## Mixing Vocabularies via XML Namespaces

- ▷ **Problem:** We would like to reuse elements from different XML vocabularies

What happens if element names coincide, but have different meanings?

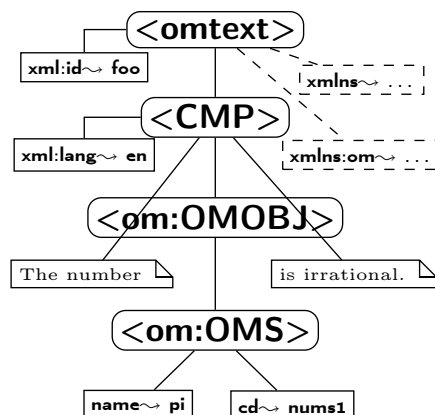
- ▷ **Idea:** Disambiguate them by vocabulary name. (prefix)
- ▷ **Problem:** What if vocabulary names are not unique? (e.g. different versions)
- ▷ **idea:** Use a long string for identification and a short prefix for referencing
- ▷ **Definition 4.5.8** An **XML namespace** is a string that identifies an XML vocabulary. Every element and attribute name in XML consists of a **local name** and a **namespace**.
- ▷ **Definition 4.5.9** **namespace declaration** is an attribute `xmlns:⟨prefix⟩=|` whose value is an **XML namespace**  $n$  on an XML element  $e$ . The first associates the **namespace prefix** `⟨prefix⟩` with the **namespace**  $n$  in  $e$ : Then, any XML element in  $e$  with a **prefixed name** `⟨prefix⟩:⟨name⟩` has **namespace**  $n$  and **local name** `⟨name⟩`.  
A **default namespace declaration** `xmlns= $d$`  on an element  $e$  gives all elements in  $e$  whose name is not **prefixed**, the **namespace**  $d$ .  
**Namespace declarations** on **subtrees** shadow the ones on **supertrees**.



One of the great advantages of viewing marked-up documents as trees is that we can describe subsets of its nodes.

## XPath, A Language for talking about XML Tree Fragments

- ▷ **Definition 4.5.10** The **XML path language** (XPath) is a language framework for specifying fragments of XML trees.
- ▷ **Example 4.5.11**



XPath exp.	fragment
/	root
omtext/CMP/*	all <b>&lt;CMP&gt;</b> children
//@name	the name attribute on the <b>&lt;OMS&gt;</b> element
//CMP/*[1]	the first child of all <b>&lt;CMP&gt;</b> elements
//*[@cd='nums1']	all elements whose cd has value nums1



An XPath processor is an application or library that reads an XML file into a **DOM** and given an XPath expression returns (pointers to) the set of nodes in the **DOM** that satisfy the expression.

## Computing with XML in python (XPath)

▷ say we have an XML tree:

```
>>> f = StringIO('<foo><bar></bar></foo>')
>>> tree = etree.parse(f)
```

▷ then `xpath()` selects the list of matching elements for an XPath:

```
>>> r = tree.xpath('/foo/bar')
>>> len(r)
1
>>> r[0].tag
```

▷ and we can do it again,...

```
>>> r = tree.xpath('bar')
>>> r[0].tag
'bar'
```

▷ The `xpath()` method has support for XPath variables:

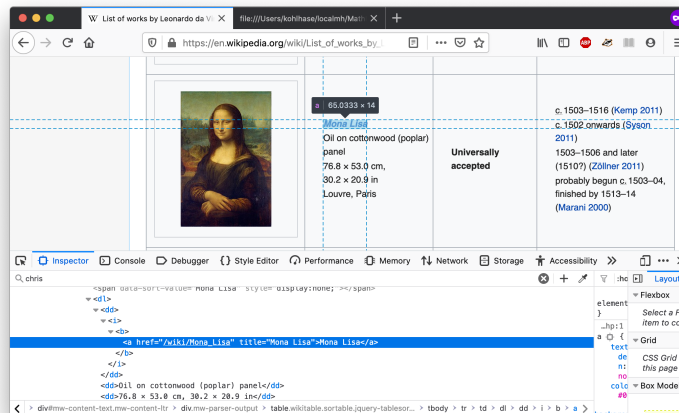
```
>>> expr = "//*[local-name() = $name]"
>>> print(root.xpath(expr, name = "foo")[0].tag)
foo
>>> print(root.xpath(expr, name = "bar")[0].tag)
bar
```



## XPath Example: Scraping Wikipedia

▷ **Example 4.5.12 (Extracting Information from HTML)**

- ▷ We want a list of all titles of paintings by Leonardo da Vinci
- ▷ open [https://en.wikipedia.org/wiki/List\\_of\\_works\\_by\\_Leonardo\\_da\\_Vinci](https://en.wikipedia.org/wiki/List_of_works_by_Leonardo_da_Vinci) in Firefox (save it into a file [mona.html](#))
- ▷ call [DOM](#) inspector to get an idea of the XPath of titles (bottom line)



The path is `table > tbody > tr > td > dl > dd > i > b > a`

**Alternatively:** right-click on highlighted line, `~` "copy" `~` "XPath", gives `/html/body/div[3]/div[3]/div[4]/div/table[4]/tbody/tr[3]/td[2]/dl/dd/i/b/a`

▷ **Idea:** we want to use the second table cells `td[2]`

▷ Program it in python using the `lxml` library: `titles` is list of title strings.

```
from lxml import html

with open('mona.html', 'r') as m:
    str = m.read()
tree = html.fromstring(str)
titles=tree.xpath('//table/tr/td[2]/i/b/a/text()')
```



And now, if you still need proof that XML is really used almost everywhere, here is the ultimate example

## XML is Everywhere (E.g. Office Suites)

▷ **Example 4.5.13 (MS Office uses XML)** The MS Office suite and LibreOffice use compressed XML as an **electronic document** format.

1. Save a MS Office file `test.docx`, add the extension `.zip` to obtain `test.docx.zip`.
2. uncompress with `unzip` (UNIX) or open File Explorer, right-click `~` "Extract All" (Windows)
3. you obtain a folder with 15+ files, the content is in `word/contents.xml`
4. other files have packaging information, images, and other objects.

⚠ this is huge and offensively ugly (but you have everything you wanted and more)





## Chapter 5

# Basic Concepts of the World Wide Web

We will now present a very brief introduction into the concepts, mechanisms, and technologies that underlie the [World Wide Web](#) – and thus [web applications](#), which are our interest here.

### 5.1 Preliminaries

The [WWWeb](#) is the hypertext/multimedia part of the Internet. It is implemented as a service on top of the Internet (at the application level) based on specific protocols and markup formats for documents.

#### The Internet and the Web

- ▷ **Definition 5.1.1** The [Internet](#) is a worldwide computer network that connects hundreds of thousands of smaller networks. (The mother of all networks)
- ▷ **Definition 5.1.2** The [World Wide Web](#) ([WWW](#) or [WWWeb](#)) is an open source information space where documents and other web resources are identified by [URLs](#), interlinked by hypertext links, and can be accessed via the [Internet](#).
- ▷ The [WWW](#) is the multimedia part of the [Internet](#), they form critical infrastructure for modern society and commerce.
- ▷ The Internet/WWW is huge:

Year	Web	Deep Web	eMail
1999	21 TB	100 TB	11TB
2003	167 TB	92 PB	447 PB
2010	????	?????	?????

- ▷ We want to understand how it works (services and scalability issues)

Given this recap we can now introduce some vocabulary to help us discuss the phenomena.

### Concepts of the World Wide Web

- ▷ **Definition 5.1.3** A **web page** is a document (usually marked up in HTML) on the **WWW** that can include multimedia data and hyperlinks.
- ▷ **Definition 5.1.4** A **web site** is a collection of related **web pages** usually designed or controlled by the same individual or company.
- ▷ a web site generally shares a common domain name.
- ▷ **Definition 5.1.5** A **hyperlink** is a reference to data that can immediately be followed by the user or that is followed automatically by a user agent.
- ▷ **Definition 5.1.6** A collection text documents with hyperlinks that point to text fragments within the collection is called a **hypertext**. The action of following hyperlinks in a hypertext is called **browsing** or **navigating** the hypertext.
- ▷ In this sense, the **WWW** is a multimedia hypertext.



## 5.2 Addressing on the World Wide Web

The essential idea is that the **World Wide Web** consists of a set of resources (documents, images, movies, etc.) that are connected by links (like a spider-web). In the **WWW**, the links consist of pointers to addresses of resources. To realize them, we only need addresses of resources (much as we have IP numbers as addresses to hosts on the Internet).

### Uniform Resource Identifier (URI), Plumbing of the Web

- ▷ **Definition 5.2.1** A **uniform resource identifier (URI)** is a global identifiers of local or network-retrievable documents, or media files (**web resources**). **URIs** adhere a uniform syntax (grammar) defined in RFC-3986 [BLFM05]. Grammar Rules contain:

$$\begin{aligned} \text{URI} &::= \text{scheme}, ':', \text{hierPart}, ['?' \text{ query}], ['#' \text{ fragment}] \\ \text{hier - part} &::= '//' \text{ pathAempty} \mid \text{pathAbsolute} \mid \text{pathRootless} \mid \text{pathEmpty} \end{aligned}$$

A **URI** is made up of the following **component**:



- ▷ a **scheme** that specifies the protocol governing the resource
  - ▷ an **authority**: the host (authentication there) that provides the resource.
  - ▷ a **path** in the hierarchically organized resources on the host.
  - ▷ a **query** in the non-hierarchically organized part of the host data.
  - ▷ a **fragment** identifier in the resource.
- ▷ **Example 5.2.2** The following are two example **URIs** and their component parts:

```

http://example.com:8042/over/there?name=ferret#nose
|-----|-----|-----|-----|
|       |       |       |       |
|scheme|authority|path  |query |fragment|
|-----|-----|-----|-----|
|       |       |       |       |
mailto:michael.kohlhase@fau.de

```

**Note:** URIs only **identify** documents, they do not have to be provide access to them (e.g. in a browser).


©: Michael Kohlhase
139


The definition above only specifies the structure of a **URI** and its functional parts. It is designed to cover and unify a lot of existing addressing schemes, including **URLs** (which we cover next), ISBN numbers (book identifiers), and mail addresses.

In many situations **URIs** still have to be entered by hand, so they can become quite unwieldy. Therefore there is a way to abbreviate them.

### ▷ Relative URIs

▷ **Definition 5.2.3** **URIs** can be abbreviated to **relative URIs**; missing parts are filled in from the context.

▷ **Example 5.2.4** Relative **URIs** are more convenient to write

relative <b>URI</b>	abbreviates	in context
#foo	«current – file»#foo	current file
bar.txt	file:///home/kohlhase/foo/bar.txt	file system
../bar/bar.html	http://example.org/bar/bar.html	on the web

▷ **Definition 5.2.5** To distinguish them from **relative URIs**, we call **URIs** **absolute URIs**.


©: Michael Kohlhase
140


The important concept to grasp for relative **URIs** is that the missing parts can be reconstructed from the context they are found in: the document itself and how it was retrieved.

For the file system example, we are assuming that the document is a file `foo.html` that was loaded from the file system – under the file system URI `file:///home/kohlhase/foo/foo.html` – and for the web example via the URI `//example.org/foo/foo.html`. Note that in the last example, the relative URI `../bar/` goes up one segment of the path component (that is the meaning of `../`), and specifies the file `bar.html` in the directory `bar`.

But **relative URIs** have another advantage over **absolute URIs**: they make a **web page** or **web site** easier to move. If a web site only has links using **relative URIs** internally, then those do not mention e.g. **authority** (this is recovered from context and therefore variable), so we can freely move the web-site e.g. between domains.

Note that some forms of **URIs** can be used for actually locating (or accessing) the identified resources, e.g. for retrieval, if the resource is a document or sending to, if the resource is a mailbox. Such **URIs** are called “uniform resource *locators*”, all others “uniform resource *locators*”.

## Uniform Resource Names and Locators

- ▷ **Definition 5.2.6** A **uniform resource locator (URL)** is a **URI** that gives access to a web resource, by specifying an access method or location. All other **URIs** are called **uniform resource names (URN)**.
- ▷ **Idea:** A **URN** defines the identity of a resource, a **URL** provides a method for finding it.
- ▷ **Example 5.2.7** The following **URI** is a **URL** (try it in your browser)  
`http://kwarc.info/kohlhase/index.html`
- ▷ **Example 5.2.8** `urn:isbn:978-3-540-37897-6` only identifies [Koh06] (it is in the library)
- ▷ **Example 5.2.9** **URNs** can be turned into **URLs** via a catalog service, e.g.  
`http://wm-urn.org/urn:isbn:978-3-540-37897-6`
- ▷ **Note:** **URIs** are one of the core features of the web infrastructure, they are considered to be the **plumbing of the WWWeb**. (direct the flow of data)



Historically, started out as **URLs** as short strings used for locating documents on the Internet. The generalization to identifiers (and the addition of **URNs**) as a concept only came about when the concepts evolved and the application layer of the Internet grew and needed more structure.

Note that there are two ways in **URI** can fail to be resource locators: first, the scheme does not support direct access (as the ISBN scheme in our example), or the scheme specifies an access method, but address does not point to an actual resource that could be accessed. Of course, the problem of “dangling links” occurs everywhere we have addressing (and change), and so we will neglect it from our discussion. In practice, the **URL/URN** distinction is mainly driven by the scheme part of a **URI**, which specifies the access/identification scheme.

## Internationalized Resource Identifiers

- ▷ **Remark 5.2.10** **URIs** are ASCII strings.
- ▷ **Problem:** This is awkward e.g. for *France Télécom*, worse in Asia.
- ▷ **Solution?:** Use **unicode** (no, too young/unsafe)
- ▷ **Definition 5.2.11** **Internationalized resource identifiers (IRIs)** extend the ASCII-based **URIs** to the **universal character set**.
- ▷ **Definition 5.2.12** **URI encoding** maps non-ASCII characters to a **ASCII** strings:
  1. map character to its UTF-8 representation
  2. represent each **byte** of the UTF-8 representation by three characters.
  3. The first character is the percent sign (%),
  4. and the other two characters are the **hexadecimal** representation of the byte.

**URI decoding** is the dual operation.

▷ **Example 5.2.13** The letter “f” (U+ 142) would be represented as %C5%82.

▷ **Example 5.2.14** `http://www.Übergrößen.de` becomes  
`http://www.%C3%9Cbergr%C3%B6%C3%9Fen.de`

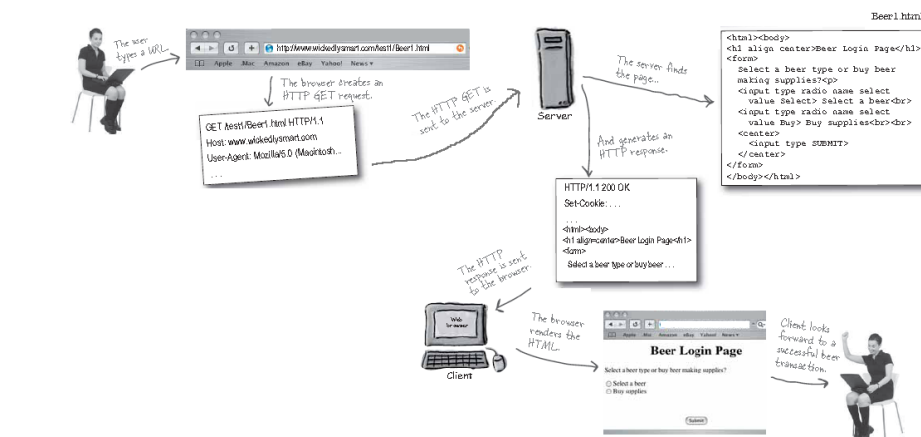
▷ **Remark 5.2.15** Your browser can still show the **URI-decoded** version (so you can read it)



## 5.3 Running the World Wide Web

The infrastructure of the **WWWeb** relies on a client-server architecture, where the servers (called **web servers**) provide documents and the clients (usually **web browsers**) present the documents to the (human) users. Clients and servers communicate via the http protocol. We give an overview via a concrete example before we go into details.

### The World Wide Web as a Client/Server System



The web browser communicates with the web server through a specialized protocol, the hypertext transfer protocol, which we cover now.

### HTTP: Hypertext Transfer Protocol

▷ **Definition 5.3.1** The **Hypertext Transfer Protocol (HTTP)** is an application layer protocol for distributed, collaborative, hypermedia information systems.

▷ June 1999: **HTTP/1.1** is defined in RFC 2616 [Fie+99].

**Definition 5.3.2** **HTTP** is used by a client (called **user agent**) to access web

resources (addressed by **uniform resource locators (URLs)**) via a **HTTP request**. The **web server** answers by supplying the resource

- ▷ **Definition 5.3.3** Most important **HTTP request methods** (5 more less prominent)

<b>GET</b>	Requests a representation of the specified resource.	<b>safe</b>
<b>PUT</b>	Uploads a representation of the specified resource.	<b>idempotent</b>
<b>DELETE</b>	Deletes the specified resource.	<b>idempotent</b>
<b>POST</b>	Submits data to be processed (e.g., from a web form) to the identified resource.	

- ▷ **Definition 5.3.4** We call a **HTTP request safe**, iff it does not change the state in the web server. (except for server logs, counters,...; no side effects)
- ▷ **Definition 5.3.5** We call a **HTTP request idempotent**, iff executing it twice has the same effect as executing it once.
- ▷ **HTTP** is a stateless protocol (very memory-efficient for the server.)



Finally, we come to the last component, the web server, which is responsible for providing the **web page** requested by the user.

## Web Servers

- ▷ **Definition 5.3.6** A **web server** is a network program that delivers **web resources** to and receives content from user agents via the **Hypertext Transfer Protocol (HTTP)**.
- ▷ **Example 5.3.7 (Common Web Servers)**
- ▷ **apache** is an open source web server that serves about 50% of the **WWW**.
  - ▷ **nginx** is a lightweight open source web server (ca. 35%)
  - ▷ **IIS** is a proprietary server provided by Microsoft.
- ▷ **Definition 5.3.8** A **web server** can **host** – i.e serve resources for – multiple domains (via configurable **hostnames**) that can be addressed in the **authority components** of **URLs**. This usually includes the special **hostnamelocalhost** which is interpreted as “this computer”.
- ▷ Even though web servers are very complex software systems, they come preinstalled on most UNIX systems and can be downloaded for Windows [Xam].



Now that we have seen all the components we fortify our intuition of what actually goes down the net by tracing the http messages.

### Example: An http request in real life

- ▷ Send off a GET request for `http://www.nowhere123.com/doc/index.html`

```
GET /docs/index.html HTTP/1.1
Host: www.nowhere123.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
(blank line)
```

- ▷ The response from the server

```
HTTP/1.1 200 OK
Date: Sun, 18 Oct 2009 08:56:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Sat, 20 Nov 2004 07:16:26 GMT
ETag: "10000000565a5-2c-3e94b66c2e680"
Accept-Ranges: bytes
Content-Length: 44
Connection: close
Content-Type: text/html
X-Pad: avoid browser bug

<html><body><h1>It works!</h1></body></html>
```





## Chapter 6

# Web Applications

In this Chapter we will see how we can turn HTML pages into web-based applications that can be used without having to install additional software.

### Web Applications: Using Applications without Installing

▷ **Definition 6.0.1** A **web application** is a program that runs on a **web server** and delivers its **user interface** as a **web site** consisting of programmatically generated **web pages** using a **web browser** as the **client**.

▷ **Example 6.0.2** Commonly used web applications include

- ▷ <http://ebay.com>; auction pages are generated from databases.
- ▷ <http://www.weather.com>; weather information generated from weather feeds.
- ▷ <http://slashdot.org>; aggregation of news feeds/discussions.
- ▷ <http://github.com>; source code hosting and project management.
- ▷ <http://studon>; course/exam management from students records.

**Common Traits:** pages generated from databases and external feeds, content submission via HTML forms, file upload, dynamic HTML.



©: Michael Kohlhase

147

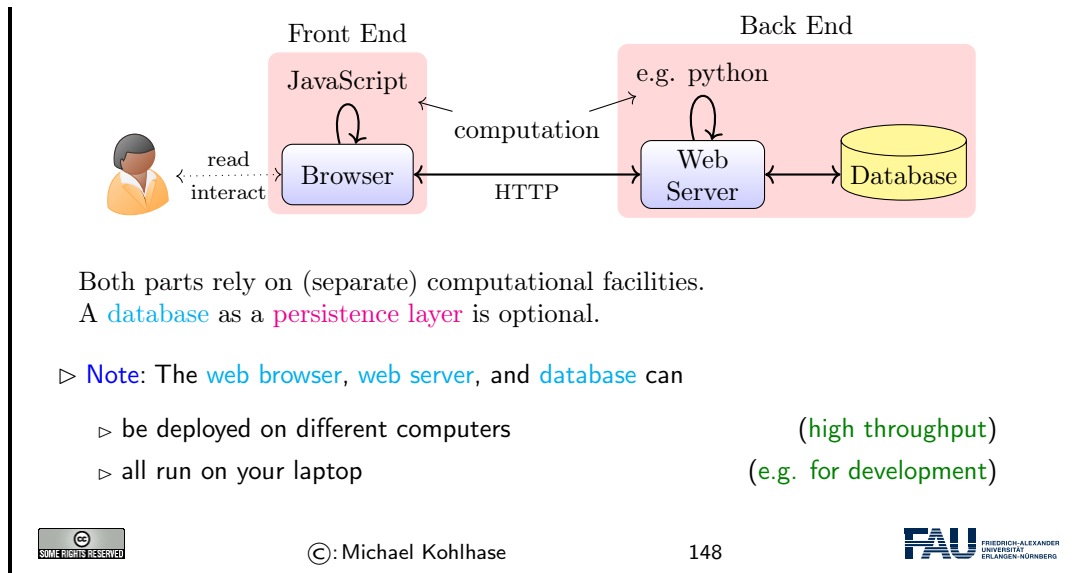


We have seen that **web applications** are a common way of building **application software**. To understand how this works let us now have a look at the components.

### ▷ Anatomy of a Web Application

▷ **Definition 6.0.3** A **web application** consists of two parts

- ▷ A **front end** that handles the user interaction.
- ▷ A **back end** that stores, computes and serves the application content.



To understand **web applications**, we will first need to understand

1. how we can express web pages in HTML and (see Section 4.3) interact with them for data input (we recap this in Section 6.1)
2. the basics of how the World-Wide Web works as a distribution framework (see Chapter 5),
3. how we can generate HTML documents programmatically (in our case in **python**; see Section 6.2) as answer pages, and finally
4. how we can make HTML pages dynamic by client-side manipulation (see Section 6.4).

## 6.1 Recap: HTML Forms Data Transmission

The first two requirement for web applications above are already met by HTML in terms of HTML forms (see slide 110 ff.). Let us recap and extend

### Recap HTML Forms: Submitting Data to the Web Server


- ▷ **Recall:** HTML forms collect data via named input elements, the submit event triggers a **HTTP** request to the URL specified in the action attribute.
- ▷ **Example 6.1.1** Forms contain input fields and explanations.

```
<form name="input" action="login.html" method="get">
  Username: <input type="text" name="user"/>
  Password: <input type="password" name="pass"/>
  <input type="submit" value="Submit"/>
</form>
```

yields the following in a **web browser**:

Username:  Password:

Pressing the submit button activates a **HTTP GET** request to the **URL** `login.html?user=⟨name⟩&pass=⟨passwd⟩`

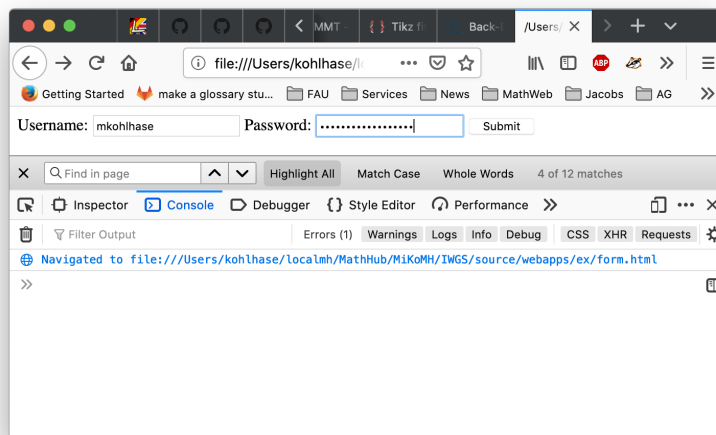
- ▷ : Never use the **GET** method for submitting passwords (see below)



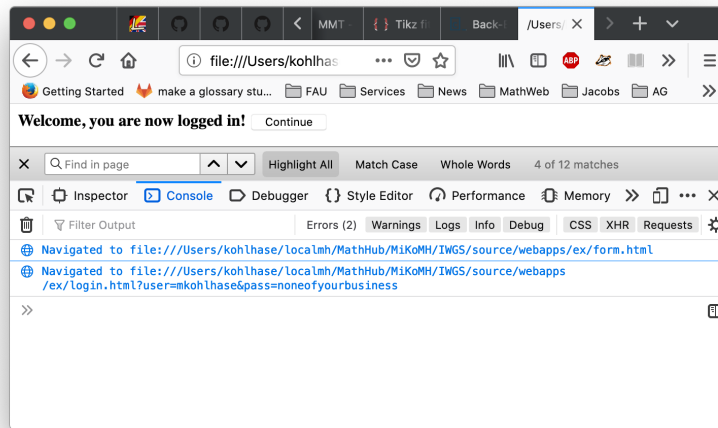
We can now use the tools any modern browser supplies to check up on this claim. In fact, using the browser tools is essential for advanced web development. Here we use the web console, that monitors any activity, to check upon what really happens when we interact with the web page.

## Checking up on the Transmission

- ▷ Let's verify the claims above using browser tools (here the web console)
- ▷ Loading the file and filling in the form: (console logs file URI)



- ▷ After submitting the form: (console logs the **HTTP** request)



A side effect of re-playing our development in the browser is that we see another type of **input element**: A password field, which hides user input from unauthorized eyes. We also see that the **GET** request incorporates the **form data** which contains the password into the URI of the request, which is visible to everyone on the web. We will come back to this problem later.

Let us now look at the data transmission mechanism in more detail to see what is actually transmitted and how.

## HTML Forms and Form Data Transmission

- ▷ We specify the **HTTP** communication of HTML forms in detail.
- ▷ **Definition 6.1.2** The HTML form element groups the layout and input elements:
  - ▷ `<form action="⟨URI⟩" method="⟨req⟩">` specifies the **form action** in terms of a **HTTP request** `⟨req⟩` to the **URI** `⟨URI⟩`.
  - ▷ The **form data** consists of a string `⟨data⟩` of the form  $n_1=v_1 \& \dots \& n_k=v_k$ , where
    - ▷  $n_i$  are the values of the **name** attributes of the input fields
    - ▷ and  $v_i$  are their values at the time of submission.
  - ▷ `<input type="submit" .../>` triggers the **form action**: it composes a **HTTP request**
    - ▷ If `⟨req⟩` is **get** (the default), then the browser issues a **GET** request `⟨URI⟩?⟨data⟩`.
    - ▷ If `⟨req⟩` is **post**, then the browser issues a **POST** request to `⟨URI⟩` with document content `⟨data⟩`.
- ▷ We now also understand the form action, but should we use **GET** or **POST**.




To understand whether we should use the [GET](#) or [POST](#) methods, we have to look into the details, which we will now summarize.

### Practical Differences between HTTP GET and POST

#### ▷ Observation 6.1.3 (Using GET vs. POST in HTML Forms)

	<a href="#">GET</a>	<a href="#">POST</a>
<i>Caching</i>	<i>possible</i>	<i>never</i>
<i>Browser History</i>	<i>Yes</i>	<i>never</i>
<i>Bookmarking</i>	<i>Yes</i>	<i>No</i>
<i>Change Server Data</i>	<i>No</i>	<i>Yes</i>
<i>Size Restrictions</i>	<i>≤ 2KB</i>	<i>No</i>
<i>Encryption</i>	<i>No</i>	<i>HTTPS</i>

**Upshot:** [HTTP GET](#) is more convenient, but less potent.

- ▷  **Always use [POST](#) for sensitive data** (passwords, personal data, etc.)!  
[GET](#) data is part of the URI and thus unencrypted, [POST](#) data via [HTTPS](#) is.



## 6.2 Generating HTML on the Server

As the [WWWeb](#) is based on a [client-server architecture](#), computation in web applications can be executed either on the [client](#) (the [web browser](#)) or the [server](#) (the [web server](#)). For both we have a special technology; we start with computation on the [web server](#).

### Server-Side Scripting: Programming Web pages

- ▷ **Idea:** Why write HTML pages if weperl can also program them! (easy to do)
- ▷ **Definition 6.2.1** A [server-side scripting framework](#) is a [web server](#) extension that generates [web pages](#) upon [HTTP](#) GET requests.
- ▷ **Example 6.2.2** perl is a scripting language with good string manipulation facilities. [PERL CGI](#) is an early [server-side scripting framework](#) based on this.
- ▷ **Observation 6.2.3** [Server-side scripting frameworks](#) allow to make use of external resources (e.g. databases or data feeds) and computational services during [web page](#) generation.
- ▷ **Observation 6.2.4** A [server-side scripting frameworks](#) solves two problems:
  1. making the development of functionality that generates HTML pages convenient and efficient, usually via a [template engine](#), and
  2. binding such functionality to [URLs](#) – the [routes](#), we call this [routing](#).



We will look at the second problem: [routing](#) first. There is a dedicated [python library](#) for that.

### 6.2.1 Routing and Argument Passing in Bottle

We will now introduce the [bottle library](#), which supplies a lightweight [web server](#) and [server-side scripting framework](#) implemented in [python](#). It is already installed on the JupyterLab cloud IDE at <http://jupyter.kwarc.info>. To install it on your laptop, just type `pip install bottle` in a shell.

#### The Web Server and Routing in Bottle WSGI

- ▷ **Definition 6.2.5** [Serverside routing](#) (or simply [routing](#)) is the process by which a [web server](#) connects a [HTTP](#) request to a function (called the [route function](#)) that provides a [web resource](#). A single [URI path/route function](#) pair is called a [route](#).
- ▷ The [bottle WSGI library](#) supplies a simple [python web server](#) and [routing](#).
  - ▷ The `run(⟨⟨keys⟩⟩)` function starts the [web server](#) with the configuration given in `⟨⟨keys⟩⟩`.
  - ▷ The `@route` decorator connects [path components](#) to [python functions](#) that return [strings](#).
- ▷ **Example 6.2.6** (A Hello World route) for [localhost](#) on [port 8080](#)

```
from bottle import route, run
```

```
@route('/hello')
```

```
def hello():
```

```
    return "Hello IWGS!"
```

```
run(host='localhost', port=8080, debug=True)
```

This [web server](#) answers to [HTTP GET](#) requests for the URL `http://localhost:8080/hello`



Let us understand Example 6.2.6 [line-by-line](#): The first line imports the [library](#). The second establishes a [route](#) with the name `hello` and binds it to the [python function](#) `hello` in [line 3](#) and [4](#). The last [line](#) configures the [bottle web server](#): it serves content via the [HTTP](#) protocol for [localhost](#) on [port 8080](#).

So, if we run the program from Example 6.2.6, then we obtain a [web server](#) that will answer [HTTP GET](#) requests to the [URL](#) `http://localhost:8080/hello` with a [HTTP](#) answer with the content `Hello IWGS!`.

To keep the example simple, we have only returned a text string; A realistic application would have generated a full HTML page (see below).

In the last [line](#) of Example 6.2.6, we have also configured the [bottle web server](#) to use “debug mode”, which is very helpful during early development.

In this mode, the bottle [web server](#) is much more verbose and provides helpful debugging information whenever an error occurs. It also disables some optimisations that might get in your way and adds some checks that warn you about possible misconfiguration.

Note that debug mode should be disabled in a production server.

But we can do more with routes!

## Dynamic Routes in Bottle

▷ **Definition 6.2.7** A [dynamic route](#) is a route annotation that contains [named wildcards](#), which can be picked up in the route function.

▷ **Example 6.2.8** Multiple `@route` annotations per [route function](#)  $f$  are allowed  $\leadsto$  the [web application](#) uses  $f$  to answer multiple [URLs](#).

```
@route('/')
@route('/hello/<name>')
def greet(name='Stranger'):
    return (f'Hello {name}, how are you?')
```

With the [wildcard](#) `<name>` we can bind the [route function](#) `greet` to all [paths](#) and via its argument `greet` customize the greeting.

[Concretely](#): a [HTTP](#) GET request to

- ▷ `http://localhost` is answered with `Hello Stranger, how are you?`.
- ▷ `http://localhost/hello/MiKo` is answered with `Hello MiKo, how are you?`.

Requests to e.g. `http://localhost/hello` or `http://localhost/hello/prof/kohlhase` lead to errors. (404: not found)



Often we want to have more control over the routes. We can get that by filters, which can involve data types and/or [regular expression](#).

## Restricting Dynamic Routes

▷ **Definition 6.2.9** [Dynamic routes](#) can be restricted by a [route filter](#) to make them more selective.

▷ **Example 6.2.10 (Concrete Filters)** `:int` for integers or `:re:<<regex>>` for [regular expressions](#)

```
@route('/tel/<id:int>') # local number
@route('/tel/<num:re:^(+|[1-9]){1}[0-9]{3,14}$>') # international
```

different route filters allow to classify paths and treat them differently.

▷ **Example 6.2.11** Multiple [named wildcards](#) are also possible, with and without filters:

```
@route('/<action>/<user:re:[a-z]+>') # matches /follow/miko
def user_api(action, user):
    ...
```



We have already seen above that we want to use **HTTP GET** and **POST** request for different facets of transmitting HTML **form data** to the **web server**. This is supported by **bottle WSGI** in two ways: we can specify the **HTTP method** of a **route** and we have access to the **form data** (and other aspects of the request).

## Method-Specific Routes: HTTP GET and POST

▷ **Definition 6.2.12** The `@route` decorator takes a **method** keyword to specify the **HTTP** request **method** to be answered. (**HTTP get** is the default)

- ▷ `@get(⟨path⟩)` abbreviates `@route(⟨path⟩, method="GET")`
- ▷ `@post(⟨path⟩)` abbreviates `@route(⟨path⟩, method="POST")`

▷ **Example 6.2.13 (Login 1)** Managing logins with **HTTP GET** and **POST**.

```
from bottle import get, post, request # or route

@get('/login') # or @route('/login')
def login():
    return '''
        <form action="/login" method="post">
            Username: <input name="username" type="text" />
            Password: <input name="password" type="password" />
            <input value="Login" type="submit" />
        </form>
    '''
```

**Note:** We can also have a **POST** request to the same **path**; we use that for handling the **form data** transmitted by the **POST** action on submit. (up next)



Recall that we have already seen most of this in slide 149. The only new thing is that we return the HTML as a string in the route function as a request to a **HTTP GET** request. Now comes the interesting part: the form uses the **POST method** in the form action and we have to specify a route for that. Recall from Observation 6.1.3 that this allows for encrypted transmission, so we are less naive than our solution from slide 149.

## ▷ Bottle Request: Dealing with POST Data

▷ **Recall:** from a HTML form we get a **GET** or **POST** request with **form data**  $n_1=v_1 \& \dots \& n_k=v_k$  (here `user=mkohlhase&login=noneofyourbusiness`)

▷ **Bottle WSGI** provides the request object for dealing with **HTTP** request data.

▷ **Example 6.2.14 (Login 2)** Continuing from Example 6.2.13: we parse the request transmitted request and check password information

```
@post('/login') # or @route('/login', method='POST')
def do_login():
    username = request.forms.get('username')
    password = request.forms.get('password')
```

```

if check_login(username, password):
    return "<p>Your login information was correct.</p>"
else:
    return "<p>Login failed.</p>"

```

We assume a python function `check_login` that checks login credentials, and keeps a list of logged-in users.



The main new thing in Example 6.2.14 is that we use the `request.forms.get` method to query the request object that comes with the [HTTP](#) request triggering the route for the [form data](#).

## 6.2.2 Templating in Python via STPL

In IWGS, we use python for programming, so let us see how we would generate HTML pages in python.

### What would we do in python

#### ▷ Example 6.2.15 (HTML Hello World in python)

```

print("<html>")
print("<body>Hello world</body>")
print("</html>")

```

**Problem 1:** Most [web page](#) content is static (page head, text blocks, etc.)

#### ▷ Example 6.2.16 (Python Solution) use python functions:

```

def htmlpage (t,b):
    f"<html><head><title>{t}</title></head><body>{b}</body></html>"
    htmlpage("Hello", "Hello IWGS")

```

**Problem 2:** If HTML markup dominates, want to use a HTML editor (mode)

#### ▷ ▷ e.g. for HTML syntax highlighting/indentation/completion/checking

**Idea:** Embed [program](#) snippets into HTML. (only execute these, copy rest)



We will now formalize and toolify the idea of “embedding code into HTML”. What comes out of this idea is called “templating”. It exists in many forms, and in most programming languages.

### ▷ Template Processing for HTML

▷ **Definition 6.2.17** A **template engine** (or **template processor**) for a document format  $F$  is a system that transforms **template files**, i.e. files with a mixture of program constructs and  $F$ -markup into a  $F$ -document by executing the program constructs (**template processing**).

▷ **Note:** No program code is left in the resulting [web page](#) after generation. (important)

security concern)

- ▷ **Remark:** We will be most interested in HTML [template engines](#).
- ▷ **Observation 6.2.18** We can turn a [template engine](#) into a [server-side scripting framework](#) by employing the [URLs of template files](#) on a server as [routes](#) and extending the [web server](#) by [template processing](#).
- ▷ **Example 6.2.19** PHP (originally “Programmable Home Page Tools”) is a very successful [server-side scripting framework](#) following this model.



©: Michael Kohlhase

160



Naturally, python comes with a [template engine](#) – in fact multiple ones. We will use the one from the bottle web application framework for IWGS.

## stpl: the “Simple Template Engine” from Bottle

- ▷ **Definition 6.2.20** [Bottle WSGI](#) supplies the [template engine](#) stpl (Simple Template Engine). ([documentation at \[STPL\]](#))
  - ▷ stpl uses the template function for [template processing](#) and `{{...}}` to embed into a template string; returns a formatted [unicode](#) string.
- ```
>>> template('Hello {{name}}!', name='World')
u'Hello World!'

>>> my_dict={'number': '123', 'street': 'Fake St.', 'city': 'Fakeville'}
>>> template('I live at {{number}} {{street}}, {{city}}', **my_dict)
u'I live at 123 Fake St., Fakeville'
```



©: Michael Kohlhase

161



The stpl template function is a powerful enabling basic functionality in python, but it does not satisfy our goal of writing “HTML with embedded python”. Fortunately, that can easily be built on top of the template functionality:

## stpl Syntax and Template Files

- ▷ **But what about...**: HTML files with embedded python?
- ▷ stpl uses [template files](#) (extension .tpl) for that.
- ▷ **Definition 6.2.21** A stpl [template file](#) mixes HTML with [stpl python](#):
  - ▷ [stpl python](#) is exactly like python but ignores indentation and closes bodies with end instead.
  - ▷ [stpl python](#) can be embedded into the HTML as
    - ▷ a [code lines](#) starting with a %,
    - ▷ a [code blocks](#) surrounded with `<%` and `%>`, and
    - ▷ an [expressions](#) `{{«exp»}}` as long as `«exp»` evaluates to a string.

▷ **Example 6.2.22** Two [template files](#)

```

<!-- next: a line of python code -->
% course = "Informatische werkzeuge ..."
<p>Some plain text in between</p>
<%
    # A block of python code
    course = name.title().strip()
%>
<p>More plain text</p>

```

```

<ul>
    % for item in basket:
        <li>{{item}}</li>
    % end
</ul>

```



So now, we have template files. But experience shows that template files can be quite redundant; in fact, the better designed the web site we want to create, the more fragments of the template files we want to reuse in multiple places – with and without adaptations to the particular use case.

## Template Functions

▷ **Definition 6.2.23** [stpl python](#) supplies the [template functions](#)

1. `include(⟨tpl⟩,⟨vars⟩)`, where `⟨tpl⟩` is another [template file](#) and `⟨vars⟩` a set of variable declarations (for `⟨tpl⟩`).
2. `defined(⟨var⟩)` for checking definedness `⟨var⟩`
3. `get(⟨var⟩, default=⟨val⟩)`: return the value of `⟨var⟩`, or a default `⟨val⟩`.
4. `setdefault(⟨name⟩,⟨val⟩)`

▷ **Example 6.2.24 (Including Header and Footer in a template)** In a coherent [web site](#), the [web pages](#) often share common header and footer parts. Realize this via the following page template:

```

% include('header.tpl', title='Page Title')
Page Content
% include('footer.tpl')

```

▷ **Example 6.2.25 (Dealing with Variables and Defaults)**

```

% setdefault('text', 'No Text')
<h1>{{get('title', 'No Title')}}</h1>
<p> {{ text }} </p>
% if defined('author'):
    <p>By {{ author }}</p>
% end

```



There is one problem however with web applications that is difficult to solve with the technologies so far. We want web applications to give the user a consistent user experience even though they are made up of multiple [web pages](#). In a regular application we only want to login once and expect the application to remember e.g. our username and password over the course of the various interactions with the system. For web applications this poses a technical problem which we now discuss.

## State in Web Applications and Cookies

- ▷ **Recall:** Web applications contain multiple pages, **HTTP** is a stateless protocol.
- ▷ **Problem:** how do we pass state between pages? (e.g. **username**, **password**)
- ▷ **Simple Solution:** Pass information along in query part of page **URLs**.
- ▷ **Example 6.2.26 (HTTP GET for Single Login)** Since we are generating pages we can generate augmented links  

```
<a href="http://example.org/more.html?user=joe,pass=hideme">... more</a>
```
- Problem:** only works for limited amounts of information and for a single session
- ▷ **Other Solution:** Store state persistently on the client hard disk
- ▷ **Definition 6.2.27** A **cookie** is a text file stored on the client hard disk by the web browser. Web servers can request the browser to store and send cookies.
- ▷ **Note:** cookies are data not programs, they do not generate pop-ups or behave like viruses, but they can include your log-in name and browser preferences.
- ▷ **Note:** cookies can be convenient, but they can be used to gather information about you and your browsing habits.
- ▷ **Definition 6.2.28** **Third party cookies** are used by advertising companies to track users across multiple sites. (but you can turn off, and even delete cookies)



Note that both solutions to the state problem are not ideal, for usernames and passwords the **URL**-based solution is particularly problematic, since **HTTP** transmits **URLs** in **GET** requests without encryption, and in our example passwords would be visible to anybody with a packet sniffer. Here cookies are little better, since they can be requested by any website you visit.

### 6.2.3 Completing the Contact Form

We now come back to our worked HTML example: the contact form from above. Here is the current state:

#### Back to our Contact Form

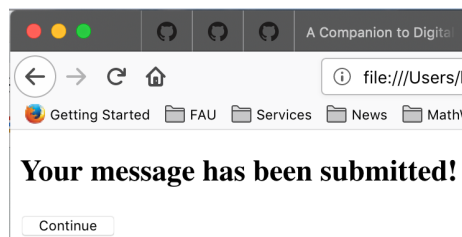
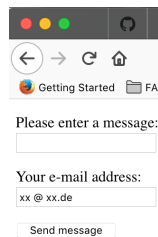
- ▷ A contact form and message receipt (communicate via **HTTP** requests)

```
<title>Contact</title>
<form action="contact-after.html">
  <h2>Please enter a message:</h2>
  <input name="msg" type="text"/>
  <h3>Your e-mail address:</h3>
  <input name="addr" type="text"
        value="xx @ xx.de"/>
  <br/>
  <input type="submit"
        value="Send message"/>
</form>
```

contact-after.html?msg=Hi&addr=foo@bar.de

```
<title>
  Contact — Message Confirmed
</title>
<form action="contact4.html">
  <h2>
    Your message has been submitted!
  </h2>
  <input type="submit"
        value="Continue"/>
</form>
```

contact.html



- ▷ **Problem:** The answer is a static HTML document independent of **form data**.
- ▷ **Solution:** Generate the answer programmatically using the **form data**. (up next)



## Completing the Contact Form

### ▷ Example 6.2.29 (Submitting a Contact Form)

```
from bottle import route, run, debug,
    template, request, get

@get('/contact-after.html')
def new_item():
    data = {'msg': request.GET.msg.strip(),
          'addr': request.GET.addr.strip()}
    send-contact-email(addr,msg)
    return template('contact-after',**data)
```

```
<p>Message submitted!</p>
<table>
  <tr>
    <td>return-address</td>
    <td>{addr}</td>
  </tr>
  <tr>
    <td>text</td>
    <td>{msg}</td>
  </tr>
</table>
```



## Sending off the e-mail

- ▷ We still need to implement the send-contact-email function, ...
- ▷ Fortunately, there is a python package for that: **smtp**lib, which makes this relatively easy. (SMTP  $\hat{=}$  "Simple Mail Transfer Protocol")

▷ **Example 6.2.30 (Continuing)**

```
import smtplib
from email.message import EmailMessage

def send_contact_email(addr, text)
    msg = EmailMessage()
    msg.set_content(text)
    msg['Subject'] = f'Contact from {addr}'
    msg['From'] = addr
    msg['To'] = info@example.org
    s = smtplib.SMTP('smtp.gmail.com', 587)
    s.send_message(msg)
    s.quit()
```

Actually, this does not quite work yet as google requires authentication and encryption, ...; (google for “python smtplib gmail”)



## 6.3 Cascading Stylesheets

In this Section we introduce a technology of digital documents which naturally belongs into Chapter 4: the specification of presentation (layout, colors, and fonts) for marked-up documents.

### 6.3.1 Separating Content from Layout

As the [WWW](#) evolved from a hypertext system purely aimed at human readers to a Web of multimedia documents, where machines perform added-value services like searching or aggregating, it became more important that machines could understand critical aspects [web pages](#). One way to facilitate this is to separate markup that specifies the content and functionality from markup that specifies human-oriented layout and presentation (together called “styling”). This is what “cascading style sheets” set out to do.

Another motivation for CSS is that we often want the styling of a [web page](#) to be customizable (e.g. for vision-impaired readers).


### CSS: Cascading Style Sheets



- ▷ **Idea:** Separate structure/function from appearance.
- ▷ **Definition 6.3.1** The **Cascading Style Sheets** (CSS), is a style sheet language that allows authors and users to attach style (e.g., fonts, colors, and spacing) to HTML and XML documents.
- ▷ **Example 6.3.2** Our text file from Example 4.3.3 with embedded CSS

```

<html>
<head>
  <style type="text/css">
    body {background-color:#d0e4fe;}
    h1 {color:orange;
        text-align:center;}
    p {font-family:"Verdana";
        font-size:20px;}
  </style>
</head>
<body>
  <h1>CSS example</h1>
  <p>Hello IWGS!</p>
</body>
</html>

```




©: Michael Kohlhasse
168


Now that we have seen the example, let us fix the basic terminology of CSS.

## CSS: Rules, Selectors, and Declarations

- ▷ **Definition 6.3.3** A CSS style sheet consists of a sequence of **rules** that in turn consist of a set of **selectors** that determine which XML **elements** the **rule** applies to and a **declaration block** that specifies intended presentation.
- ▷ **Definition 6.3.4** A CSS **declaration block** consists of a semicolon-separated list of **declarations** in curly braces. Each **declaration** itself consists of a **property**, a colon, and a **value**.
- ▷ **Example 6.3.5** In Example 6.3.2 we have three **rules**, they address color and font **properties**:

```

body {background-color:#d0e4fe;}
h1 {color:orange;
    text-align:center;}
p {font-family:"Verdana";

```

**Observation:** In modern **web sites**, CSS contributes as much – if not more – to the appearance as the choice of HTML elements.

In Example 6.3.5 the **selectors** are just **element** names, they specify that the respective **declaration blocks** apply to all elements of this name.

We explore this new technology by way of an example. We rework the title box from the HTML example above – after all treating author/affiliation information as headers is not very semantic. Here we use `div` and `span` elements, which are generic block-level (i.e. paragraph-like) and inline containers, which can be styled via CSS classes. The class `titlebox` is represented by the CSS **selector** `.titlebox`.

### ▷ A Styled HTML Title Box (Source)

- ▷ **Example 6.3.6 (A style Title Box)** The HTML source:

```

<head>
<title>A Styled HTML Title</title>

```

```

<link rel="stylesheet" type="text/css" href="style.css"/>
</head>
<body>
  <div class="titlebox">
    <div class="title">Anatomy of a HTML Web Page</div>
    <div class="author">
      <span class="name">Michael Kohlhas</span>
      <span class="affil">FAU Erlangen–Nuernberg</span>
    </div>
  </div>
  ...

```

And the CSS file referenced in the `<link>` element in [line 3](#):

```

.titlebox {border: 1px solid black;padding: 10px;
           text-align: center
           font-family: verdana;}
.title {font-size: 300%;font-weight: bold}
.author {font-size: 160%;font-style: italic;}
.affil {font-variant: small-caps;}

```



©: Michael Kohlhas

170



And here is the result in the browser:

## A Styled HTML Title Box (Result)



©: Michael Kohlhas

171



### 6.3.2 A small but useful Fragment of CSS

CSS is a huge ecosystem of technologies, which is spread out over about 100 particular specifications – see [CSSa] for an overview.

We will now go over a small fragment of CSS that is already very useful for web applications in more detail and introduce it by example. For a more complete introduction, see e.g. [CSSc].

Recall that [selectors](#) are the part of CSS [rules](#) that determine what elements a [rule](#) affects. We now give the most important cases for our applications.

## CSS Selectors

- ▷ **Question:** Which elements are affected by a CSS [rule](#)?
- ▷ Elements of a given name (optionally with given attributes)
  - ▷ **Selectors:** name  $\hat{=}$  `<elname>`, attributes  $\hat{=}$  `[<attname>=<attval>]`
  - ▷ **Example:** `p[xml:lang='de']` applies to `<p lang="de">...</p>`

- ▷ Any elements with a given class attribute
  - ▷ **Selector:** `.<classname>`
  - ▷ **Example:** `.important` applies to `<el> class='important'>...</el>`
- ▷ The element with a given id attribute
  - ▷ **Selector:** `#<id>`
  - ▷ **Example:** `#myRoot` applies to `<el> id='myRoot'>...</el>`
- ▷ Multiple **selectors** can be combined in a comma-separated list
- ▷ for a full list see [https://www.w3schools.com/cssref/css\\_selectors.asp](https://www.w3schools.com/cssref/css_selectors.asp)



We now come to one of the most important conceptual parts of CSS: the box model. Understanding it is essential for dealing with CSS-based layouts.

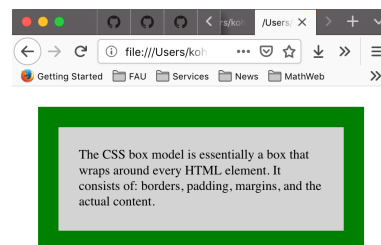
## The CSS Box Model

- ▷ **Definition 6.3.7** For layout, CSS considers all HTML elements as boxes, i.e. document areas with a given **width** and **height**. A CSS **box** has four parts:
  - ▷ **content**: the content of the box, where text and images appear.
  - ▷ **padding**: clears an area around the content. The padding is transparent.
  - ▷ **border** a border that goes around the padding and content.
  - ▷ **margin** clears an area outside the border. The margin is transparent.

The latter three wrap around the **content** and add to its size.

- ▷ all parts of a box can be customized with suitable CSS **properties**:

```
div {
  background-color: lightgrey;
  width: 300px;
  border: 25px solid green;
  padding: 25px;
  margin: 25px;
}
```



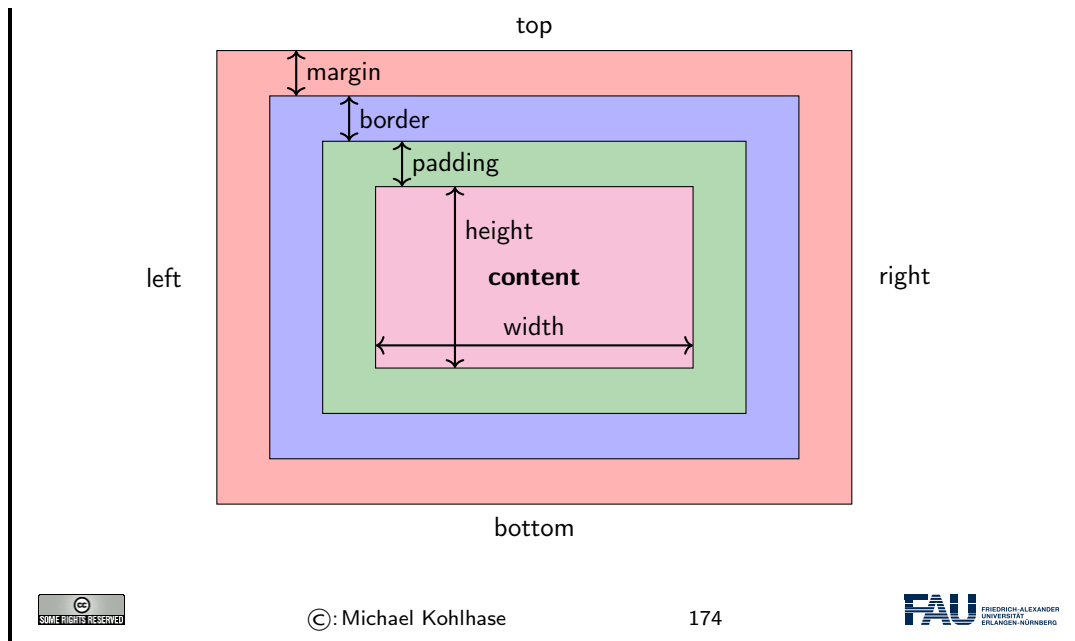
Note that the overall width of the CSS box is 375 pixels.



As a summary of the above, we can visualize the CSS box model in a diagram:

## The CSS Box Model: Diagram

- ▷ The following diagram summarizes the CSS box model



We now come to a topic that is quite mind-boggling at first: The “cascading” aspect of CSS style sheets. Technically, the story is quite simple, there are two independent mechanisms at work:

- *inheritance*: if an element is fully contained in another, the inner (usually) inherits all properties of the outer.
- *rule prioritization*: if more than one selector applies to an element (e.g. one by element name and one by id attribute), then we have to determine what rule applies.

Technically, prioritization takes care of them in an integrated fashion.

### Cascading of selectors in CSS: Prioritization

▷ Multiple CSS selectors apply with the following priorities:

1. important (i.e. marked with !important) before unimportant
2. inline (specified via the style attribute)
3. media-specific rules before general ones
4. user-defined CSS stylesheet (e.g. in the FireFox profile)
5. specialized before general selectors (complicated; see e.g. [CSSb])
6. rule order: later before earlier selectors
7. parent inheritance: unspecified properties are inherited from the parent.
8. style sheet included or referenced in the HTML document.
9. browser default

But do not despair with this technical specification, you do not have to remember it to be effective with CSS practically, because the rules just encode very natural “behavior”. And if you need to understand what the browser – which implements these rules – really sees, use the integrated inspector tool (see slide 180 for details).

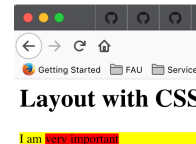
We now look at an example to fortify our intuition.

## Cascading of selectors in CSS: Prioritization Example

▷ **Example 6.3.8** Can you explain the colors in the [web browsers](#) below?

```
<h1>Layout with CSS</h1>
<div id="important" class="blue">
  I am <span class="markedimportant">very important</span>
</div>
```

```
.markedimportant {background-color:red !important}
#important {background-color:green}
.blue {background-color:blue}
#important {background-color:yellow}
```



©: Michael Kohlhasse

176



For instance, the words *very important* get a red background, as the class `markedimportant` is marked as important by the CSS keyword `!important`, which makes (cf. rule 1 above) the color red win against the color yellow inherited from the parent `<div>` element (rule 7 above).

Let us now look at CSS inheritance in a little more detail

## Cascading in CSS: Inheritance

▷ **Definition 6.3.9** If an element is fully contained in another, the inner **inherits** some **properties** (called **inheritable**) of the outer. In a nutshell

- ▷ text-related **properties** are **inheritable**; e.g. color, font, letter-spacing, line-height, list-style, and text-align
- ▷ box-related **properties** are not; e.g. background, border, display, float, clear, height, width, margin, padding, position, and text-align.

**Note:** **Inheritance** is integrated into prioritization (recall case 7. above)

▷ **Inheritance** makes for consistent text **properties** and smaller CSS stylesheets.



©: Michael Kohlhasse

177



So far, we have looked at the mechanics of CSS from a very general perspective. We will now come to a set of CSS behaviors that are useful for specifying layouts of pages and texts.

Recall that CSS is based on the **box** model, which understands HTML elements as boxes, and layouts as properties of **boxes** nested in **boxes** (as the corresponding HTML elements are).

If we can specify how inner boxes float inside outer boxes – via the CSS float rules, we can already do quite a lot, as the following examples show.

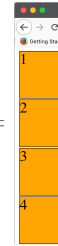
## CSS-Flow: How Boxes Flow to their Place

▷ CSS-Flow describes how different elements are distributed in the visible area (**how they flow; hence the name**)

▷ **Example 6.3.10** Block-level Boxes (here divs) flow to the left

```
<div class="square">1</div>
<div class="square">2</div>
<div class="square">3</div>
<div class="square">4</div>
```

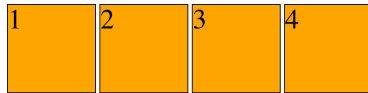
```
.square {font-size:200%;
height:100px;
width:100px;
border:1px solid black;
margin:2px;
background-color:orange;}
```



▷ **Example 6.3.11** `float:left` floats boxes as far as they will go (**without overlap**)

```
<body>
<div class="square">1</div>
<div class="square">2</div>
<div class="square">3</div>
```

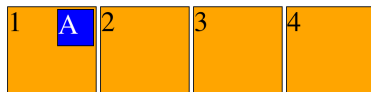
```
.square {font-size:200%;
height:100px;
width:100px;
border:1px solid black;
margin:2px;
background-color:orange;
float:left}
```



▷ **Example 6.3.12** `float:right` in a `div` will float inside the corresponding box

```
<div class="square">1
  <div class="smallsq">A</div>
</div>
<div class="square">2</div>
<div class="square">3</div>
<div class="square">4</div>
```

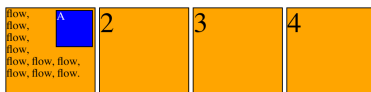
```
.smallsq {color:white;
height: 40px;width: 40px;
border: 1px solid black;
margin: 2px;
background-color: blue;
float: right}
```



▷ **Example 6.3.13** `float:left` will let contents flow around an obstacle

```
<div class="square"
  style="font-size:small">
  <div class="smallsq">A</div>
  flow, flow, flow, flow, flow,
  flow, flow, flow, flow, flow.
</div>
```

```
.smallsq {color:white;
height: 40px;width: 40px;
border: 1px solid black;
margin: 2px;
background-color: blue;
float: right}
```

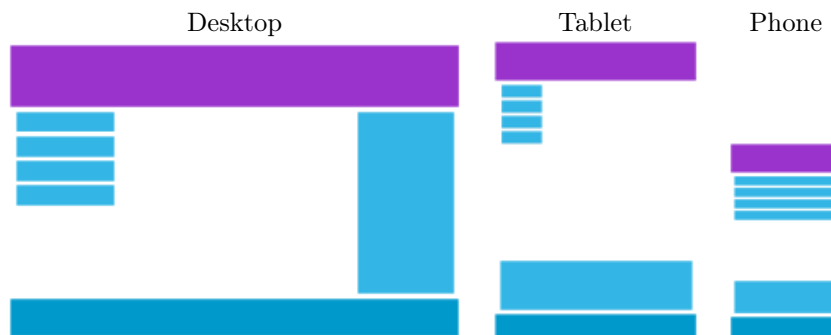


The large space (>2px) is caused because there is no linebreaking

One of the important applications of the content/form separation made possible by CSS is to tailor [web page](#) layout to the screen size and resolution of the device it is viewed on. Of course, it would be possible to maintain multiple layouts for a [web page](#) – one per screensize/resolution class, but a better way is to have one layout that changes according to the device context. This is what we will briefly look at now.

## CSS Application: Responsive Design

- ▷ **Problem:** What is the screen size/resolution of my device?
- ▷ **Definition 6.3.14** **Responsive web design (RWD)** designs web documents so that they can be viewed with a minimum of resizing, panning, and scrolling – across a wide range of devices (from desktop computer monitors to mobile phones)
- ▷ **Example 6.3.15** A **web page** with content blocks



**Implementation:** CSS-based layout with relative sizes and **media queries** – CSS conditionals based on client screen size/resolution/...

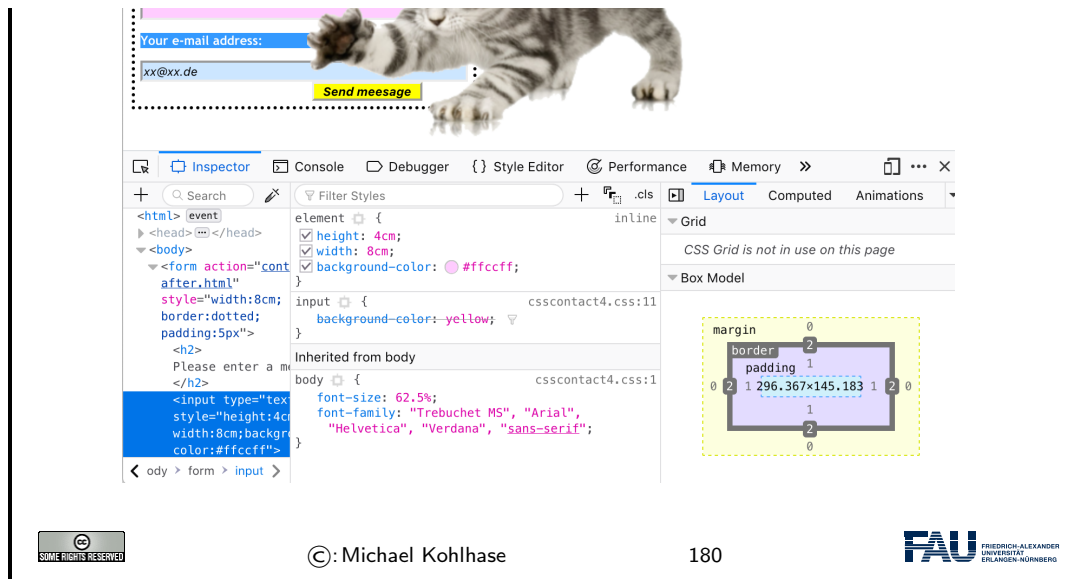


### 6.3.3 CSS Tools

In this Subsection we introduce a technology of digital documents which naturally As CSS has grown to be very complex and moreover, the **browser DOM** of which CSS is part can even be modified after loading the HTML (see Section 6.4), we need tools to help us develop effective and maintainable CSS. We will

#### ▷ But how to find out what the browser really sees?

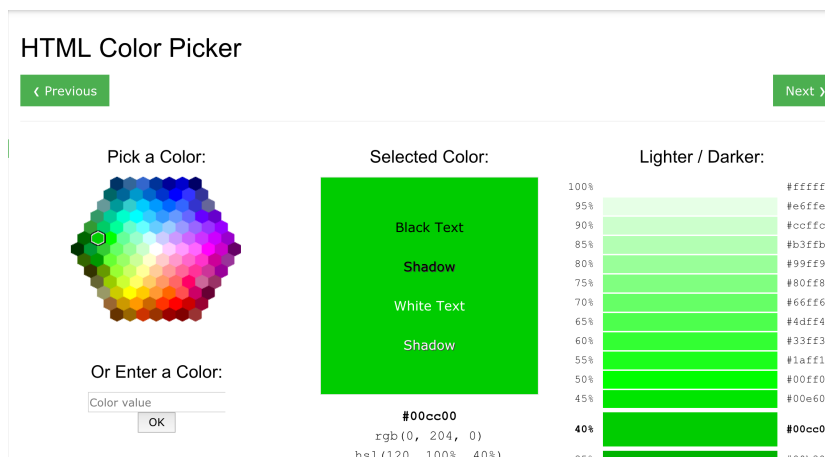
- ▷ CSS has many interesting inheritance rules
- ▷ **Definition 6.3.16** The **page inspector** tool gives you an overview over the internal state of the browser.
- ▷ **Example 6.3.17**



In CSS we can specify colors by various names, but the full range of possible colors can only be specified by numeric (usually [hexadecimal](#)) numbers. For instance in Example 6.3.2, we specified the background color of the page as `#d0e4fe`, which is a pain for the author. Fortunately, there are tools that can help.

## Picking CSS Colors

- ▷ **Problem:** Colors in CSS are specified by funny names (e.g. `CornflowerBlue`) or [hexadecimal](#) numbers, (e.g. `#6495ED`).
- ▷ **Solution:** Use an online color picker, e.g. [https://www.w3schools.com/colors/colors\\_picker.asp](https://www.w3schools.com/colors/colors_picker.asp)



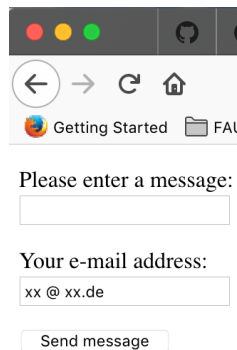
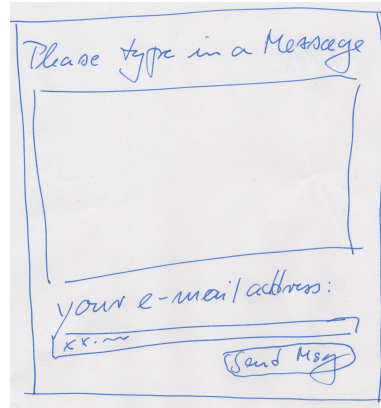
### 6.3.4 Worked Example: The Contact Form

To fortify our intuition on CSS, we take up the “contact form” example from above and improve the layout in a step-by-step process concentrating on one aspect at a time.

#### CSS in Practice: The Contact Form Example (Continued)

▷ Recap: The unstyled contact form – Dream vs. Reality

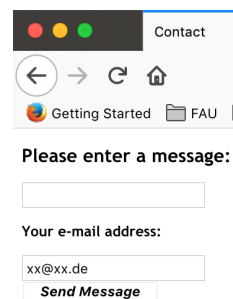
```
<title>Contact</title>
<form action="contact-after.html">
  <h2>Please enter a message:</h2>
  <input name="msg" type="text"/>
  <h3>Your e-mail address:</h3>
  <input name="addr" type="text"
    value="xx @ xx.de"/>
  <br/>
  <input type="submit"
    value="Send message"/>
</form>
```



▷ Add a CSS file with font information

```
<link rel="stylesheet" type="text/css"
  href="css/contact1.css" />
<input class="important" type="submit"
  value="Send Message"/>
```

```
body {font-size: 62.5%;
  font-family: "Trebuchet MS",
    "Arial", "Helvetica",
    "Verdana", "sans-serif"}
.important{font-style: italic;}
input[type="submit"]{font-weight: bold;}
```

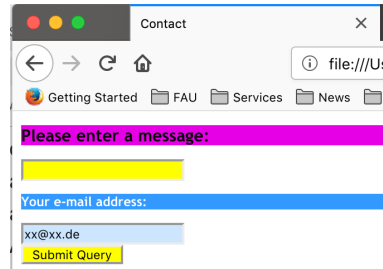


▷ Add lots of color

(oops, what about the size)

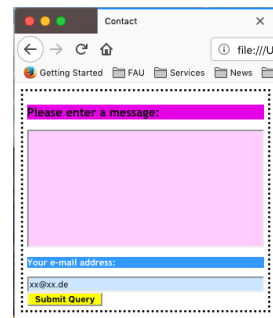
```
<h2>Please enter a message:</h2>
<h3>Your e-mail address:</h3>
<input class="important" name="addr"
style="background-color:#cce6ff"
type="text" value="xx@xx.de"/>
```

```
h2 {background-color: #e600e6;}
h3 {background-color: #3399ff;
color: white;}
input {background-color:yellow}
```



▷ Add size information and a dotted frame

```
<form action="contact-after.html"
style="width:8cm;border:dotted;padding:5px">
<h2>Please enter a message:</h2>
<input name="msg" type="text"
style="height:4cm;width:8cm;
background-color:#ffccff"/>
<br/>
<h3>Your e-mail address:</h3>
<input class="important" name="addr"
type="text"
value="xx@xx.de" style="width:8cm;
background-color:#cce6ff"/>
```

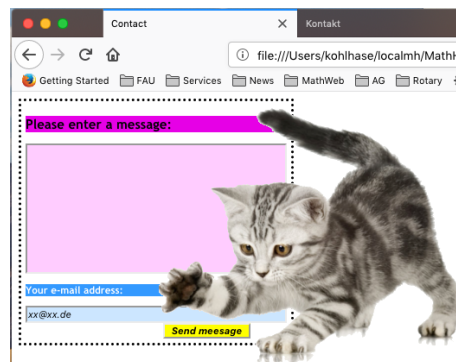


▷ Add a cat that plays with the submit button

(because we can)

```

```



This worked example should be enough to cover most layout needs in practice. Note that in most use cases, these generally layout primitives will have to be combined in different and may be even new ways.

Actually, the last “improvement” may have gone a bit overboard; but we used it to show how absolute positioning of images (or actually any CSS boxes for that matter) works in practice.

## 6.4 Dynamic HTML: Client-side Manipulation of HTML Documents

We now turn to client-side computation:

One of the main advantages of moving documents from their traditional ink-on-paper form into an electronic form is that we can interact with them more directly. But there are many more interactions than just browsing hyperlinks we can think of: adding margin notes, looking up definitions or translations of particular words, or copy-and-pasting mathematical formulae into a computer algebra system. All of them (and many more) can be made, if we make documents programmable. For that we need three ingredients:

- i) a machine-accessible representation of the document structure, and
- ii) a program interpreter in the web browser, and
- iii) a way to send programs to the browser together with the documents.

We will sketch the [WWWeb](#) solution to this in the following.

To understand client-side computation, we first need to understand the way browsers render HTML pages.

### Background: Rendering Pipeline in Browsers

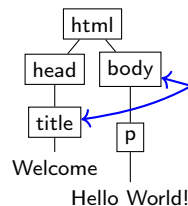
- ▷ **Observation:** The nested, markup codes turn HTML documents into trees.
- ▷ **Definition 6.4.1** The **document object model (DOM)** is a data structure for the HTML document tree together with a standardized set of access methods.
- ▷ **Rendering Pipeline:** Rendering a **web page** proceeds in three steps
  1. the browser receives a HTML document,
  2. parses it into an internal data structure, the **DOM**,
  3. which is then painted to the screen. (**repaint whenever DOM changes**)

#### HTML Document

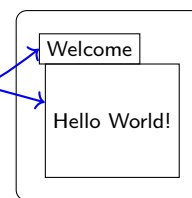
```
<html>
<head>
  <title>Welcome</title>
</head>
<body>
  <p>Hello World!</p>
</body>
</html>
```

parse →

#### DOM



#### Browser



The **DOM** is notified of any user events

(**resizing, clicks, hover,...**)

The most important concept to grasp here is the tight synchronization between the **DOM** and the screen. The **DOM** is first established by parsing (i.e. interpreting) the input, and is synchronized with with the browser UI and document viewport. As the **DOM** is persistent and synchronized, any change in the DOM is directly mirrored in the browser viewpoint, as a consequence we only

need to change the **DOM** to change its presentation in the browser. This exactly the purpose of the client side scripting language, which we will go into next.

### 6.4.1 JavaScript in HTML

#### Dynamic HTML

- ▷ **Idea:** generate parts of the **web page** dynamically by manipulating the DOM.
- ▷ **Definition 6.4.2** JavaScript is an object-oriented scripting language mostly used to enable programmatic access to the DOM in a web browser.
- ▷ JavaScript is standardized by ECMA in [Ecm].
- ▷ **Example 6.4.3** We write the some text into a HTML document object (the document API)

```
<html>
<head>
  <script type="text/javascript">document.write("Dynamic HTML!");</script>
</head>
<body><!-- nothing here; will be added by the script later --></body>
</html>
```



©: Michael Kohlhase

184



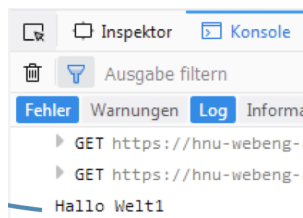
The example above already shows a JavaScript command: `document.write`, which replaces the content of the `<body>` element with its argument – this is only useful for testing and debugging purposes.

Here are three browser-level functions that can be used for user interaction (and finer debugging as they do not change the DOM).

#### Browser-level JavaScript functions

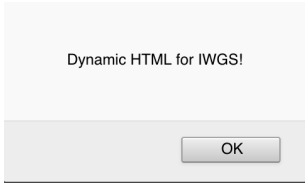
- ▷ **Example 6.4.4 (Logging to the browser console)**

```
console.log("hello IWGS")
```



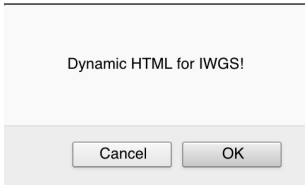
- ▷ **Example 6.4.5 (Raising a Popup)**



```
alert("Dynamic HTML for IWGS!")
```



▷ **Example 6.4.6 (Asking for Confirmation)**

```
var returnvalue = confirm("Dynamic HTML for IWGS!")
```




©: Michael Kohlhase
185


JavaScript is a client-side programming language, that means that the programs are delivered to the browser with the HTML documents and is executed in the browser. There are essentially three ways of embedding JavaScript into HTML documents:

### Embedding JavaScript into HTML

▷ In a `<script>` element in HTML, e.g.

```
<script type="text/javascript">
  function sayHello() { console.log('Hello IWGS!'); }
</script>
```



▷ External JavaScript file via a `<script>` element with `src`

```
<script type="text/javascript" src="../js/foo.js"/>
```

Advantage: HTML and JavaScript code are clearly separated

▷ In event-handler attributes of various HTML elements, e.g.

```
<input type="button" value="Hallo" onclick="alert('Hello IWGS')"/>
```


©: Michael Kohlhase
186


A related – and equally important – question is when the various embedded JavaScript fragments are executed. Here, the situation is more varied

### Execution of JavaScript Code

▷ **Question:** When and how is JavaScript Code Executed?

▷ **Answer:** While loading the HTML page or afterwards – triggered by events

▷ JavaScript in a script element: during page load (not in a function)

```
<script type="text/javascript">alert('Huhu');</script>
```

JavaScript in an **event-handler attribute** onclick, ondblclick, onmouseover, ...” whenever the corresponding **event** occurs.

▷ JavaScript in a “special link”: when the anchor is clicked

```
<a href="javascript:..." />
```



©: Michael Kohlhase

187



The first key concept we need to understand here is that the browser essentially acts as an user interface: it presents the HTML pages to the user, waits for actions by the user – usually mouse clicks, drags, or gestures; we call them **events**– and reacts to them.

The second is that all events can be associated to an element node in the DOM: consider an HTML anchor node, as we have seen above, this corresponds to a rectangular area in the browser window. Conversely, for any point  $p$  in the browser window, there is a minimal DOM element  $e(p)$  that contains  $p$  – recall that the DOM is a tree. So, if the user clicks while the mouse is at point  $p$ , then the browser triggers a click event in  $e(p)$ , determines how  $e(p)$  handles a click event, and if  $e(p)$  does not, bubbles the click event up to the parent of  $e(p)$  in the DOM tree.

There are multiple ways a DOM element can handel an event: some elements have default event handlers, e.g. an HTML anchor `<a href="⟨URI⟩">` will handle a click event by issuing a **HTTP** GET request for  $\langle\text{URI}\rangle$ . Other HTML elements can carry **event-handler attributes** whose JavaScript content is executed when the corresponding event is triggered on this element.



Actually there are more events than one might think at first, they include:

1. Mouse events; click when the mouse clicks on an element (touchscreen devices generate it on a tap); contextmenu: when the mouse right-clicks on an element; mouseover / mouseout: when the mouse cursor comes over / leaves an element; mousedown / mouseup: when the mouse button is pressed / released over an element; mousemove: when the mouse is moved.
2. Form element events; submit: when the visitor submits a `<form>`; focus: when the visitor focuses on an element, e.g. on an `<input>`.
3. Keyboard events; keydown and keyup: when the visitor presses and then releases the button.
4. Document events; DOMContentLoaded:– when the HTML is loaded and processed, DOM is fully built, but external resources like pictures `<img>` and stylesheets may be not yet loaded. load: the browser loaded all resources (images, styles etc); beforeunload / unload: when the user is leaving the page.
5. resource loading events; onload: successful load, onerror: an error occurred.

Let us now use all we have learned in an example to fortify our intuition about using JavaScript to change the DOM.

### Example: Changing Web Pages Programmatically

▷ **Example 6.4.7 (Stupid but Fun)**

<pre> &lt;body&gt; &lt;h2&gt;A Pyramid&lt;/h2&gt; &lt;div id="pyramid"/&gt;  &lt;script type="test/javascript"&gt;   var char = "#";   var triangle = "";   var str = "";   for(var i=0;i&lt;=10;i++){     str = str + char;     triangle = triangle + str + "&lt;br/&gt;"   }   var elem = document.getElementById("pyramid");   elem.innerHTML=triangle; &lt;/script&gt; &lt;/body&gt; &lt;/html&gt; </pre>	<p><b>Eine Pyramide</b></p> <pre> # ## ### #### ##### ##### ##### ##### ##### ##### </pre>		
	<p>©: Michael Kohlhasse</p>	<p>188</p>	

The HTML document in Example 6.4.7 contains an empty `<div>` element whose `id` attribute has the value `pyramid`. The subsequent `script` element contains some code that builds a DOM node-set of 10 text and `<br/>` nodes in the `triangle` variable. Then it assigns the DOM node for the `<div>` to the variable `elem` and deposits the `triangle` node-set as children into it via the JavaScript `innerHTML` method.

We see the result on the right of Example 6.4.7. It is the same as if the `#`-strings and `<br/>` sequence had been written in the HTML – which – at least for pyramids of greater depth – would have been quite tedious for the author.

## 6.4.2 JQuery: Write Less, Do More

While JavaScript is fully sufficient to manipulate the HTML DOM, it is quite verbose and tedious to write. To remedy this, the web developer community has developed libraries that extend the JavaScript language by new functionalities that more concise programs and are often used Instead of pure JavaScript.

### JQuery: Write Less, Do More

▷ **Definition 6.4.8** JQuery is a feature-rich JavaScript library that simplifies tasks like HTML document traversal and manipulation, event handling, animation, and Ajax.

▷ **Using:**

▷ Download from <https://jquery.com/download/>, save on your system (remember where)

▷ integrate into your HTML (usually in the `<head>`)

```
<script type="text/javascript" src="client-js/jquery-3.2.1.min.js"/>
```

or from the Internet directly (only works if you are online)

.2.2

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js" />
```



The key feature of JQuery is that it borrows the notion of “selectors” to describe HTML node-sets from CSS – actually, JQuery uses the CSS [selectors](#) directly – and then uses JavaScript-like methods to act on them. In fact, the name JQuery comes from the fact that selectors “query” for nodes in the DOM.

## JQuery Philosophy and Layers

### ▷ JQuery Philosophy:

```
$("#myId").show().css("color", "green").slideDown();
```

- ▷ find elements in the DOM by selectors, e.g. `$("#myId")`
- ▷ do something to them, e.g. `show()` (chaining of methods)
- ▷ change their layout by changing CSS attributes, e.g. `css("color","green")`
- ▷ change their behavior, e.g. `slideDown()`

### ▷ Good News: JQuery selectors $\hat{=}$ CSS selectors



We will now show a couple of JQuery methods for inserting material into HTML elements and discuss their behavior in examples

## Inserting Material into the DOM

### ▷ Inserting before the first child:

```
$('#content').prepend(function(){return 'in front';});
```

### ▷ Inserting after the last child:

```
$('#content').append('<p>Hello</p>');  
$('#content').append(function(){ return 'hinten'; });
```

### ▷ Inserting before/after an element:

```
$('#price').before('Preis:');  
$('#price').after(' EUR')
```



Let us fortify our intuition about dynamic HTML by going into a more involved example. We use the `toggle` method from the JQuery layout layer to change visibility of a DOM element. This method adds and removes a `style="display:none"` attribute to an HTML element and thus toggles the visibility in the browser window.

## Applications and useful tricks in Dynamic HTML

▷ **Example 6.4.9** hide document parts by setting CSS style attributes to `display:none`

```
<html>
<head>
  <title>Toggling</title>
  <style type="text/css">#dropper { display: none; }</style>
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js" />
  <script language="JavaScript" type="text/javascript">
    $("button").click(function(){$("#dropper").toggle();});
  </script>
</head>
<body>
  <h2>Toggling the visibility of material</h2>
  <button>...more </button>
  <div id="dropper"><p>Now you see it!</p></div>
</body>
</html>
```

**Application:** write “gmail” or “google docs” as JavaScript enhanced web applications.  
(client-side computation for immediate reaction)

▷ **Current Megatrend:** Computation in the “cloud”, browsers (or “apps”) as user interfaces



Current web applications include simple office software (word processors, online spreadsheets, and presentation tools), but can also include more advanced applications such as project management, computer-aided design, video editing and point-of-sale. These are only possible if we carefully balance the effects of server-side and client-side computation. The former is needed for computational resources and data persistence (data can be stored on the server) and the latter to keep personal information near the user and react to local context (e.g. screen size).



# Chapter 7

## What did we learn in IWGS-1?

### Outline of IWGS 1:

- ▷ programming in python (main tool in IWGS)
  - ▷ systematics and culture of programming
  - ▷ program and control structures
  - ▷ basic data structures like numbers and strings, character encodings, unicode, and regular expressions
- ▷ digital documents and document processing
  - ▷ text files
  - ▷ markup systems, HTML, and CSS
  - ▷ XML: Documents are trees.
- ▷ Web technologies for interactive documents and web applications
  - ▷ Internet infrastructure: web browsers and servers
  - ▷ serverside computing: bottle routing and
  - ▷ clientside interaction: dynamic HTML, Javascript, HTML forms
- ▷ Web Application Project (fill in the blanks to obtain a working web app)



©: Michael Kohlhase

193



### Outline of IWGS-II:

- ▷ Project Management and Collaboration on Data, Documents, and Software
  - ▷ Revision Control Systems
  - ▷ Issue Trackers and Project Wikis
- ▷ Data bases
  - ▷ CRUD operations, DB querying, and python embedding

- ▷ XML and JSON for file-based data storage
- ▷ Image Processing
  - ▷ Basics
  - ▷ Image transformations, Image Understanding
- ▷ Legal Foundations of Information Systems
  - ▷ Copyright & Licensing
  - ▷ Data Protection (GDPR)
- ▷ Ontologies, Semantic Web, and WissKI
  - ▷ Ontologies (inference  $\leadsto$  get out more than you put in)
  - ▷ Semantic Web Technologies (standardize ontology formats and inference)
  - ▷ Using SWTech for cultural heritage  $\leadsto$  the WissKI System

## Part II

# IWGS-II: DH Project Tools



# Chapter 8

## Semester Change-Over

### 8.1 Administrativa

We will now go through the ground rules for the course. This is a kind of a social contract between the instructor and the students. Both have to keep their side of the deal to make learning as efficient and painless as possible.

#### Prerequisites

- ▷ IWGS-1 (If you did not hear it, read the notes)
- ▷ **General Prerequisites:** Motivation, interest, curiosity, hard work
  - ▷ we will teach you all you need to know (apart from IWGS-1)
- ▷ You can do this course if you want!



©: Michael Kohlhase

195



Now we come to a topic that is always interesting to the students: the grading scheme: The short story is that things are complicated. We have to strike a good balance between what is didactically useful and what is allowed by Bavarian law and the FAU rules.

#### Assessment, Grades

- ▷ **Grading Background/Theory:** only modules are graded (by the law)
  - ▷ module “DH-Einführung”  $\hat{=}$  courses IWGS1/2, DH-Einführung
  - ▷ DHE module grade  $\leadsto$  pass/fail determined by “portfolio”  $\hat{=}$  collection of contributions/assessments
- ▷ **Assessment Practice:** The IWGS assessments in the “portfolio” consist of
  - ▷ weekly homework assignments (practice IWGS concepts and tools)
  - ▷ 60 minutes exam directly after lectures end:  $\sim$  Feb.10. (to show you master them)

- ▷ **Retake Exam:** 60 min exam at the end of the semester (~ Sep 30.)
- ▷ **To help you succeed:** we offer you
  - ▷ **External motivation:** points for homeworks and a grade for exam (even though only pass/fail relevant in the end)
  - ▷ **Mid-semester mini-exam** (online, optional, corrected but ungraded), (so you can predict the exam style)
  - ▷ weekly online quizzes that help you prepare for the course (ungraded ~ check understanding/preparation)



©: Michael Kohlhase

196



Homework assignments, quizzes and end-semester exam may seem like a lot of work – and indeed they are – but you will need practice (getting your hands dirty) to master the concepts. We will go into the details next.

## IWGS Homework Assignments

- ▷ **Homeworks:** will be small individual problem/programming/system assignments (but take time to solve) group submission if and only if explicitly permitted
- ▷ **Admin:** To keep things running smoothly
  - ▷ Homeworks will be posted on StudOn (<https://studon.fau.de/studon/crs2287043.html>)
  - ▷ Homeworks are handed in electronically (plain text, program files, PDF)
  - ▷ go to the tutorials, discuss with your TA (they are there for you!)
- ▷ **Homework Discipline:**
  - ▷ start early! (many assignments need more than one evening's work)
  - ▷ Don't start by sitting at a blank screen
  - ▷ Humans will be trying to understand the text/code/math when grading it.



©: Michael Kohlhase

197



It is very well-established experience that without doing the homework assignments (or something similar) on your own, you will not master the concepts, you will not even be able to ask sensible questions, and take nothing home from the course. Just sitting in the course and nodding is not enough!

If you have questions please make sure you discuss them with the instructor, the teaching assistants, or your fellow students. There are three sensible venues for such discussions: online in the lecture, in the tutorials, which we discuss now, or in the course forum – see below. Finally, it is always a very good idea to form study groups with your friends.



## IWGS Tutorials

- ▷ Weekly tutorials and homework assignments (first one in week two)

Teaching Assistants: (Doctoral Students in CS)

- ▷ Jonas Betzendahl: [jonas.betzendahl@fau.de](mailto:jonas.betzendahl@fau.de)
- ▷ Philipp Kurth: [philipp.kurth@fau.de](mailto:philipp.kurth@fau.de)

They know what they are doing and really want to help you learn! (dedicated to DH)

- ▷ Goal 1: Reinforce what was taught in class (important pillar of the IWGS concept)
- ▷ Goal 2: Let you experiment with python (think of them as Programming Labs)
- ▷ Life-saving Advice: go to your tutorial, and prepare it by having looked at the slides and the homework assignments
- ▷ Inverted Classroom: the latest craze in didactics (works well if done right)  
in CS: Lecture + Homework assignments + Tutorials  $\hat{=}$  Inverted Classroom

SOME RIGHTS RESERVED      ©: Michael Kohlhasse      198      **FAU** FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Do use the opportunity to discuss the IWGS topics with others. After all, one of the non-trivial inter/transdisciplinary skills you want to learn in the course is how to talk about Computer Science topics – maybe even with real Computer Scientists. And that takes practice, practice, and practice.

But what if you are not in a lecture or tutorial and want to find out more about the IWGS topics?

## Textbook, Handouts and Information, Forums

- ▷ No Textbook: but lots of online tutorials on the web
- ▷ Course notes will be posted at <http://kwarc.info/teaching/IWGS> (see references)
  - ▷ I mostly prepare/adapt/correct them as we go along.
  - ▷ please e-mail me any errors/shortcomings you notice. (improve for the group)
- ▷ Announcements will be posted on the StudOn course forum: [https://www.studon.fau.de/studon/goto.php?target=frm\\_2319978](https://www.studon.fau.de/studon/goto.php?target=frm_2319978)
- ▷ Check the forum frequently for
  - ▷ announcements, homework questions, ...
  - ▷ discussion among your fellow students
- ▷ If you become an active discussion group, the forum turns into a valuable resource!

SOME RIGHTS RESERVED      ©: Michael Kohlhasse      199      **FAU** FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

### Outline of IWGS-II:

- ▷ Project Management and Collaboration on Data, Documents, and Software
  - ▷ Revision Control Systems
  - ▷ Issue Trackers and Project Wikis
- ▷ Data bases
  - ▷ CRUD operations, DB querying, and python embedding
  - ▷ XML and JSON for file-based data storage
- ▷ Image Processing
  - ▷ Basics
  - ▷ Image transformations, Image Understanding
- ▷ Legal Foundations of Information Systems
  - ▷ Copyright & Licensing
  - ▷ Data Protection (GDPR)
- ▷ Ontologies, Semantic Web, and WissKI
  - ▷ Ontologies (inference  $\leadsto$  get out more than you put in)
  - ▷ Semantic Web Technologies (standardize ontology formats and inference)
  - ▷ Using SWTech for cultural heritage  $\leadsto$  the WissKI System



In IWGS-II, we want to consolidate the methods and technologies we learn in a small information system, which students build in groups, and which will serve as a running example for the course. These projects will consist of documents, data, and programs.

### IWGS-II Project

- ▷ **Idea:** Consolidate the techniques from IWGS-I and IWGS-II into a prototypical information system for Art History @ FAU. (Practical Digital Humanities)
- ▷ **A Running Example:** Research image + metadata collection “Bauernkirmes” provided by Prof. Peter Bell



- ▷ **What will you do?**: Build a web-based image/data manager, test image algorithms, annotate ontologically, . . .
- ▷ **How will we organize this**: Mostly via the group homework assignments (together they will make the project)



©: Michael Kohlhasse

201



Some IWGS students were worried that they will not be able to participate fully in the project, since they are not at the university often. A lot of the project collaboration will go via a collaboration and project management system – cf. Chapter 10.



## Chapter 9

# Databases

We now come to one of the core tools of computer science: databases give us a means to store large collections of data and organize them for efficient access. We will introduce the underlying concepts by example, go over the basics of relational database systems and the SQL language, and experiment with a concrete system: SQLite and its embedding into python.

**Acknowledgements:** We have borrowed and adapted examples and from [SSU04] and [PMDA] in this Chapter.

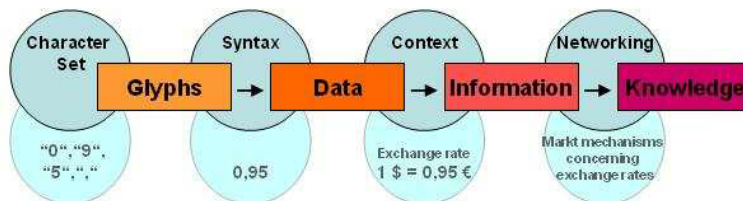
### 9.1 Introduction

Before we do anything else, we will look at various concepts around **data** to clarify concerns.

#### Databases, Data, Information, and Knowledge

▷ **Definition 9.1.1** Discrete, objective facts or observations, which are unorganized and uninterpreted are called **data** (singular **datum**).

▷ According to Probst/Raub/Romhardt [PRR97]



▷ **Example 9.1.2** The height of Mt. Everest (8.848 meters) is a **datum**.

▷ **Definition 9.1.3** A **database** is an organized collection of **data**, stored and accessed electronically from a computer system.



©: Michael Kohlhase

202

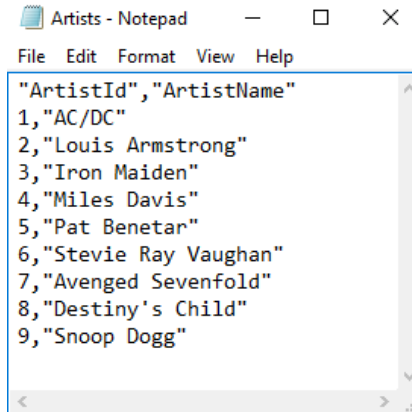
FAU  
FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG

To get an intuition about the possibilities of storing **data**, we look at some common ways – some of which we have already seen – and characterize them by some practical dimensions.

## Storing Data Electronically

▷ Four conventional ways of storing data: (mileage varies)

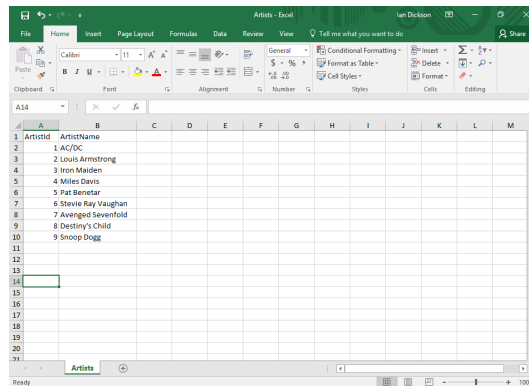
- ▷ In the **computer's memory** (RAM) (very fast (+), random access (-), but not persistent (-))
- ▷ In a **text file** (persistent (+), fast (+), sequential access (-), unstructured (-))



```

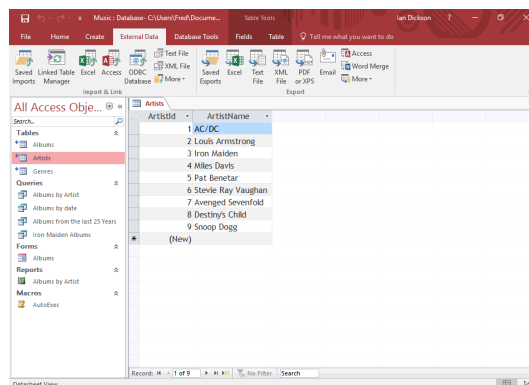
"ArtistId","ArtistName"
1,"AC/DC"
2,"Louis Armstrong"
3,"Iron Maiden"
4,"Miles Davis"
5,"Pat Benetar"
6,"Stevie Ray Vaughan"
7,"Averged Sevenfold"
8,"Destiny's Child"
9,"Snoop Dogg"
  
```

▷ In a **spreadsheet** (persistent (+), 2D-structured (+-), relations (+), slow (-))



ArtistId	ArtistName
1	AC/DC
2	Louis Armstrong
3	Iron Maiden
4	Miles Davis
5	Pat Benetar
6	Stevie Ray Vaughan
7	Averged Sevenfold
8	Destiny's Child
9	Snoop Dogg

▷ In a **database** (persistent (+), scalable (+), relations(+), managed (+), slow (-))



ArtistId	ArtistName
1	AC/DC
2	Louis Armstrong
3	Iron Maiden
4	Miles Davis
5	Pat Benetar
6	Stevie Ray Vaughan
7	Averged Sevenfold
8	Destiny's Child
9	Snoop Dogg
(New)	

- ▷ **Databases** constitute the most scalable, persistent solution.



We will study the practical aspects of one particularly important class of **database** systems: **relational database management systems**.

## 9.2 Relational Databases

We will now study a particular kind of database: **relational databases**, as these are the most widely used and structured ones.<sup>2</sup>

EdN:2

### (Relational) Database Management Systems

- ▷ **Definition 9.2.1** A **database management system (DBMS)** is program that interacts with end users, applications, and a database to capture and analyze the data and provides facilities to administer the database.
- ▷ There are different types of **DBMS**, we will concentrate on **relational** ones.
- ▷ **Definition 9.2.2** In a **relational database management system (RDBMS)**, data are represented as **tables**: every **datum** is represented by a **row** (also called **database record**), which has a **value** for all **columns** (also called an **attributes**) or **field**s. A **null value** is a special “**value**” used to denote a missing **value**.
- ▷ **Remark:** Mathematically, each **row** is an  **$n$ -tuple** of values, and thus a **table** an  **$n$ -ary relation**.  
(useful for standardizing **RDBMS** operations)
- ▷ **Example 9.2.3 (Bibliographic Data)**

Last	First	YOB	YOD	Title	YOP	Publisher	City
Twain	Mark	1835	1910	Huckleberry Finn	1986	Penguin USA	NY
Twain	Mark	1835	1910	Tom Sawyer	1987	Viking	NY
Cather	Willa	1873	1947	My Antonia	1995	Library of America	NY
Hemingway	Ernest	1899	1961	The Sun Also Rises	1995	Scribner	NY
Wolfe	Thomas	1900	1938	Look Homeward, Angel	1995	Scribner	NY
Faulkner	William	1897	1962	The Sound and the Fury	1990	Random House	NY

- ▷ **Definition 9.2.4** **Tables** are identified by **table name** and individual components of **records** by **column name**.



As **RDBMS** constitute the backbone of modern information technology, there are many many implementations, commercial ones and open source ones as well. For our purposes, open-source systems are completely sufficient, so we list the most important ones here.

<sup>2</sup>EdNOTE: MK: In the last years, NoSQL databases and JSON have gained prominence. Intro them at the end and reference them here.

## Open-Source Relational Database Management Systems

- ▷ **Definition 9.2.5** MySQL is an open source **RDBMS**. For simple data sets and Web application MySQL is a fast and stable multi-user system featuring an **SQL** database server that can be accessed by multiple clients.



- ▷ **Definition 9.2.6** PostgreSQL is an open source **RDBMS** with an emphasis on extensibility, standards compliance, and scalability.



- ▷ **Definition 9.2.7** SQLite is an embeddable **RDBMS**. Instead of a database server, SQLite uses a single database file, therefore no server configuration is necessary.



- ▷ **Remark 9.2.8** At the level we use **SQL** in IWGS, all are equivalent.
- ▷ We will use SQLite in IWGS, since it is easiest to install and configure.



Now that we have made our first steps in the **SQL** language and with **RDBMS** in general, let us pick a concrete **RDBMS** to experiment with.

## Working with SQLite (via the SQLite shell)

- ▷ In IWGS we will use SQLite, since it is very lightweight, easy to install, but feature-complete, and widely used.
- ▷ Download SQLite at <https://www.sqlite.org/download.html>,
- ▷ e.g. `sqlite-dll-win64-x64-3280000.zip` for windows.
- ▷ unzip it into a suitable location, start `sqlite3.exe` there
- ▷ this opens a **command line interpreter**: the **SQLite shell**. (all DBs have one)  
test it with `.help` that tells you about more “dot **commands**”.

```
> sqlite3
SQLite version 3.24.0 2018-06-04 19:24:41
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .help
.archive ... Manage SQL archives: ".archive --help" for details
.auth ON|OFF Show authorizer callbacks
[...]
```

- ▷ If you have a database file `books.db` from Example 9.3.8, use that.

```
> sqlite3 books.db
SQLite version 3.24.0 2018-06-04 19:24:41
Enter ".help" for usage hints.
> .tables
```

Books

```
>select * from Books;
```

```
Twain|Mark|1835|1910|Huckleberry Finn|1986|Penguin USA|NY
Twain|Mark|1835|1910|Tom Sawyer|1987|Viking|NY
Cather|Willa|1873|1947|My Antonia|1995|Library of America|NY
Hemingway|Ernest|1899|1961|The Sun Also Rises|1995|Scribner|NY
Wolfe|Thomas|1900|1938|Look Homeward, Angel|1995|Scribner|NY
Faulkner|William|1897|1962|The Sound and the Fury|1990|Random House|NY
Tolkien|John Ronald Reuel|1892|1973|The Hobbit|1937|George Allen & Unwin|UK
```

.tables shows the available tables

select \* from Books is **SQL** (see below); it shows all entries of the Books table.



©: Michael Kohlhasse

206



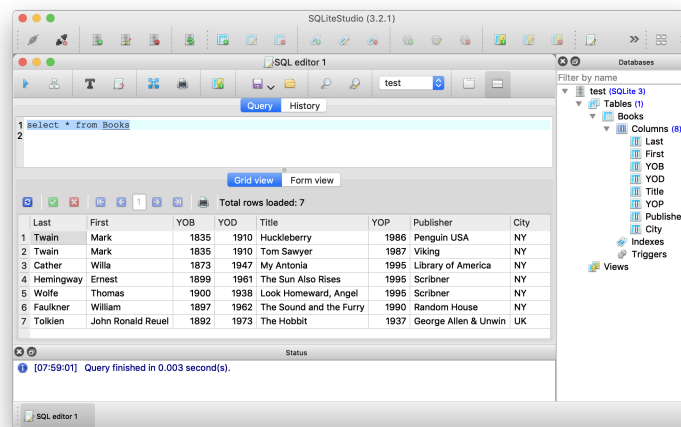
Interacting with SQLite via the **database shell** is nice, but can be quite tedious. Fortunately, there are better alternatives.

## A Graphical User Interface for SQLite

▷ **Definition 9.2.9** A **database browser** is a graphical user interface for a **RDBMS** that (typically) bundles an **SQL instruction** editor with displays for results and the **database schema**.

▷ I will sometimes use one for SQLite in the slides: SQLite Studio (lots of others)

▷ download at <https://sqlitestudio.pl/index.rvt?act=download>



▷ Everything we can do with this, we can do with the database shell as well. (just looks nicer)



©: Michael Kohlhasse

207



## 9.3 SQL – A Standardized Interface to RDBMS

**Idea:** To interact with in **RDBMSs**, we need a language to describe **tables** to the system, so that they can be created, read, updated, and deleted. In fact while we are at it, we need a language

for all **RDBMS** operations. The domain-specific language **SQL** (pronounced like “sequel”) fills this need. It is internationally standardized, so that it can be used as the lingua franca for all **RDBMSs**, insulating users and application programmers against system internals.

## SQL: the Structured Query Language

- ▷ **Idea:** We need a language for describing all operations of a **RDBMSs**.
  - ▷ **basics:** creating, reading, updating, deleting database components (**CRUD**)
  - ▷ **querying:** selecting from and inserting into the database
  - ▷ **access control:** who can do what in a database
  - ▷ **transactions:** ensuring a consistent database state.
- ▷ **Definition 9.3.1** **SQL**, the **structured query language** is a domain-specific language for managing data held in a **RDBMS**. **SQL instructions** are directly executed by the **RDBMS** to change the database state or compute answers to **SQL** queries.



©: Michael Kohlhasse

208



We start off with a fragment of **SQL** that is concerned with setting up the **database schema**, which gives structure to the data in the database. This schema is used by the **RDBMS** to optimize database accesses.

## DDL: Data Definition Language

- ▷ **Definition 9.3.2** The **data definition language** (**DDL**) is a subset of **SQL** instructions that address the creation and deletion of database objects.
- ▷ **Definition 9.3.3** The **SQL** statement **CREATE TABLE**⟨name⟩ (⟨coldefs⟩) creates a **table** with name ⟨name⟩. ⟨coldefs⟩ are **column specifications** that specify the **columns**: it is a comma-separated list of **column names** and **SQL data types**. The totality of all **column specifications** of all **tables** in a **database** is called the **database schema**.
- ▷ **Example 9.3.4 (Creating a Table)** The following **SQL** statement creates the table from Example 9.2.3
 

```
CREATE TABLE Books (
  Last varchar(128), First varchar(128),
  YOB int, YOD int, Title varchar(255), YOP int,
  Publisher varchar(128), City varchar(128)
);
```
- ▷ other **CREATE** statements exist, e.g. **CREATE DATABASE** ⟨name⟩.
- ▷ **Definition 9.3.5** The **SQL** statement **DROP** ⟨obj⟩ ⟨name⟩ deletes the database object of class ⟨obj⟩ with name ⟨name⟩.



©: Michael Kohlhasse

209



We have seen above that the **database schema** needs a **data type** for every **column**. We give an overview over the most important ones here.

## SQL Data Types (for Column Specifications)

- ▷ **Definition 9.3.6** SQL specifies **data types** for **values** including:
  - ▷ **VARCHAR** ( $\langle\langle\text{length}\rangle\rangle$ ): character strings, including Unicode, of a variable length is up to the maximum length of  $\langle\langle\text{length}\rangle\rangle$ .
  - ▷ **BOOL** truth values: **true**, **false** and case variants.
  - ▷ **INT**: Integers
  - ▷ **FLOAT**: floating point numbers
  - ▷ **DATE**: dates, e.g. **DATE** '1999–01–01' or **DATE** '2000–2–2'
  - ▷ **TIME**: time points in ISO format, e.g. **TIME** '00:00:00' or **time** '23:59:59.99'
  - ▷ **TIMESTAMP**: a combination of **DATE** and **TIME** (separated by a blank).
  - ▷ **CLOB** ( $\langle\langle\text{length}\rangle\rangle$ ) (character large object) up to (typically) 2 Gi B
  - ▷ **BLOB** ( $\langle\langle\text{length}\rangle\rangle$ ) (binary large object) up to (typically) 2 Gi B



We now come to the **SQL** commands for inserting content into the database tables we have created above. This is quite straight-forward.

## SQL: Adding Records to Tables

- ▷ **Definition 9.3.7** SQL provides the **INSERT INTO** command for **inserting** records into a **table**. This comes in two forms:
  1. **INSERT INTO**  $\langle\langle\text{table}\rangle\rangle$  **VALUES** ( $\langle\langle\text{vals}\rangle\rangle$ ); where  $\langle\langle\text{vals}\rangle\rangle$  is a comma-separated list of values given in the order the columns were declared in the **CREATE TABLE** instruction.
  2. **INSERT INTO**  $\langle\langle\text{table}\rangle\rangle$  ( $\langle\langle\text{cols}\rangle\rangle$ ) **VALUES** ( $\langle\langle\text{vals}\rangle\rangle$ ) where  $\langle\langle\text{vals}\rangle\rangle$  is a comma-separated list of values given in the order of  $\langle\langle\text{cols}\rangle\rangle$  (a subset of columns) all other fields are filled with **NULL**
- ▷ **Example 9.3.8 (Inserting into the Books Table)** The given the table Books from Example 9.3.4 we can add a record with
 

```
INSERT INTO Books
VALUES ('Tolkien', 'John Ronald Reuel', 1892, 1973, 'The Hobbit', 1937,
      'George Allen & Unwin', 'UK');
```
- ▷ **Example 9.3.9 (Inserting Partial Data)** Using the second form of the **INSERT** instruction, we can insert partial data. (all we have)

```
INSERT INTO Books (First, Last, YOB, title, YOP)
VALUES ('Michael', 'Kohlhase', '1964', 'IWGS Course Notes', '2018');
```



With an insert facility, we need to be able to delete records as well, again it is straight-forward, with the exception that we have to identify which records to delete.

## SQL: Deleting Records from Tables

- ▷ **Definition 9.3.10** The **SQL delete** statement allows to change existing records.

```
DELETE FROM «table» WHERE «condition»;
```

- ▷ **Example 9.3.11** Deleting the record for “Huckleberry Finn”.

```
DELETE FROM Works WHERE Title = 'Huckleberry Finn'
```

⚠: If we leave out the **WHERE** clause, all **rows** are deleted.

- ▷ **Note:** There is much more to the **WHERE** clause, we will get to that when we come to **SQL** querying (see [Section 9.7](#))



And now we come to a variant of database insertion: record update. In principle, this could be achieved by deleting the record and then re-inserting the changed one, but the update instruction presented here is more efficient.

## SQL: Updating Records in Tables

- ▷ **Definition 9.3.12** The **SQL update** statement allows to change existing records.

```
UPDATE «table»  
SET «column»1 = «value»1, «column»2 = «value»2, ...  
WHERE «condition»;
```

- ▷ **Example 9.3.13** Updating the publisher in “Huckleberry Finn”.

```
UPDATE Books  
SET Publisher = 'Chatto & Windus', YOP = 1884, City = 'London'  
WHERE Title = 'Huckleberry Finn'
```

⚠ **Again:** If we leave out the **WHERE** clause, all **rows** are updated.



## 9.4 ER-Diagrams and Complex Database Schemata

We now come to a very important aspect of structured databases: designing the **database schema** – and with this determining the data efficiency and computational performance of the database itself. We get glimpse of the standard tool: **entity relationship diagrams** here.

### ▷ Avoiding Redundancy in Databases

- ▷ Recall the books table from Example 9.2.3:

Last	First	YOB	YOD	Title	YOP	Publisher	City
Twain	Mark	1835	1910	Huckleberry Finn	1986	Penguin USA	NY
Twain	Mark	1835	1910	Tom Sawyer	1987	Viking	NY
Cather	Willa	1873	1947	My Antonia	1995	Library of America	NY
Hemingway	Ernest	1899	1961	The Sun Also Rises	1995	Scribner	NY
Wolfe	Thomas	1900	1938	Look Homeward, Angel	1995	Scribner	NY
Faulkner	William	1897	1962	The Sound and the Fury	1990	Random House	NY

**Observation:** Some of the fields appear multiple times, e.g. “Mark Twain”.

⇒ **⚠:** When the database grows this leads to scalability problems

- ▷ in **querying**: e.g. if we look for all works by Mark Twain
- ▷ in **maintenance**: e.g. if we want to replace the pen name “Mark Twain” by the real name “Samuel Langhorne Clemens”.
- ▷ **Idea**: Separate concerns (here Authors, Works, and Publishers) into separate entities, mark their relations.
  - ▷ Develop a graphical notation for planning
  - ▷ Implement that into the database



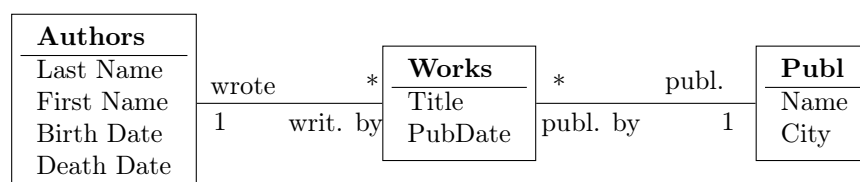
After this discussion on why we need to design an efficient **database schema** to the **entity relationship diagrams** themselves.

## Entity Relationship Diagrams

- ▷ **Definition 9.4.1** An **entity relationship diagram (ERD)** illustrates the logical structure of databases. It consists of **entities** that characterize (sets of) objects by their **attributes** and **relations** between them.
- ▷ **Example 9.4.2 (An ERD for Books)** Recall the Books table from Example 9.2.3:

Last	First	YOB	YOD	Title	YOP	Publisher	City
Twain	Mark	1835	1910	Huckleberry Finn	1986	Penguin USA	NY
Twain	Mark	1835	1910	Tom Sawyer	1987	Viking	NY
Cather	Willa	1873	1947	My Antonia	1995	Library of America	NY
Hemingway	Ernest	1899	1961	The Sun Also Rises	1995	Scribner	NY
Wolfe	Thomas	1900	1938	Look Homeward, Angel	1995	Scribner	NY
Faulkner	William	1897	1962	The Sound and the Fury	1990	Random House	NY

- ▷ **Problem**: We have duplicate information in the authors and publishers
- ▷ **Idea**: Spread the Books information over multiple tables.



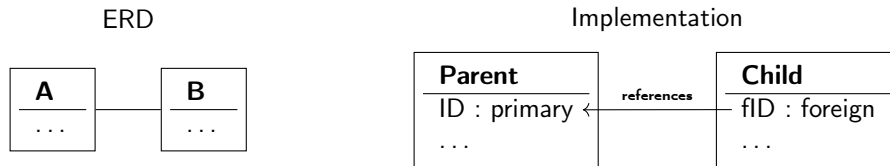
Generally, a good database design is almost always worth the effort, since it makes the code and maintenance of the applications based on this database much simpler and intuitive.

We are fully aware, that this little example completely under-sells [entity relationship diagrams](#) and does not do this important topic justice. Fortunately, the DH students at FAU have the mandatory course “Konzeptuelle Modellierung” which does.

We now come to the implementation of the ideas from the [entity relationship diagrams](#). The key idea is to have references between tables. These are mediated by special database [columns](#) types, which we now introduce.

### Linking Tables via Primary and Foreign Keys

- ▷ **Definition 9.4.3** A [column](#) in a [table](#) can be designated as a [primary key](#). This constrains its values to be non-[null](#) and [unique](#) i.e. all distinct.  
In [DDL](#), we just add the keyword **PRIMARY KEY** to the [column specification](#).
- ▷ **Definition 9.4.4** A [foreign key](#) is a [column](#) (or collection of [columns](#)) in one [table](#) (called the [child table](#)) that refers to the [primary key](#) in another [table](#) (called the [reference table](#) or [parent table](#)).
- ▷ **Intuition:** Together [primary keys](#) and [foreign keys](#) can be used to link tables or (dually) to spread information over multiple tables.



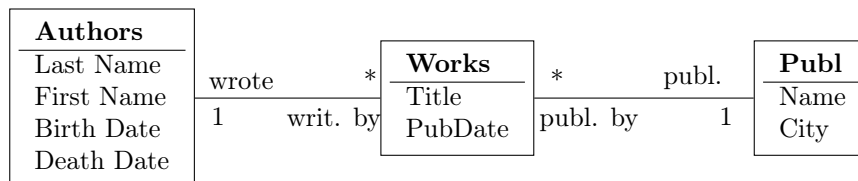
- ▷ **BTW:** [Primary keys](#) are great for for identification in the **WHERE** clauses of [SQL instructions](#).



We now fortify our intuition on [primary](#) and [foreign keys](#) by taking up Example 9.4.2 again.

### Linking Tables via Primary and Foreign Keys (Example)

- ▷ **Example 9.4.5** Continuing Example 9.4.2, we now implement



by introducing [primary keys](#) in the Authors and Publishers tables and referencing them by [foreign keys](#) in the Works table.

```
CREATE TABLE Authors (AuthorID INTEGER PRIMARY KEY,
  Last varchar(128), First varchar(128), YOB int, YOD int);
```

```

CREATE TABLE Publishers (PublisherID INTEGER PRIMARY KEY,
    Name varchar(128), City varchar(128));

CREATE TABLE Works (
    Title varchar(255), YOP int, AuthorID int, PublisherID int,
    FOREIGN KEY(AuthorID) REFERENCES Authors(AuthorID),
    FOREIGN KEY(PublisherID) REFERENCES Publishers(PublisherID));

```



©: Michael Kohlhasse

217



## Linking Tables via Primary and Foreign Keys (continued)

- ▷ **Example 9.4.6 (Inserting into the Works Table)** The given the tables Works, Authors, and Publishers from Example 9.4.5 we can add a record with

```

INSERT INTO Authors VALUES (1, 'Twain', 'Mark', 1835, 1910);
INSERT INTO Publishers VALUES (1, 'Penguin USA', 'NY');
INSERT INTO Works VALUES ('Huckleberry Finn', 1986, 1, 1);

INSERT INTO Publishers VALUES (2, 'Viking', 'NY');
INSERT INTO Works VALUES ('Tom Sawyer', 1987, 1, 2);

```



©: Michael Kohlhasse

218



**Note:** We have introduced new integer-typed **columns** for the **primary key** in the Authors and Publishers tables. In principle, we could have designated any existing **column** as a **primary key** instead, if we were sure that the entries are unique – in our case an unreasonable assumption, even for the publishers.

We have also chosen not to introduce a **primary key** in the Works table, which is probably a design mistake in the long run, because this would be very important to have for **deletions** and **updates**.

## 9.5 RDBMS in Python

Let us now see how we can interact with SQLite programmatically from python instead of from the SQLite shell or the database browser.

### Using SQLite from python

- ▷ We will use the PySQLite package
- ▷ install it locally with `pip install pysqlite for python3`.
  - ▷ use **import** `sqlite3` to import the library in your programs.
- ▷ Typical python program with `sqlite3`:

```

import sqlite3
# Open database connection
db = sqlite3.connect(⟨⟨host⟩⟩,⟨⟨user⟩⟩,⟨⟨pass⟩⟩,⟨⟨DBname⟩⟩)

```

```
# prepare a cursor object using cursor() method
cursor = db.cursor()
# execute SQL commands using the execute() method.
cursor.execute("«SQL»")
«data processing code»
# make sure data reaches disk
db.commit()
# disconnect from server
db.close()
```

We will assume this as a wrapper for all code examples below.



The script schema shows the normal way of setting up the interaction with a database using `sqlite3`:

1. We first connect to the database by specifying the database file in which the data is kept. Normally, this will be file on the local file system, but we can also use a file that is available on a remote host «host». Of course, to write to this file will normally require [authentication](#), therefore `sqlite3.connect` also takes a user name «user» and a password «pass» as additional arguments. An alternative for the «DBName» argument is the string `:memory:` which results in an in-memory database (no persistent storage). The result of the `sqlite3.connect` function is a database [object](#) `db`.
2. Then we create a [cursor](#) object `cursor` (cf. slide 231 for more details) by using the [cursor method](#) of the database [object](#) `db`.
3. Then we execute [SQL instructions](#) via `cursor.execute` and do the data processing we need for our application.
4. To make sure that the changes we made to the database are actually reflected on disk in the database file «DBName», we commit the changes to disk via `db.commit()`.
5. Finally, we close the database connection via the `db.close` [method](#) to make sure that all our changes have reached the database file.

We will now put this schema to use using Example 9.3.8 as a basis.

## Creating Tables in python

### ▷ Example 9.5.1 Creating the table of Example 9.3.4

```
import sqlite3
# our database file
database = "C:\\sqlite\\db\\books.db"
# a string with the SQL instruction to create a table
create = """CREATE TABLE Books (
    Last varchar(128), First varchar(128), YOB int, YOD int,
    Title varchar(255), YOP int, Publisher varchar(128), City varchar(128));"""
insert1 = """INSERT INTO Books
    VALUES ('Twain', 'Mark', '1835', '1910', 'Huckleberry Finn', '1986',
    'Penguin USA', 'NY');"""
insert2 = """INSERT INTO Books
    VALUES ('Twain', 'Mark', '1835', '1910', 'Tom Sawyer', '1987',
    'Viking', 'NY');"""
# connect to the SQLite DB and make a cursor
db = sqlite3.connect(database)
cursor = db.cursor()
# create Books table by executing the cursor
cursor.execute("DROP TABLE Books;")
```

```

cursor.execute(create)
cursor.execute(insert1)
cursor.execute(insert2)
db.commit() # commit to disk
db.close() # clean up by closing

```



In this example we first create an **SQL instruction** as a string, so that we can give them as arguments to the `cursor.execute` method conveniently.

Note that `cursor.execute` only executes a single **SQL instructions** (for safety reasons; see slide 222 – why does this help there?).

Note that we drop the **Books** table before (re)creating it, to be sure that we have the right structure and avoiding errors, when we run the **python** script above twice. An alternative would have been to use **CREATE TABLE IF NOT EXISTS**, which only creates the table if there is none. But in our example here, where we directly fill the table, dropping any old tables with the name **Books** seems the right thing to do.

There is an issue that sometimes baffles beginners: I have created a table, inserted lots of data into it, closed the database, and the next time I connect to the database, it is empty ~ very annoying.

To understand this phenomenon, we have to understand a bit more how databases like **SQLite** work and the tradeoffs face when working with such systems.

### To commit or not to commit?

- ▷ **Recall:** SQLite computes with tables in **memory** but uses **files** for persistence.
- ▷ **Also Recall:** **Memory** access is 100-10.000 times as fast as **file** access.
- ▷ **Idea 1:** Keep tables in **memory**, write to **file** only when necessary.
- ▷ **Idea 2:** Give the user/programmer control over when to write to **file**
  - ▷ `db = sqlite3.connect(⟨file⟩)` connects to ⟨file⟩, but computes in **memory**,
  - ▷ `db.commit()` writes in-memory changes to ⟨file⟩.
- ▷ **Problem:** We can have multiple database connections to the same database file in parallel, there may be race conditions and conflicts.
- ▷ **Our Solution:** Commit often enough! (your responsibility/fault)
- ▷ **General Solution:** **RDBMS** offer **database transactions**. (not covered in IWGS)
- ▷ **Lazy Solution:** Set the connection to **autocommit mode**: (system decides)  
`sqlite3.connect(⟨file⟩, isolation_level = None)`



**Excursion:** The general solution to the problem of accessing a database from multiple programs or processes in parallel is solved by a complex technology called **database transactions**, which allow users' to define a sensible unit of work (via **begin/end** bracketing) called a **transaction** and makes sure that the process

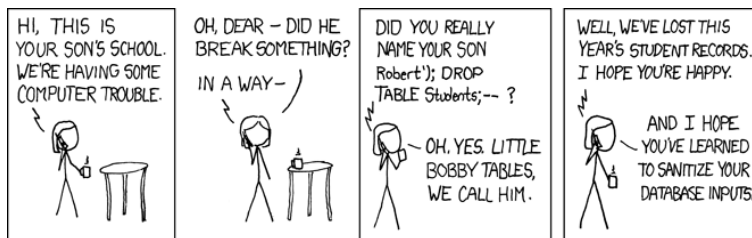
- behaves as if the user's process has sole access to the database system for the duration of the **transaction** (**isolation**)
- any changes made during the **transaction** can be rolled back if an error occurs during processing (**integrity**).

**Transactions** are an essential, but complex technology that is beyond the scope of the IWGS course. For our understanding, `db.commit` is essentially just the end bracket of a **transaction**.

Now that we understand how to deal with databases programmatically, we can come to a real-world menace: **SQL injection attacks**. A large portion of the “hacking” events, where a database is taken over by malicious agents are based – at least in part – on such a technique. Therefore it is important to understand the basic principles involved, if only to understand how we can safeguard against them – see e.g. slide 233 below.

### Beware of the HTML/python/SQLite Interaction

▷ **What have we learned?:** At least you now understand the following web comic:  
(<https://xkcd.com/327/>)



▷ **Definition 9.5.2** We call this an **SQL injection attack**.

▷ **Hint:** Imagine a Web Application where you add student names for enrolment. This has a python **line**

```
name = input("Please enter student name: ")
cursor.execute(f"INSERT INTO Students VALUES (... ,{Name}, ...);")
```

which for the input `Robert'); DROP TABLE Students;` generates and executes the **SQL** instructions

```
INSERT INTO Students VALUES (... , 'Robert'); DROP TABLE Students;
```



Now we can understand why the restriction of `cursor.execute` to only one **SQL instruction** enhances security of the code: The hypothetical `cursor.execute('INSERT ...')` command expects one **instruction**, but with the parameter substitution in the f-string gets two. This would have raised an error and saved the school administration.

## 9.6 Excursion: Programming with Exceptions in Python

Before we go on, we discuss how we can deal with errors in python flexibly, so that our **web application** web applications will not drop into the python level and present the user with a stack trace.

We first introduce what errors really are in the python context and how they are **raised** and **handled**. Then we look at what this means for our handling of database connections.

### How to deal with Errors in python

▷ **Theorem 9.6.1 (Kohlhase's Law)** *I can be an idiot, and I do make mistakes!*

▷ **Definition 9.6.2** An **exception** is a special python **object**. **Raising** an exception  $e$  terminates computation and passes  $e$  to the next higher level.

▷ **Example 9.6.3 (Division by Zero)** The python interpreter reports unhandled **exceptions**

```
>>> -3 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

**Exceptions** are first-class citizens in python, in particular they

- ▷ are classified by their **classes** in a hierarchy.
- ▷ **exception classes** can be defined by the user (they inherit from the **Exception class**)

```
class DivByZero (Exception)
    pass
```

- ▷ can be **raised** when an abnormal condition appears

```
if denominator == 0 :
    raise DivByZero
else
    «computation»
```

- ▷ can be **handled** in a **try/except** block (there can be multiple)

```
try:
    «tentative computation»
except : «err»1, ..., «err»n :
    «errorhandling»
finally :
    «cleanup»
```



Let us now apply python **exception** to our situation. Here the most important source of errors is the database connection step, where a database file might be missing or a remote host with the database file offline.

### Playing it Safe with Databases

- ▷ **Observation 9.6.4** *Things can go wrong when connecting to a database (e.g. missing file)*
- ▷ **Idea:** **Raise exceptions** and **handle** them.

- ▷ **Example 9.6.5** we encapsulate a **try/except** block into a function for convenience

```
import sqlite3
from sqlite3 import Error
def sql_connection():
    try:
        db = sqlite3.connect(':memory:')
        print("Connection is established: Database is created in memory")
    except Error :
        print(Error)
    finally:
        db.close()
```

The `sqlite3` package provides its own [exceptions](#), which we import separately. Other errors can be [handled](#) in additional **except** clauses.



## 9.7 Querying and Views in SQL

So far we have created, filled, and possibly updated databases, but we have not done anything useful with them. That is the realm of [querying](#) in [SQL](#), which we will now come to.

We will first cover [SQL querying](#) from a single table. There are many variants of the **SELECT/-FROM/WHERE** instruction. We explain the most commonly used ones.

### SQL Querying: The SELECT Statement

- ▷ [SQL](#) uses the **SELECT instruction** for retrieving data from a [database](#).
- ▷ **SELECT** `⟨columns⟩` **FROM** `⟨table⟩` returns all records from `⟨table⟩` restricted to the [fields](#) from `⟨columns⟩`.
- ▷ **Definition 9.7.1** We call a **SELECT instruction** a [query](#).
- ▷ **Example 9.7.2** **SELECT** Title, YOP **FROM** Books;
 

Huckleberry Finn	1986
Tom Sawyer	1987
My Antonia	1995
The Sun Also Rises	1995
Look Homeward, Angel	1995
The Sound and the Fury	1990
The Hobbit	1937
- ▷ **SELECT DISTINCT** removes duplicate values
- ▷ **SELECT \* FROM** `⟨table⟩` returns all records from `⟨table⟩`.
- ▷ **SELECT** `⟨columns⟩` **FROM** `⟨table⟩` **WHERE** `⟨cond⟩` returns all records that match condition `⟨cond⟩`
- ▷ **Example 9.7.3** **SELECT** First, Last **FROM** Books **WHERE** YOP = 1995;

Willa Cather
Ernest Hemingway
Thomas Wolfe

▷ **SELECT** `⟨columns⟩` **FROM** `⟨table⟩` **ORDER BY** `⟨columns⟩` orders the results by `⟨columns⟩`

▷ **Example 9.7.4** Ordering can be ascending (**ASC**) or descending (**DESC**)  
**SELECT First, Last FROM Books ORDER Last ASC, YOP DESC;**



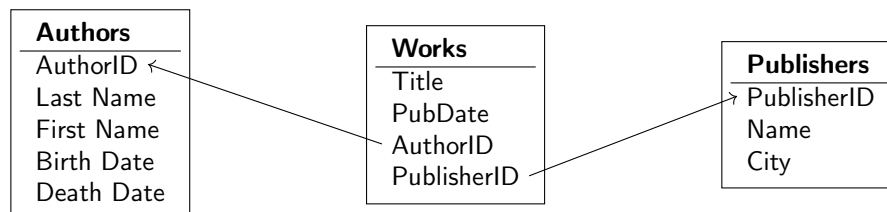
There are some more variants, for instance we can add a **GROUP BY** clause, which allows to group the result table according to various conditions.

We now generalize **SQL queries** by combining multiple tables into a virtual **table** from which we aggregate the results. Joins over that combine multiple tables in queries are the technique that allows to split data into multiple tables in the first place: we can re-recreate the “original big table” via a query.

We will restrict ourselves to the simplest kind of table join: the “inner join” below. There are quite a few variants of joins; we refer the reader to the literature on them.

## Joining Tables in Queries

▷ **Problem:** We can query single tables, how cross-table queries? E.g. in



▷ **Idea:** virtually joining tables for the query

▷ **Definition 9.7.5** A **table join** (or simply **join**) is a means for combining **columns** from one (**self-join**) or more tables by using **values** common to each.

▷ **Example 9.7.6** **Joining** all three tables from Example 9.4.2.

```

SELECT
  Authors.Last, Authors.First, Authors.YOB, Authors.YOD,
  Title, YOP, Publishers.Name, Publishers.City
FROM
  Works
  INNER JOIN Authors ON Authors.AuthorID = Works.AuthorID
  INNER JOIN Publishers ON Publishers.PublisherID = Works.PublisherID
  
```



The key idea in the **query** in Example 9.7.6 are the **join** statements in the last two lines. They do two things: first they tell **SQL** to extend the **Works** table with data from the two tables **Authors** and **Publishers**, and second they tell **SQL** how the extension should work: by making sure that in the extension the records in the **Works** table are extended with the (unique!) record in the **Authors**

table, that has the same `AuthorID`, and analogously for the records from the `Publishers` table. Thus the two joins implement the two arrows in the ER diagram at the top of the slide. The result of this query is displayed on the next slide.

## Joining Tables in Queries (Result)

### ▷ Example 9.7.7

SQLiteStudio (3.2.1)

SQL editor 1

```

1 SELECT
2   Authors.Last, Authors.First, Authors.YOB, Authors.YOD, Title, YOP,
3   Publishers.Name, Publishers.City
4 FROM Works
5 INNER JOIN Authors ON Authors.AuthorID = Works.AuthorID
6 INNER JOIN Publishers ON Publishers.PublisherID = Works.PublisherID

```

Grid view Form view

Total rows loaded: 8

	Last	First	YOB	YOD	Title	YOP	Name	City
1	Twain	Mark	1835	1910	Huckleberry Finn	1986	Penguin USA	NY
2	Twain	Mark	1835	1910	Tom Sawyer	1987	Viking	NY
3	Cather	Willa	1873	1947	My Antonia	1995	Library of America	NY
4	Hemingway	Ernest	1899	1961	The Sun Also Rises	1995	Scribner	NY
5	Wolfe	Thomas	1900	1938	Look Homeward, Angel	1995	Scribner	NY
6	Faulkner	William	1897	1962	The Sound and the Fury	1990	Random House	NY
7	Tolkien	John Ronald Reuel	1892	1973	The Hobbit	1937	George Allen & Unwin	UK



Note that the result of the query from Example 9.7.6 shown in Example 9.7.7 exactly recreates the original big `Books` table from Example 9.2.3. So we see that we have “lost nothing” by separating the data into three more efficient and less redundant – tables.

We have seen above that we can [join](#) physical database tables to larger virtual ones whenever we need it in a [SQL query](#). This is good, but it can be made even better. [RDBMS](#) allow to persist virtual [tables](#) in the [database schema](#) itself as [views](#).

## Database Views: Persisting Queries

▷ **Observation:** Via the [join](#) in Example 9.7.6, the `Works` table queries like the original `Books` table.

▷ **Wouldn't it be nice** If we could also insert/update into that?

▷ **Definition 9.7.8** A [database view](#) (or simply [view](#)) is a virtual [table](#) based on the result-set of a [query](#). A [view](#) contains [rows](#) and [columns](#), just like a real [table](#). The [fields](#) in a [view](#) are [fields](#) from one or more real [tables](#) in the database.

▷ **Remark 9.7.9** We can even [insert](#), [delete](#), and [update](#) records in a [view](#), just as in any other [table](#) of the [database](#).

The [RDBMS](#) achieves this by automatically translating any change to the [view](#) into a set of changes to the underlying physical tables.

▷ : but not in SQLite, (this is an omission due to simplicity)



**Remark:** With [views](#) we can “have our cake and eat it too”: We can make our [database schema](#) space-efficient by removing redundancies using “small tables” and still have our “big tables” that make our life convenient e.g. when [inserting](#) records. Consider our Books example again: we can give the query from Example 9.7.6 a name and let the [RDBMS](#) treat it as a (virtual) table.

## Database Views: Persisting Queries (Books Example)

▷ **Example 9.7.10** Use the query from Example 9.7.6 to define a view

```
CREATE VIEW Books AS
SELECT
  Authors.Last AS Last, Authors.First AS First,
  Authors.YOB AS YOB, Authors.YOD AS YOD,
  Title, YOP,
  Publishers.Name AS Publisher, Publishers.City AS City
FROM
  Works
  INNER JOIN Authors ON Authors.AuthorID = Works.AuthorID
  INNER JOIN Publishers ON Publishers.PublisherID = Works.PublisherID
```

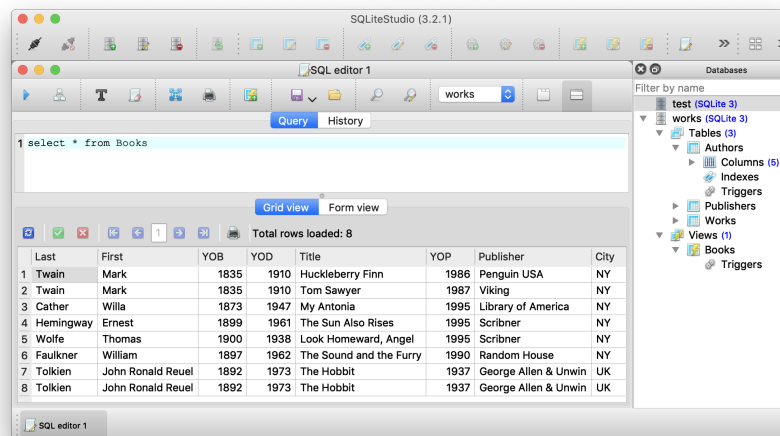
Use AS clauses in SELECT to specify [column names](#).



The proof is in the pudding. We see that Books view behaves exactly like a table when we query from it. Note that in the database schema view on the right the database browser window we can see that it is actually a view.

## Database Views: Persisting Queries (Books Example)

▷ **Example 9.7.11**



## 9.8 Querying via Python

Now it is time to turn to understanding [querying](#) programmatically in `python`. The main concept to grasp is that of a [cursor](#).

### Working with Cursors

- ▷ **Definition 9.8.1** A [cursor](#) is a named object that encapsulates a set of query results in a (virtual) database table.
- ▷ To work with a [cursor](#) in `sqlite3`,
  - ▷ create a [cursor object](#) via the `cursor` method of your database [object](#).
  - ▷ Open the cursor to establish the result set via its `execute` method
  - ▷ Fetch the data into local variables as needed from the [cursor](#).
- ▷ The cursor class in `sqlite3` provides additional methods:
  - ▷ `fetchone()`: return one row as an array/list
  - ▷ `fetchall()`: return all rows a list of lists.
  - ▷ `fetchsome(⟨⟨n⟩⟩)`: return `⟨⟨n⟩⟩` rows a list of lists.
  - ▷ `rowcount()`: the number of [rows](#) in the [cursor](#)

**Intuition:** [Cursors](#) allow programmers to repeatedly use a database [query](#).



©:Michael Kohlhase

231



EdN:3

Again, we fortify our intuitions by making a little example: we pretty-print the some of the information by looping over result of fetching all the records from a given cursor.<sup>3</sup>

### ▷ Extended Example: Listing Authors from the Books Table

#### ▷ Example 9.8.2

```
sql = 'SELECT First, Last, YOB FROM Books WHERE YOD < 1950;'
cursor.execute(sql)
print('There are ', cursor.rowcount, ' books, whose authors died before 1950:\n')
for row in cursor.fetchall():
    print(row[0], ' ', row[1], ' ; born ', row[3], '\n')
print('That is all; if you want more, add more to the database!')
```



©:Michael Kohlhase

232



Finally we come back to the topic of preventing [SQL injection attacks](#). We had seen that these occur when we build the argument string for a `cursor.execute` call. While the single-instruction-restriction of is some help, it is not enough. We essentially have to remove all the [SQL instructions](#) from any input string we substitute with. Fortunately, [SQL](#) is standardized, so we can implement that once and for all.

<sup>3</sup>EdNOTE: MK: show the results

## SQLite3 Parameter Substitution

▷ **Observation 9.8.3** We often need variables as parameters in `cursor.execute`.

▷ **Example 9.8.4** In Example 9.8.2 we can ask the user for a year.

▷ The python way would be to use **f-strings**

```
year = input('Books, whose author died before what year?')
sql = f'SELECT First, Last, YOB FROM Books WHERE YOD < {year}'
cursor.execute(sql) # ⚠ never use f-strings here --> insecure
```

But this leads to vulnerability by **SQL injection attacks**. (↪ **Bobby Tables**)

▷ **Definition 9.8.5** `sqlite3` supplies a **parameter substitution** that **SQL-sanitizes** parameters (removes problematic **SQL instructions**).

▷ The `sqlite3` way uses **parameter substitution** (multiple ? possible ↪ tuple)

```
year = input('Books, whose author died before')
select = 'SELECT Title, YOP FROM Books WHERE YOD < ?'
cursor.execute(select, (year,))
```

or in the “named style” ↪ order-independent (argument is a dictionary)

```
century = input('Century of the books?')
select = 'SELECT Title, YOP FROM Books WHERE YOP <=:start AND YOP >:end'
datadict = {'start': (century - 1) * 100, 'end': century * 100}
cursor.execute(select, datadict)
```



If we have a large number of uniform **SQL instructions**, then we can bundle them, by iterating over a list of **parameters**. In the example below, we explicitly write down the list, but in applications, the list would be e.g. read from a metadata file.

## Inserting Multiple Records (Example)

▷ The `cursor.executemany` method takes an **SQL instruction** with parameters and a list of suitable tuples and executes them.

▷ **Example 9.8.6** So the final form of insertion in Example 9.5.1 would be:

```
booklist = [
    ('Twain', 'Mark', 1835, 1910, 'Huckleberry Finn', 1986, 'Penguin USA', 'NY'),
    ('Twain', 'Mark', 1835, 1910, 'Tom Sawyer', 1987, 'Viking', 'NY'),
    ('Cather', 'Willa', 1873, 1947, 'My Antonia', 1995, 'Library of America', 'NY'),
    ('Hemingway', 'Ernest', 1899, 1961, 'The Sun Also Rises', 1995, 'Scribner', 'NY'),
    ('Wolfe', 'Thomas', 1900, 1938, 'Look Homeward, Angel', 1995, 'Scribner', 'NY'),
    ('Faulkner', 'William', 1897, 1962, 'The Sound and the Fury', 1990, 'Random House', 'NY'),
    ('Tolkien', 'John Ronald Reuel', 1892, 1973, 'The Hobbit', 1937, 'George Allen & Unwin', 'UK')
]
cursor.executemany('INSERT INTO Books VALUES (?, ?, ?, ?, ?, ?, ?)', booklist)
cursor.execute(insert2)
db.close() # clean up by closing
```





## Chapter 10

# Collaboration and Project Management

To facilitate group work – both for the IWGS-II project and future projects down the line, we will start off the semester by looking at state-of-the art project and content management systems and directly use that in the project.

We will concentrate on two parts of such a system:

- collaborative, versioned document/program development via GIT (see Section 10.1)
- issue tracking and management via GitHub/GitLab (Section 10.4).

Systems like GitLab or GitHub also offer additional features like developer communication, continuous integration, automated deployment, monitoring and security management (collectively called DevOps) which are way beyond the scope of IWGS.

### 10.1 Revision Control Systems

We address a very important topic for project management: supporting the life-cycle of project documents, data, and software in a collaborative process. In this Section we discuss how we can use a set of tools that have been developed for supporting collaborative development of large program collections can be used for general project artefact management.

We will first introduce the problems and attempts at solutions and then introduce two classes of revision control systems and discuss their paradigmatic systems.

#### 10.1.1 Dealing with Large/Distributed Projects and Document Collections

In this Subsection we will look at problems in managing the artefacts of large projects. Such projects range from technical documentation for complex systems over knowledge collections like the Wikipedia, to software collections like the Linux kernel. They have in common that a *large group of authors/developers* manage a *large artefact collection* over a *long period of time*.

#### Large/Distributed Collections of Project Artefacts

▷ **Observation 10.1.1** *Artefact collections can become large and long-lived.*

- ▷ **Problem:** How to manage them effectively?
- ▷ **Example 10.1.2** We will use the following projects/systems as running examples and characterize them by size.
  - ▷ The “Subversion Book” [CSFP04] (ca. 450 pages, 9 translations, 3 main authors, hundreds of contributors, since 2002)
  - ▷ **linux** kernel (ca. 16M lines of code, ca. 12 k contributors, since 1991),
  - ▷ wikipedia ( $\geq 5$ M articles,  $\geq 280$  languages, ca. 40M files,  $\geq 130$ k active users, since 2001).
  - ▷ “2048”: a simple browser/app game with lots and lots of variants (forks) in three years.



The first is a relatively standard book about a revision control system (see below), while the wikipedia and **linux** kernel are paradigmatic examples of a large document collections and software development. The last example was chosen as an example of a population of program variants that develop together, exchanging code and ideas as they evolve.

For most of the examples above it is clear that the artefact collections are ever-changing; after all that is their ultimate purpose. But even for documents that we perceive as rather static (e.g. novels) there is a “document lifecycle” – if only before it is published.

## Lifecycle Management for Digital Documents

- ▷ Documents may have a non-trivial life-cycle involving multiple actors.
- ▷ **Example 10.1.3** For any book we have the following stages:
  1. skeleton/layout (chapters, characters, interactions)
  2. first complete draft (given out to test readers)
  3. private editing cycle  $\leadsto$  accepted draft (testing with more readers, refining/condensing the story)
  4. publisher’s editing cycle  $\leadsto$  final draft (professional editor proposes refinements to the draft)
  5. copyediting for spelling, adherence of publisher’s house style
  6. adding artwork/cover  $\leadsto$  first published edition
  7. e-dition (eBook) etc. (different artwork, links, interactivity)
- ▷ **Example 10.1.4** For technical books, multiple editions follow to adapt them to changing domain or correct errors.



For technical documents the lifecycle does not end here. They usually go through several “editions” as the subject matter changes (or the presentation improves). As the revisions can be minor, only parts of the lifecycle described above may be necessary.

As the lifecycle problems are common to all artefact collections, various solutions and practices have evolved to cope with them. We will briefly present and evaluate them in the following.

For all them the critical question is how they deal with multiple files and multiple/distributed authors/developers – a single author/developer working on a single file can usually cope quite well. Multiple variants of the document collections – e.g. in different languages or variants of the domain further complicate matters and mandate system support.

The first practice of collaborating on a document is probably the most widespread: multiple authors collaborate on a single document – or very a limited number of documents and distribute the respective newest state to their collaborators. Some [word processors](#) have support for tracking changes, which may help in the process. Even though the version information could in principle be looked up in the document metadata, it is common practice to add the current date and the last author in the file date.

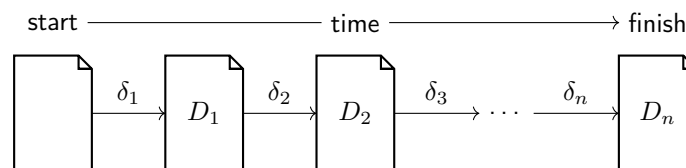
### Document Lifecycle Mgmt. & Collaboration Approaches

▷ **Practice:** Send around MS Word documents by e-mail (dates in file name)

▷ **Characteristics/Problems:**

- ++ well-understood technology (no training need)
- version tracking as a social process (error prone)
- merging diverging versions is annoying (manual process)
- archiving past versions optional/manual (storage problems)
- no multifile support, no snapshots

▷ **Summary:** only supports serial collaboration, no multifile support



larger teams  $\leadsto$  more time wasted



©: Michael Kohlhase

237



The main problem in this practice is that if two – or more – authors change the document in different ways, we say that the document diverges, someone must merge the variants to get to a common state again – a tedious undertaking at best without machine support. The solution to this problem is to socially enforce a linear development timeline: “if you make an iteration until tomorrow morning, then I can take over until noon, ...”.

Instead of distributing the documents to the collaborators we can also upload the respective version to a central server which keeps the respective “current version” for download by the collaborators.

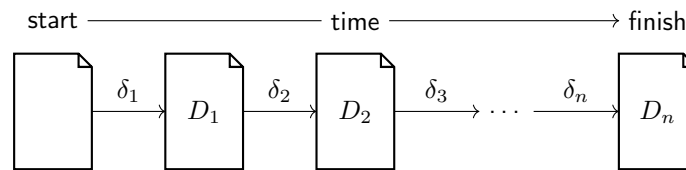
### Document Lifecycle Mgmt. & Collaboration Approaches

▷ **Practice:** Put your documents on Dropbox or MS Sharepoint, or use a Wiki.

▷ **Characteristics/Problems:**

- local install of (proprietary) software
- + auto-synchronization between cloud and user copies upon save
- + auto-archiving past versions in cloud
- merging diverging versions unsupported (manual process)
- no multifile support, no snapshots

▷ **Summary:** only supports serial collaboration



larger teams  $\leadsto$  more time wasted



©: Michael Kohlhasse

238



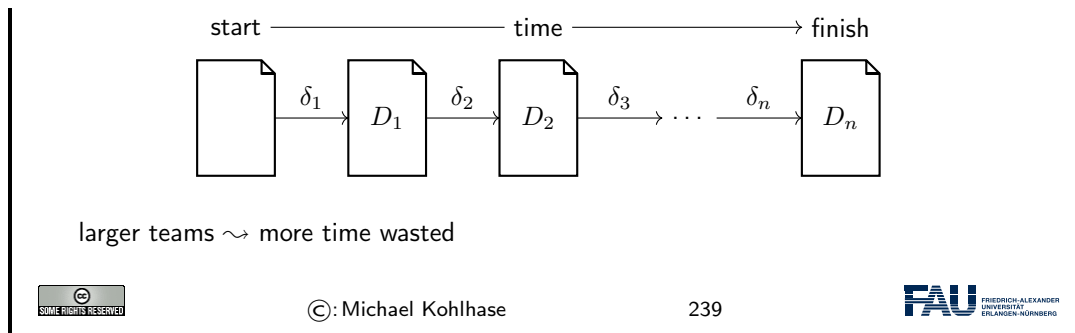
A central server immediately solves the problem of identifying the “current version”, and usually also provides date/time of the last change and the author of that change. A server also enforces a linear development. On a naive server later uploads overwrite previous ones. To remedy this, more advanced servers give the authors access to old versions of documents. This is in fact very important, since it may be necessary to revert certain changes, e.g. to reinstate inadvertent deletions.

While a history-aware server (Dropbox and MS Sharepoint are) allows for a non-linear multi-file development path in principle, system support for this is missing.

The next practice is somewhat complementary from the last, even though it is technically a radical extension: changes are uploaded to the server and merged into the document character-by-character. In particular, this guarantees a linear timeline and a consistent document state.

## Document Lifecycle Mgmt. & Collaboration Approaches

- ▷ **Practice:** Use real-time collaborative editors like EtherPad or wordprocessors like GoogleDocs or Office Online.
- ▷ **Characteristics/Problems:**
  - + browser-based, no installation necessary
  - + real-time auto-synchronization between cloud and user copies
  - +– extremely detailed auto-archiving past versions in cloud
  - no diverging versions
  - no multifile support, no snapshots
- ▷ **Summary:** only supports serial collaboration



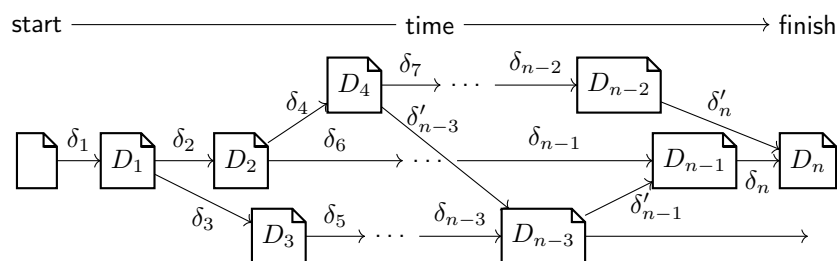
While document consistency is directly guaranteed by the system, intra document, semantic consistency is very hard to achieve, as there is usually no possibility to block out other authors in order to do a larger rewrite. Though the systems give access to the version history, its character-by-character nature makes it very difficult to spot useful versions.

It is a general observation that while real-time collaborative editing is very convenient and effective for single small documents, where semantic intra- and inter-document consistency plays an subordinate role, it does not scale to large document collections and author collectives.

The last practice in collaborative document lifecycle management is to use a revision control system. These systems were originally built for managing the lifecycles of large software projects with multiple, distributed developer groups and even more individual files. As a consequence, they answer all the shortcomings of the practices we have reviewed above, but are restricted to [text files](#) – as programs tend to be.

## Document Lifecycle Mgmt. & Collaboration Approaches

- ▷ **Practice:** Use revision control system (good for ASCII-based file formats)
- ▷ **Characteristics/Problems:**
  - special install, training necessary
  - optimized for character/line-based formats
  - + user-initiated synchronization between cloud and user copies
  - + auto-archiving past versions on server
  - ++ multifile support, snapshots, merging support, tagging
- ▷ **Summary:** supports parallel, branching collaboration



The main idea behind such systems is that we can manage very large document collections and author collectives by making the “document collection changes” – expressed by  $\delta$  in the figure above – the prime objects in our system. Changes can be passed around, applied to working copies, and merged – if we restrict ourselves to [text files](#).

If we look at the paradigmatic document collections from our motivation, then we see that Wikipedia uses the “central server” solution – it is based on a wiki server, while all the others use version control systems.

We will now take a closer look at revision control systems and how they work. Following a somewhat historic path, we will first look at a paradigmatic centralized revision control systems and then advance to the currently dominant distributed system, building on the concepts introduced for the centralized system.

### 10.1.2 Centralized Version Control

We start out with the basics of revision control system based on a relatively simple architecture with a central repository with which all developers interact.

#### Revision Control Systems

- ▷ **Definition 10.1.5** A **revision control system** is a software system that tracks the change process of a document collection via a federation of **repositories** that store the **development history** of the collection. Each step in the **development history** is called a **revision**.
  - ▷ **Definition 10.1.6** Users do not directly work on the **repository**, but on a **working copy** that is synchronized with the repository by **revision control actions**
    1. **checkout**: creates a new working copy from the **repository**
    2. **update**: **merges** the differences between the revision of the working copy and the revision of the **repository** into the **working copy**.
    3. **commit**: transmits the differences between the **repository revision** and the **working copy revision** to the **repository**, which registers them, **patches** the **repository revision**, and makes this the new **repository revision** – called the **head revision** or simply the **head**.
  - ▷ **Observation 10.1.7** The **commits** determine the **revisions** in a **revision control system**.
- Remark:** **Revision control systems** usually store the **head revision** explicitly and can compute **development histories** via reverse **diffs**.



Definition 10.1.5 and Definition 10.1.6 are very general, so that they can cover a wide variety of architectures.

Before we become more concrete, let us have a look at the basic ingredient of **revision control systems**: computing differences, applying them to documents, and reconciling differences.

## ▷ Computing and Managing Differences with diff & patch

- ▷ **Definition 10.1.8** `diff` is a file comparison utility that computes differences between two strings or **text files**: the **source**  $f_1$  and the **target**  $f_2$ . Differences are output linewise in a **diff**  $\delta(f_1, f_2)$ .
- ▷ **Definition 10.1.9** `patch` is a sister utility that applies a **diff**  $\delta := \delta(f_1, f_2)$  to  $f_1$  – resulting in  $f_2$ ; we say it **patches**  $f_1$  with  $\delta$ .
- ▷ **Example 10.1.10** We compare two simple **text files**:

The quick brown fox jumps over the lazy dog	The quack brown fox jumps over the loozy dog	1c1,2 < The quick brown ---- > The quack brown > 3c4 < the lazy dog ---- > the loozy dog
---------------------------------------------------	----------------------------------------------------	------------------------------------------------------------------------------------------------------------------

- ▷ **Definition 10.1.11** A **diff** consists of a sequence of **hunks** that in turn consist of a **locator** which indicates the **source line number** followed by the lines **deleted** in the **source** and **added** in the **target**.



In practice, – unlike in our didactic example – **diffs** are usually (much) smaller than either the **source** or the **target**. This makes the design decision of passing around **diffs** instead of files in **revision control systems** efficient.

For **revision control systems** we need more than just `diff` and `patch`. When we are sending around **diffs** along non-linear **development histories**, then we also have to reconcile **diffs** that come via different paths.

## Merging Differences

- ▷ There are basically two ways of **merge** the differences of files into one.
- ▷ **Definition 10.1.12** In **two-way merge**, an automated procedure tries to combine two different files by copying over differences by guessing or asking the user.
- ▷ **Definition 10.1.13** In a **three-way merge** the files  $f_1$  and  $f_2$  are assumed to be created by changing a joint original (the **parent**)  $p$  by editing.  
If there are **hunks**  $h_1$  in  $\delta(f_1, p)$  and  $h_2$  in  $\delta(f_2, p)$  that affect the same **line** in  $p$ , then we call the pair  $(h_1, h_2)$  a **conflict**.  
The result of a **three-way merge** are two **diffs**  $\mu_i^3(f_1, f_2, p)$ , which contain the non-conflicting differences of  $\delta(f_i, p)$  and (representations called **conflict markers** of) the **conflicts**.
- ▷ **Note:** In **revision control systems** **conflicts** must be **resolved** by choosing one of the

alternatives or creating a manually merged revision before changes can be [committed](#).



## Merging Differences with merge3

▷ **Definition 10.1.14** The `merge3` tool computes a [three-way merge](#).

▷ **Example 10.1.15** We compare two simple [text files](#) with a parent:

mine.txt	your.txt	parent.txt	conflict marker
This is the file. Hello	This is the file. hello	This is the file. hi	This is the file. <<<<<<< mine.txt Hello       parent.txt hi ===== hello >>>>>>> your.txt

**Remark:** The [conflict markers](#) in actual [revision control systems](#) are similar, but may vary.

▷ **Note:** There are good visual [merge3](#) tools that help you cope with merges. Some [text editors](#) also have support for [resolving conflict markers](#).

▷ **Remark:** There are analoga to `diff` and `patch` for other file formats, but in practice, [revision control](#) is mostly restricted to [text files](#).



With this, we can now understand the revision control workflows in our concrete system.

In its simplest form, a [revision control system](#), can be understood using the Subversion system that is heavily used in open source projects that have a relatively hierarchical development model.

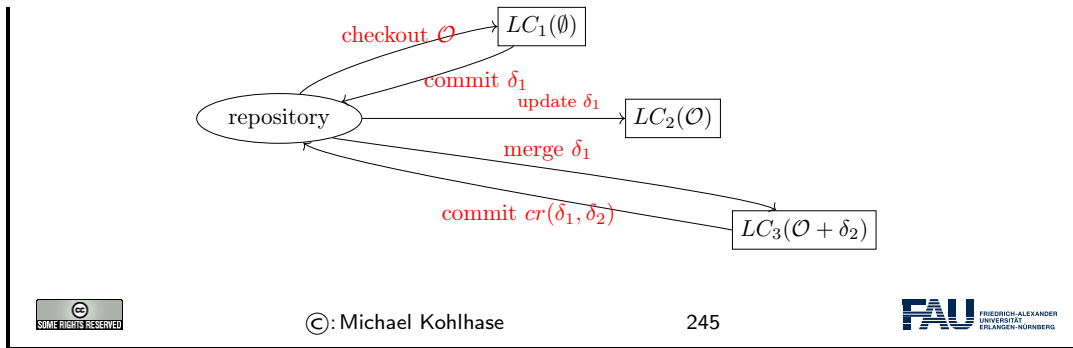
## Centralized Version Control (e.g. with Subversion)

▷ **Definition 10.1.16** Subversion is a centralized [revision control system](#) that features

- ▷ a single, central [repository](#) (for current revision and reverse diffs)
- ▷ local [working copies](#) (asynchronous checkouts, updates, commits)

They are kept synchronized by passing around [diffs](#) and [patching](#) the [repository](#) and [working copies](#). Conflicts are resolved by (three-way) [merge](#).

▷ **Example 10.1.17 (A Workflow with three Working Copies)**



In the workflow of Example 10.1.17 is a typical one:

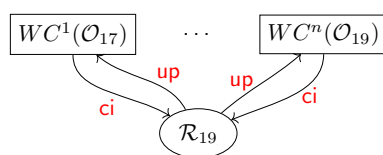
1. A first user **checks out** a new **working copy**  $LC_1$ , from the empty repository, adds a couple of files – we denote the new document collection at this point with  $\mathcal{O}$ , and **commits** the difference  $\delta_1$  between the working copy and  $\mathcal{O}$  to the repository which  $\delta_1$  logs it as “revision 1”.
2. There is another repository  $LC_2$ , which has been checked out earlier (i.e. based on “revision 0”), and which is now no longer in sync with the repository. So we can **update** (i.e. **patch**) it to “revision 1” by transferring  $\delta_1$  to  $LC_2$ , which thus has same content as  $LC_1$ , namely  $\mathcal{O}$ .
3. For a third repository  $LC_3$  which has been checked out at “revision 0” we assume that it has been changed by adding different files, the difference being  $\delta_2$ . Note that as these changes are relative to “revision 0”, they cannot simply be committed to the repository. Therefore we need to **update** it. As  $LC_3$  already contains changes, this amounts to a **merge** of  $\delta_1$  and  $\delta_2$  to get a new local copy that is essentially  $\mathcal{O} + \delta_2$ , which is now relative to “revision 1”. This can now be **committed** to the repository to form “revision 2”.

**Note:** that in all of this it does not matter who the authors of the respective changes and the owners of the respective working copies are. They might be different persons, or a single author might have multiple working copies, e.g. one on the work computer, one on a laptop, and one on the home desktop. They are all held in sync by **updates**, **commits**.

With this basic mechanism, we can already model quite complex and collaborative workflows. The basic idea is simple: we just use the **update/commit** cycle to synchronize a set of working copies.

### Collaboration via Revision Control (e.g. Subversion)

- ▷ **Idea:** We can use **revision control** for collaboration with multiple **working copies**.
- ▷ **Diff-Based Collaboration:** Subversion takes care of the synchronization:



```

23
24 class String
25 <==== HEAD:lib/jekyll/core_ext.rb
26 def cutoff(desired = 5)
27 =====
28 def cutoff(desired = 400)
29 >>>>>> conflicts:lib/jekyll/core_ext.rb
30 return self if self.length <= desired

```

- ▷ you can only **commit**, if your revision is the **head** (otherwise **update**)
- ▷ update **merges** the **changes** into your **working copy**.
- ▷ If there are changes on the same line, you have a **conflict**, which must be **resolved**.



**Note:** that these collaborative workflows can be asynchronous. In particular working copies can lag behind the repository as long as they want – they only have to synchronize for **commits**. This gives a lot of freedom in the development process.

**Also note:** that unless the repository and the working copies are on the same computer – which is somewhat unlikely. Commits and updates are only possible while online, this sometimes prevents authors/developers from grouping changes logically as they have to collect them until they are online again.

Subversion even allows to **update** to a specific revision, e.g. if an author wants to base her work on that – or wants to revert some changes<sup>1</sup>. In fact, Subversion supports branching: committing different development lines to the repository, but we will not go into this here and leave the discussion for later when we discuss distributed revision control systems where branching is the main mechanism of operation.

### Branching: Supporting Multiple Lines of Development

- ▷ **Observation 10.1.18** *A central repository entails – ultimately – a single line of development.  $\Leftarrow$  changes have to be **merged** into the **repository** eventually.*
- ▷ **But:** we want to develop – and **commit** – to variants in parallel.
- ▷ **Definition 10.1.19** A **branch** is a copy of an object under revision control (such as a source code file or a directory tree) so that it can be developed in parallel.
- ▷ In particular, **branches** allow parallel development histories via separate **commits**.
- ▷ **commits** from one **branch** can be **merged** into another.
- ▷ **Example 10.1.20** In software development we profit from separate
  - ▷ **master branch/trunk**– main line of development, used for integration.
  - ▷ **release branch**– only bug fixing; no new features
  - ▷ **feature branch**– develop a new feature; close branch upon merge
  - ▷ **staging branch**– integrate multiple fixes/features
- ▷ **Definition 10.1.21** A **branch** controlled by a different developer or not intended to be **merged** back is called a **fork**.



Branches are easy to realize in the **diff/patch/merge**-based architecture.

### 10.1.3 Distributed Revision Control

In this Subsection we will introduce distributed revision control systems using the GIT system as an example. As this is the currently dominant system, we will also go into more detail about concrete usage of the system.

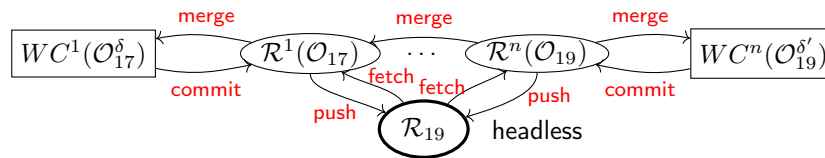
<sup>1</sup>Don't drink and write!

## Distributed Version Control

### Problems with Centralized Revision Control (Subversion):

1. we can only commit when online!
2. all collaboration goes via **one, central repository**. (prescribes workflow)

### Idea: Distribute the repositories and move patches between them.



1. **local commits** to **local repositories**
2. **all repositories created equal** (flexible organization)

▷ **Definition 10.1.22** We call a **revision control system distributed**, iff it allows multiple **repositories** that can exchanged **patches**. Contrastingly we call a **revision control system centralized**, if it only allows one **repository**.

▷ **Definition 10.1.23** We call a repository **headless** (or **bare**), if used without working copy, usually in a **web server**.



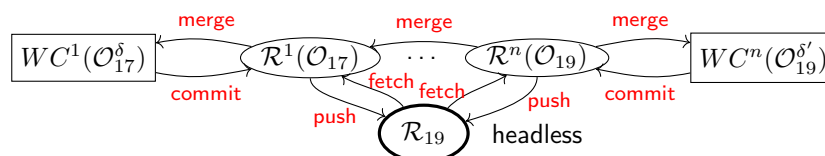
The concept of **distributed revision control systems** is motivated by the two shortcomings at the top of the slide, which can be remedies by a single – if relatively radical idea: allowing lots of repositories that can communicate with each other by exchanging **patches**. Local repositories allow commits while offline and distributed repositories allow for flexible architectures.

Of course, there is a price to pay: instead of having three main **revision control actions** we now have five. We need to be able to move commits to a remote repository and fetch commits from one. This makes the model quite a lot more complicated.

## Centralized vs. Distributed Version Control

▷ **Intuition:** **Distributed revision control systems** generalize centralized ones.

Centralized	Distributed	Centralized	Distributed
repository	headless repository	commit	commit + push
working copy	repository + working copy	update	fetch + merge
		checkout	fetch + checkout





We now come to the most prominent of the **distributed revision control system**: GIT. It implements the concepts motivated above. Somewhat paradoxically, the **distributed** nature of the workflows makes it simpler and more efficient to implement.

### Distributed Version Control with GIT

▷ **Definition 10.1.24** GIT is a distributed **revision control system** that features

- ▷ **local repositories** in each **working copy**  $\leadsto$  local **commit/merge**
- ▷ multiple **remote repositories** connected to a **local repository**
  - ▷ **clone** a **remote repository**  $\leadsto$  make **local repository/working copy**
  - ▷ **local repository** changes can be **fetched** from and **pushed** to a **remote repository** (the **upstream/downstream** repositories).
  - ▷ The dual to a **push** is a **pull**, it updates the **working copy** from a **remote repository**:  $\text{pull} \hat{=} \text{fetch} + \text{merge}$ .
- ▷ **branches** and **forks** (**remote upstream repository**)

**Software Support:** There are various software systems that facilitate providing repositories, e.g.

- ▷ ▷ GitHub, a **repository hosting service** at <http://GitHub.com> (free public/private repositories)
- ▷ GitLab, an open source **repository management system** and **repository hosting service** at <http://GitLab.com> (free public/private repositories)



#### 10.1.4 Working with GIT in small Projects

Now that we understand the concepts, let us see how we can use them in practice. For this we assume that students have installed GIT on their computers, so that they can use it; [CS14, section 1.5] gives an excellent introduction.

For this Subsection, we restrict ourselves to the workflows in small projects, where a simple centralized structure suffices. Also, we explain GIT functionality “from scratch”, and do not presuppose a **repository management system**.

In all of our concrete examples, we will use UNIX **shell commands**; for Windows users should use the GIT **shell**, a GIT-enhanced version of the UNIX **shell** that comes with the GIT distribution, and *not* the Windows command prompt. There are graphical front-ends for the GIT client, but our experience shows that using **shell commands** helps understand the concepts and workflows much better.

#### Working with GIT (Initializing a Local Repository)

- ▷ Download GIT from <https://git-scm.com/downloads>, install (you want to use it on your local machine)
- ▷ We will use git from the **shell** on your system (Mac OS X or linux) or Git Bash

that comes with your GIT download (Windows). (graphical front ends exist but hinder understanding)

▷ Test whether your installation works: `git --version`

▷ **Definition 10.1.25** Initialize a local repository:

- ▷ `git init` turns the current directory into a GIT working copy by adding a local repository as a hidden `.git` folder.
- ▷ `git init <name>` makes working copy + local repository in the <name> sub-directory.

**Alternative:** Clone a remote repository, i.e. `git init + git pull`

```
git clone https://gitlab.cs.fau.de/iwgs-ss19/collaboration.git
Cloning into 'collaboration'...
Username for 'https://gitlab.cs.fau.de': yp70uzj
Password for 'https://yp70uzj@gitlab.cs.fau.de':
...
```

▷



Before you start, you should configure some global options for GIT (just adapt the following lines and type them into the shell).

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

The following two lines configure GIT to always pull the branch called **master** from the repository called **origin**

```
$ git config branch.master.remote origin
$ git config branch.master.merge refs/heads/master
```

With this configuration you can replace `git push origin master` with a simple `git push`.

## Working with GIT (Remote Repositories)

▷ **Recap:** A repository can be connected to one or several remote repositories.

▷ GIT commands for working with remote repositories:

<code>git remote add &lt;name&gt; &lt;URI&gt;</code>	gives the repos at <URI> the name <name>
<code>git remote</code>	shows names of all remote repositories

▷ `git remote -v` shows the remote repositories e.g.

```
MiKo:collaboration kohlhase$ git remote -v
origin https://gitlab.cs.fau.de/iwgs-ss19/collaboration.git (fetch)
origin https://gitlab.cs.fau.de/iwgs-ss19/collaboration.git (push)
```

▷ `git remote add <name> <URI>` adds remote repositories e.g.

```
kohlhase$ git remote add upstream git@gl.kwarc.info:test/collab.git
kohlhase$ git remote -v
origin https://gitlab.cs.fau.de/iwgs-ss19/collaboration.git (fetch)
```

```
origin https://gitlab.cs.fau.de:iwgs-ss19/collaboration.git (push)
upstream https://gl.kwarc.info:test/collab.git (fetch)
upstream https://gl.kwarc.info:test/collab.git (push)
```

- ▷ We can now **pull/push** to the new **remote repository**, e.g. `git push upstream master`
- ▷ **Note:** `git push` is just syntactic sugar for `git push origin master`



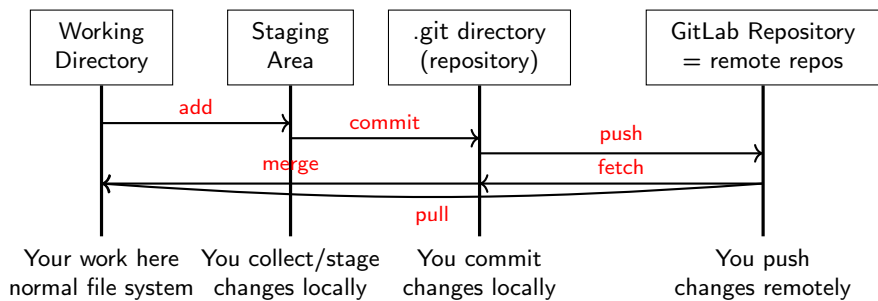
We will now come to a GIT peculiarity that is important to understand for working with GIT: Often we only want to **commit** only a subset of the changed files – e.g. because the changes already constitute a achievement of their own or we want to split the development into multiple **commits**. There are essentially two ways of achieving this.

1. giving the **commit** action a list of files to be committed, or
2. marking files for a future **commit** – this is called **staging**.

The second method is more flexible, since we do not have to remember which files participate in a commit and we can **stage** files as we go along. Therefore GIT uses this method, even though it adds conceptual complexity – actually, the first method can be recovered by syntactic sugar.

## Working with GIT (Staging and Committing)

- ▷ **Overview:** GIT local workflow: **staging** files for **commit**, by `git add`



**commits** acts only on staged files ~ `git add foo.tex` (GIT must know about them)



## Working with GIT (Staging and Committing)

- ▷ basic GIT commands (there are many variants and options ~ study them)

<code>git clone «URI»</code>	clones the repos at «URI»
<code>git add «file»</code>	stages «file»
<code>git commit -m'«msg»'</code>	commits staged files with commit message «msg»
<code>git status</code>	gives information about the working copy.
<code>git push «repos» «branch»</code>	pushes all commits to branch «branch» on «repos»
<code>git pull «repos» «branch»</code>	fetches and merges branch «branch» from «repos»



We have only shown the most basic commands here. There are many other commands and options that make your life much easier. For instance, the `-a` option is very useful for `git commit`: it automatically stages all the changed files. `git commit -am'foo'` commits all your change in the current directory (which is often what you want).

Let us now fortify our intuition on working with GIT by exhibiting a typical (but elementary) workflow.

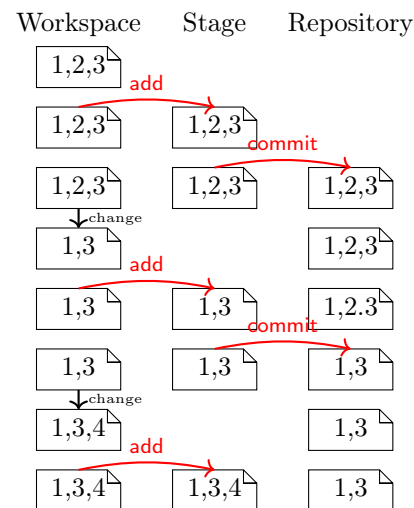
### An Example Git Workflow

▷ **Example 10.1.26** A typical, elementary workflow in GIT

```
> git init
Initialized empty Git repository in /tmp
> echo "1,2,3" > test.txt
> git add test.txt
> git commit -m'initializing'

> echo "1,3" > test.txt
> git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update ...
  (use "git checkout -- <file>..." to...
        modified:   test.txt
no changes added to commit
(use "git add" and/or "git commit -a")

> git add test.txt
> git commit -m'bla' test.txt
> echo "1,3,4" > test.txt
> git add test.txt
```



Note that the `shell` command `echo <string> > <file>` updates the contents of the file `<file>` to `<string>` or creates `<file>` with this content in the first place. We use this command to make the file changes visible in the `shell` on the left side.

### 10.1.5 Working with GIT in large Projects

In this Subsection, we will (further) discuss the concepts for using GIT in large, long-lived projects. This is less important for IWGS, since projects are rather small. But we want to at least make students aware of GIT branching and the GIT flow paradigm, and we want to clear up the mystery of which GIT often speaks of `master`.

We can now come back to the topic, where GIT really shines: [branching](#). The main reason for this is that [merging](#) is so well-supported in GIT. Together with the distributed “local-repository” architecture, this allows for very flexible organization of workflows. We will discuss the basics of branch-based and fork-based workflows here.

### GIT Branches and Forks

▷ GIT special commands for making, switching, and merging branches.

<code>git branch &lt;&lt;branch&gt;&gt;</code>	makes a branch with name <<name>>
<code>git checkout &lt;&lt;branch&gt;&gt;</code>	switches a working copy to branch <<branch>>
<code>git branch -v</code>	shows all branches
<code>git branch -d &lt;&lt;branch&gt;&gt;</code>	deletes branch <<branch>>

**Intuition:** In GIT branches are very similar to repositories, but more lightweight. Repositories can have different permissions; branches inherit these.

► **Fork-based Collaboration:** If you want to contribute to a repository  $\mathcal{R}$  you have no push-rights on,

1. **clone**  $\mathcal{R}$  to a new repository  $\mathcal{R}'$  you own (i.e. **fork** it;  $\mathcal{R}'$  is a **fork** of  $\mathcal{R}$ )
2. develop your contribution on  $\mathcal{R}'$ .
3. ask  $\mathcal{R}$ s owners to pull from  $\mathcal{R}'$  (**pull request**)

GIT repository management systems like GitHub and GitLab support this.

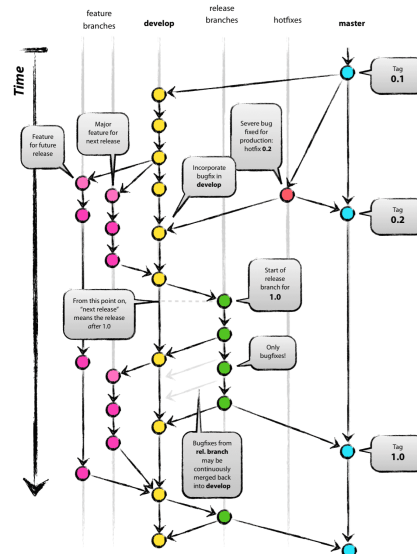


What we have seen above, let us briefly discuss an elaborate workflow suitable for large development teams, which has become known under the name “GitFlow”.

## GitFlow: An Elaborate Development Model based on GIT

► **Definition 10.1.27** [Dri10] suggests a development model called **GIT flow**

- A **master branch** master that all other branches merge into
- New functionality is developed “feature-by-feature” on **feature branches**
- A **development** branch (usually called **devel**) that integrates all feature branches and is merged into **master** once the integrated functionality is stable.
- (possibly) **release branches** for every release; they collect bugfixes, but no new features.



## 10.2 Working with GIT and GitLab/GitHub

In principle we know all we need for running GIT in practice. But if we want to make use of [remote repositories](#) – and without that, we lack most of the advantages of [revision control systems](#) – we have to deploy a [web server](#) which takes on the upstream repositories.

Even though this is relatively simple to set up, there are now dedicated [web applications](#) that supply [repositories](#) and additional project management infrastructure.

### Working with GitLab/GitHub

- ▷ GIT it sufficient to set up a [remote repository](#). (but tedious [CS14, chapter 4])
- ▷ **Idea:** Use a GIT [repository manager](#) like GitLab/GitHub (we use GitLab)
- ▷ **Definition 10.2.1** A [repository management system](#) is an [web application](#) that supports the administration of a [repository server](#), i.e. [web server](#) that provides access to a set of [headless repositories](#) and manages [authentication](#) and [authorization](#).
- ▷ **Example 10.2.2** GitLab is an open source [repository management system](#) and [repository hosting service](#) at <http://GitLab.com> (free public/private repositories)
- ▷ **Definition 10.2.3** A [repository hosting service](#) is a web-based [repository management system](#) that also offers storage space for [repositories](#).
- ▷ **Example 10.2.4** GitHub is a [repository hosting service](#) at <http://GitHub.com> (free public repositories)  
 GitHub is now the default hosting service for open source software development, it hosts more than 50 Million [repositories](#).



We could be using GitHub for IWGS – and we would probably do so for an open-source software project – but we will use the FAU offering: a GitLab instance that offers [repository hosting](#) to all FAU members and login via IDM. The instructors of IWGS have installed a special [group](#) for [repository hosting](#).


### Working with GitLab/GitHub (continued)

- ▷ **Definition 10.2.5** Often, [repository management systems](#) organize [repositories](#) (called [projects](#) in GitLab) hierarchically into [groups](#) (also called [namespaces](#)) and provide a [personal group](#) to all users.
- ▷ **Concretely:** we use the FAU GitLab: <https://gitlab.cs.fau.de>
  1. sign in with the FAU Single Sign On (aka. [IDM account](#))
  2. this makes an account there and gives you a [personal group](#) <https://gitlab.cs.fau.de/⟨SSID⟩>
  3. IWGS has a course [group](#) <https://gitlab.cs.fau.de/iwgs-ss19> (the [course project goes there](#))



Now we are ready to play with GitLab, and please do, there is nothing you can do wrong. And – that is the beauty of revision control systems – few things you cannot undo.

## Making Repositories on GitLab

- ▷ Make a new **project** with , play with it (you can always delete it)
  - ▷ **Definition 10.2.6** **Group/project** visibility can be one of three states:
    - ▷ **Private**: Project access must be granted explicitly to each user.
    - ▷ **Internal**: The project can be accessed by any authenticated user.
    - ▷ **Public**: The project can be accessed without any authentication.
- Private** and **public** make most sense in our setting.
- ▷ **Exercise**: Make a repository, clone it locally, add a file to it, commit that, let your friends clone/change/commit it, merge their changes, ... (see the homework)



To understand what these visibility levels mean, we have to talk about authorization in GitLab, i.e. how we can manage what interactions a particular (class of) user is allowed to do.

## Authorization in GitLab: Managing Access Permissions

- ▷ **Definition 10.2.7** **Authorization** refers to a set of rules that determine who is allowed to do what.
- ▷ **Definition 10.2.8** **Authorization** is often operationalized by assigning **permission levels** and binding the **authorization** to execute particular interactions to **permission levels**.
- ▷ **Definition 10.2.9** GitLab has five **permission levels** for **repositories**:
  1. **guests** can **clone** and see/**report issues** ...
  2. **reporters** can also **assign issues** ...
  3. **developers** can also **push**, create **branches** ...
  4. **maintainers** can also assign **permission levels** ...
  5. **owners** can also delete **repository** ...

**Intuition**: In a **public repository**, everyone is **guest**, in a **internal** one, logged in users are.



## 10.3 Excursion: Authentication with SSH

We now come to a topic that is of practical relevance, whenever we work with [web applications](#) that work with restricted resources – in this case the content of your [private repositories](#): [authentication](#). Generally, there are two [authentication](#) methods: the one via passwords built into HTTPS and ssh-authentication, which we will briefly discuss here, since it is the more convenient method for interacting with GitLab (and GitHub).

Before we come to ssh-authentication, let us clarify the concept of authentication in general.

### ▷ Authentication

- ▷ **Definition 10.3.1** [Authentication](#) is the process of ascertaining that somebody really is who they claim to be.
- ▷ **Definition 10.3.2** [Authentication](#) can be performed by ascertaining an [authentication factor](#), i.e. testing for something the user
  - ▷ [knows](#), e.g. a password or answer to a security question – [knowledge factor](#)
  - ▷ [has](#), e.g. an ID card, key, implanted device, software token, – [ownership factor](#)
  - ▷ [is](#) or [does](#), e.g. a fingerprint, retinal pattern, DNA sequence, or voice – [inheritance factor](#).

**Note:** Password [authentication](#) is known to be problematic. (and you have to remember/type it)

- ▷ **One Problem:** Server and user must both know the password to [authenticate](#) passwords are symmetric keys: the server can leak them.



We now come to an authentication method that leaves the user out of the loop completely. It works via cryptographic keys, which are exchanged between the [GIT client](#) and [server](#). In this particular setup, we make use of [public key cryptography](#), which only transfers public keys and keeps the private keys local; minimizing the user of passwords and leakage.

The details of this are quite involved, so we only give a very brief introduction of the moving parts.

### Authentication by Cryptographic Public Keys

- ▷ **Definition 10.3.3** [Cryptography](#) is the practice of transmitting a [plain text](#)  $t$  by [encoding](#) it into a [cypher text](#)  $t'$ , to hide its content from anyone but the legitimate receiver who can [decode](#)  $t'$  to  $t$ .
- ▷ [Public key cryptography](#) split the key into an [encode key](#)  $e$  and a [decode key](#)  $d$ 
  - ▷ key  $e$  can encode a text  $t$  to  $t'$ , but only  $d$  can decode  $t'$  to  $t$ .
- ▷ built into the SSH communication protocol.
  1. user generates key pair  $(e, d)$ , deposits  $d$  on server as certificate, keeps  $e$  secret.

2. user encodes a text  $t$  with  $e$  to  $t'$  send  $t + t'$  to server
3. server decodes  $t'$  to  $t''$  with  $d$  and verifies  $t = t'' \leadsto$  OK, iff  $t = t''$ .

▷ **Advantage:** Passwords cannot be leaked, need not be transmitted, retyped.



In practice, working with SSH-based authentication is quite easy to work with: we have to generate a public/private key pair – there are standard utilities for that, deposit the public key in GitLab, and then use **clone** using the SSH URI supplied by GitLab.

## Working with GIT (Cloning a Remote Repository with SSH)

▷ **Alternative:** **Clone** a **remote repository** via SSH URL

```
kohlhase$ git clone git@gitlab.cs.fau.de:iwgs-ss19/collaboration.git
Cloning into 'collaboration'...
remote: Enumerating objects: 12, done.
remote: Counting objects: 100% (12/12), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 12 (delta 1), reused 0 (delta 0)
Receiving objects: 100% (12/12), done.
Resolving deltas: 100% (1/1), done.
```

▷ **But we need a key pair** for this to work.

Go to <https://gitlab.cs.fau.de/profile/keys> and follow the instructions there

▷ **essentially:** generate a key pair, copy one into GitLab.



We will now complement **revision control systems**, as discussed above, with **issue tracking systems**. The former support dealing with changes in the collaborative development of document collections, the latter support the collaborative management of **issues** – the reasons for changes.

## 10.4 Bug/Issue Tracking Systems

In this Section we will discuss **issue tracking systems**, which support the collaborative management of reports on a particular problem, feature request or general task, as well as its status and other relevant data. These systems originated from tracking systems for help desks and in software engineering, but have evolved into general project planning systems.

### **issue tracking systems**

We will mainly look at systems that originate from software engineering applications here.

## Bug/Issue Tracking Systems

▷ **Definition 10.4.1** An **issue tracker** (also called **issue tracking system** simply

**bugtracker**) is a software application that keeps track of reported **issues**– i.e. software bugs, tasks, and feature requests – in software development projects.

▷ **Example 10.4.2** There are many open-source and commercial **bugtrackers**

- ▷ bugzilla: <http://bugzilla.org> (Mozilla's bugtracker)
- ▷ TRAC: <http://trac.edgewall.org> (mostly for Subversion)
- ▷ GitHub: <http://github.com> (probably the most used)
- ▷ GitLab: <http://gitlab.com> (open source version of GitHub)
- ▷ JIRA: <https://www.atlassian.com/software/jira> (proprietary)

Most **bugtrackers** are **web applications** and also integrate a **wiki** and integrate a **revision control system** via extended **markdown**.



©: Michael Kohlhasse

265



It is no coincidence that **issue trackers** often come bundled with **revision control systems**; they form the perfect complement: while the latter track large digital artefacts over extended development cycles, the **issue trackers** track the tasks induced by the development over the same time frame. It is natural that the two should be well-synchronized for a successful development project.

**Issue trackers** manage **issues** and track their status over its whole lifetime – from the initial report to its resolution. This results in a particular set of components that are present in all systems.

### ▷ The Anatomy of an Issue

▷ **Definition 10.4.3** An **issue** (or **bug report**) specifies

- ▷ **title**: a short and descriptive overview (one line)
- ▷ **description**: a precise description of the expected and actual behavior, giving exact reference to the component, version, and environment in which the bug occurs. (bugs must be reproducible and localizable)
- ▷ **issue metadata**: who, when, what, why, state, ... (see below)
- ▷ **conversation**: a forum-like facility for discussing an **issue**.
- ▷ **attachment**: e.g. a screen shot, set of inputs, etc.

▷ **Definition 10.4.4** A **feature request** is an **issue** that only specifies the expected behavior and proposes ways of implementing that.



©: Michael Kohlhasse

266



The **conversation** of an **issue** is a lightweight text category, which should be efficient to write, but has some structure to make reading and understanding the concepts and details involved. In particular, it is important to be able to refer to the program code, other issues, other developers, commits, etc.

Most **bugtrackers** use the **markdown** format, which strikes a good balance between structure and brevity of **markup codes** and extend it with **bugtrackers-specific markup**.

We use the opportunity to introduce **markdown** in general before we come to the extensions.

## Markdown a simple Markup Format Generating HTML.

- ▷ **Idea:** We can translate between **markup formats**.
- ▷ **Definition 10.4.5** **Markdown** is a family of **markup formats** whose **control words** are unobtrusive and easy to write in a **text editor**. It is intended to be converted to HTML and other formats for display, e.g. in
- ▷ **Example 10.4.6** **Markdown** is used in applications that want to make user input easy and effective, e.g. **wikis** and **issue tracking systems**.
- ▷ **Workflow:** Users write **markdown**, which is formatted to HTML and then served for display.
- ▷ A good cheat-sheet for **markdown control words** can be found at <https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet>.



Instead of introducing the **markdown** syntax systematically, let us look at an example that shows the most prominent **control words** in action and see how things look in a **markdown**-based application (and behind the scenes as HTML).

## Markdown a simple Markup Language Generating HTML

- ▷ **Example 10.4.7** We show the most important Markdown commands.

Markdown syntax	Generated HTML
<pre># Heading ## Sub-heading ### Another deeper heading  Paragraphs are separated by a blank line.  Two spaces at the end of a line leave a line break.  Text attributes <i>italic</i> , <b>bold</b>**, 'monospace'.  Bullet list: * apples * oranges * pears  Numbered list: 1. apples 2. oranges 3. pears  A [link](http://example.com).</pre>	<pre> Heading  Sub-heading  Another deeper heading  Paragraphs are separated by a blank line.  Two spaces at the end of a line leave a line break.  Text attributes <i>italic</i>, <b>bold</b>, <code>monospace</code>.  Bullet list: <ul style="list-style-type: none"> <li>• apples</li> <li>• oranges</li> <li>• pears</li> </ul>  Numbered list: <ol style="list-style-type: none"> <li>1. apples</li> <li>2. oranges</li> <li>3. pears</li> </ol>  A <a href="http://example.com">link</a>.</pre>



Markdown was originally developed for wikis, and its markup infrastructure reflects that. For use in issue tracking systems, we need to also reference to the program code, other issues, other developers, commits, etc.

## GitHub flavored markdown: Tracker-Specific Extensions

- ▷ **Remark 10.4.8** Source code hosting systems offer special extensions for referencing their components.
- ▷ **Definition 10.4.9** GitHub flavored markdown (GFM) is a markdown dialect extended for the use in GIT-based issue tracking systems; see [GFM:on] for the specification.
- ▷ **Example 10.4.10** GitHub/GitLab recognize most of GFM, most usefully
  - ▷ @foo for team members (@all for all project members), e.g. cc: @miko
  - ▷ #123 for issues, e.g. depends on #4711
  - ▷ !123 for merge requests, e.g. but merge #19 first
  - ▷ \$123 for code snippets, e.g. see \$123 for an example usage

- ▷ 1234567 for commits, e.g. *fixed by 4c0decb yesterday.*
- ▷ [file](path/to/file) for file references,  
e.g. *as we see in [pre.tex](../lib/pre.tex)*
- ▷ **Observation 10.4.11** *Very useful for project planning and reporting in GitLab and GitHub.*



The anatomy of an issue only enables/restricts the form of an issue, not what would help the project along. We will explore that – to get you thinking – in a counter-example and the show what would have helped the developers.

## Issues – How to Write a Good One

- ▷ The **descriptions** or **issues** should be concise, but describe all pertinent aspects of the situation leading to the unexpected behavior.
- ▷ **Example 10.4.12 (A bad bug report description)**  
*My browser crashed. I think I was on foo.com. I think that this is a really bad problem and you should fix it or else nobody will use your browser.*
- ▷ **Example 10.4.13 (A good one)**  
*I crash each time I go to foo.com (Mozilla build 20000609, Win NT 4.0SP5). This link will crash Firefox reproducibly unless you remove the border=0 attribute:*  
`<IMG SRC="http://foo.com/topicfoos.gif" width=34 border=0 alt="News">`

**Remember:** developers are also human (try to minimize their work)  
Think about what would help you understand and reproduce the problem.



Let us now survey the typical workflow supported by a **issue tracking systems** by presenting the typical life-cycle of an **issue**.

## ▷ Bugtracker Workflow

- ▷ **Definition 10.4.14 (Typical Workflow)** supported by all **bugtrackers**
  - ▷ user **reports issue** (files report in the system)
  - ▷ other users extend/discuss/up/downvote **issue**
  - ▷ QA engineer **triages** issues – classification, remove duplicates, identify dependencies, tie to component, ... and **assigns** to developer.
  - ▷ developer **accept** or re-**assigns issue** (fixes who is responsible primarily)
  - ▷ project planning by identification of sub-issues, dependencies (new issues)
  - ▷ bug fixing (design, implementation, testing)
  - ▷ issue landing (sign-off, integration into code base)
  - ▷ release of the fix (in the next revision)

- ▷ QA engineer or developer **close s issue**
- ▷ **Observation 10.4.15** An **issue tracker** can serve as a full-blown project planning system, if used accordingly.
- ▷ **Definition 10.4.16** For timing work plans, most **issue trackers** provide **milestones** that **issues** can be targeted to.



The workflow presented on the last slide is supported by metadata recorded in the **issue**, most importantly some kind representation of a **issue state**.

### Administrative Metadata for Issues

- ▷ To make the **issue**-based workflows work we need data.
- ▷ **Definition 10.4.17 (Administrative Metadata)**  
**Issue metadata** can specify
  - ▷ **issue number**: for referencing with e.g. #15
  - ▷ an **assignee**: a developer currently responsible
  - ▷ **participants**: people who get notified of changes/comments
  - ▷ **labels**: for specializing bug search
  - ▷ a **state**: e.g. one of new, assigned, fixed/closed, reopened.
  - ▷ a **resolution** for fixed bugs, e.g.
    - ▷ **FIXED**: source updated and tested
    - ▷ **INVALID**: not a bug in the code
    - ▷ **WONTFIX**: “feature”, not a bug
    - ▷ **DUPLICATE**: already reported elsewhere; include reference
    - ▷ **WORKSFORME**: couldn’t reproduce issue
  - ▷ **dependencies**: which issues does this one depend on/block?



The **resolutions** can be realized in different ways in different **bugtrackers**. The ones shown here are hard-coded in bugzilla. GitHub and GitLab use a system of developer-definable labels and a set of **issue** boards which are inspired by Kanban boards to assign and move between **states** and **resolutions**.



## Chapter 11

# Project: A Web GUI for a Books Database

In this Chapter we will pull together the technologies we have learned into a simple web application project. We will do so in multiple setps. We first make a bare-bones application (see Section 11.1) and then step by step extend it with new features.

**Bricolage Programming:** With this project we want to demonstrate a common practice of modern programming: pulling together program fragments or solution ideas from various sources (e.g. the course notes or various tutorials or even answers from [stackoverflow.com](https://stackoverflow.com)) and then adapting them to the current project and fitting them together into a coherent program that works as expected.

This approach to programming is often called “bricoleur style” [Tur95] because it relies on handicraft-like tinkering with pieces of existing materials.

Contrary to what many classical programming courses still insinuate – they seem to say that you have to know everything before you can start with a project – the advent of the internet with its multitude of high-quality programming-related resources has made bricoleur-style programming effective and efficient.

Actually, bricolage is a technique that should be leaned and adopted as a tool, especially for part-time programmers as practitioners in the digital humanities tend to be.

The web application project in this Chapter is a bricolage project, only that we have all the ideas in the course notes already and we do not have to google for them on the web.

### 11.1 A Basic Web Application

We bring together all we have learned into a basic web application that allows to list all the books in a database, as well as add, edit, and delete book records.

We use our running example of the books table as a basis, and add a [web application](#) layer via the [bottle WSGI server-side scripting framework](#) in python.

We have intentionally kept the application very simple, so that it can serve as the basis of other projects. The full source is available at <https://gl.mathhub.info/MiKoMH/IWGS/blob/master/source/databases/ex/books-app.py>. The respective template files are siblings.

#### Building a full Web Application with Database Backend

▷ **Observation 11.1.1** *With the technology in fallback=the chapters on web appli-*

cations and databases we can build a full *web application* in less than

- ▷ 100 lines of python code and (back-end/routes)
- ▷ less than 70 lines of HTML *template files*. (front-end)

**Functionality:** Maintain a database of books, in particular: (e.g. your library at home)

- ▷ ▷ add a new book to the database
- ▷ delete a book from the database
- ▷ update (i.e. change) an existing book
- ▷ The source is at <https://gl.mathhub.info/MiKoMH/IWGS/blob/master/source/booksapp/ex/books-app.py>.



©: Michael Kohlhase

273



Now, if you download the file `books-app.py` and all the sibling template files `*.tpl` at <https://gl.mathhub.info/MiKoMH/IWGS/blob/master/source/booksapp/yex>, you can start the application from the shell by typing `python books-app.py`. This will yield something like

```
> python3 books-app.py
Bottle v0.12.18 server starting up (using WSGIRefServer())...
Listening on http://localhost:8080/
Hit Ctrl-C to quit.
```

So enter the url `http://localhost:8080/` into the URL bar of your browser, and test the setup.

We do the usual things to set up the web application: we load the libraries, connect to the data base, and so on.

## The Books Application: Setup

- ▷ We have already seen how to set up the database in slide 234.

```
import sqlite3
from sqlite3 import Error
from bottle import route, run, debug, template, request, get, post

# our database file
database = "books.db"
db = sqlite3.connect(database)
```

- ▷ But we want to receive result rows as dictionaries, not as tuples, so we add
- ```
db.row_factory = sqlite3.Row
```

- ▷ give ourselves a cursor to work with
- ```
cursor = db.cursor()
```

- ▷ We start the bottle server
- ```
run(host='localhost', port=8080, debug=True)
```

- ▷ And of course, we eventually commit and close the database in the end
- ```
db.commit()
db.close()
```



The next step is to create a table for the books. This is a completely standard [SQL](#) CREATE statement which we execute in the cursor we have established in setup.

### The Books Application: Backend

- ▷ we specify the database schema and create the Books table

```
bookstable = """
CREATE TABLE IF NOT EXISTS Books (
    Last varchar(128), First varchar(128),
    YOB int, YOD int, Title varchar(255), YOP int,
    Publisher varchar(128), City varchar(128)
);
"""
cursor.execute(bookstable)
```



The next step is strictly optional. But it is so annoying to have to start with an empty database when the web application first comes up. So we provide a list of seven books. But, if we are not careful, these books will be inserted into the database every time we start up the application. Recall that we did not drop the Books table in the code above.

### The Books Application: Books to Play With

- ▷ If the Books table is empty, we insert something to play with

```
booklist = [
    ('Twain', 'Mark', 1835, 1910, 'Huckleberry Finn', 1986, 'Penguin USA', 'NY'),
    ('Twain', 'Mark', 1835, 1910, 'Tom Sawyer', 1987, 'Viking', 'NY'),
    ('Cather', 'Willa', 1873, 1947, 'My Antonia', 1995, 'Library of America', 'NY'),
    ('Hemingway', 'Ernest', 1899, 1961, 'The Sun Also Rises', 1995, 'Scribner', 'NY'),
    ('Wolfe', 'Thomas', 1900, 1938, 'Look Homeward, Angel', 1995, 'Scribner', 'NY'),
    ('Faulkner', 'William', 1897, 1962, 'The Sound and the Fury', 1990, 'Random House', 'NY'),
    ('Tolkien', 'John Ronald Reuel', 1892, 1973, 'The Hobbit', 1937, 'George Allen & Unwin', 'UK')
]
# if the Books table is empty, we fill it with booklist
row = cursor.execute('SELECT * FROM Books LIMIT 1').fetchall()
if not row:
    cursor.executemany('INSERT INTO Books VALUES (?, ?, ?, ?, ?, ?, ?)', booklist)
```

**Idea:** To find out if the table is empty (surprisingly clumsy)

- ▷ we fetch a list with at most one row (LIMIT 1);
- ▷ if Books is empty, row is the empty list which evaluates to false in a conditional.



In a more complete version of the books application we would probably have used a keyword argument like `--prefill` to the program. But python command line parsing is beyond what we want to cover in IWGS.

The next thing is to create a route for the main page of the application, i.e. the page `booksapp.py` serves at `http://localhost:8080/`. We want a listing of all the books in the database in a table.

## The Books Application Routes: The Application Root

▷ We only need to add the **bottle routes** for the various sub-pages.

▷ The main page: listing the book records in the database

```
@route('/')
def books():
    query = 'SELECT rowid,Last,First,YOB,YOD,Title,YOP,Publisher,City FROM Books'
    cursor.execute(query)
    booklist = cursor.fetchall()
    return template('books',books=booklist,num=len(booklist))
```

▷ This uses the following templates: the first generates a table of books from the template file books.tpl

```
<p>There are {{num}} books in the database</p>
<table>
    % include('th.tpl')
    % for book in books : include('book.tpl',**book) end
    <tr><th><a href="/add"><button>add a book</button></a></th></tr>
</table>
```



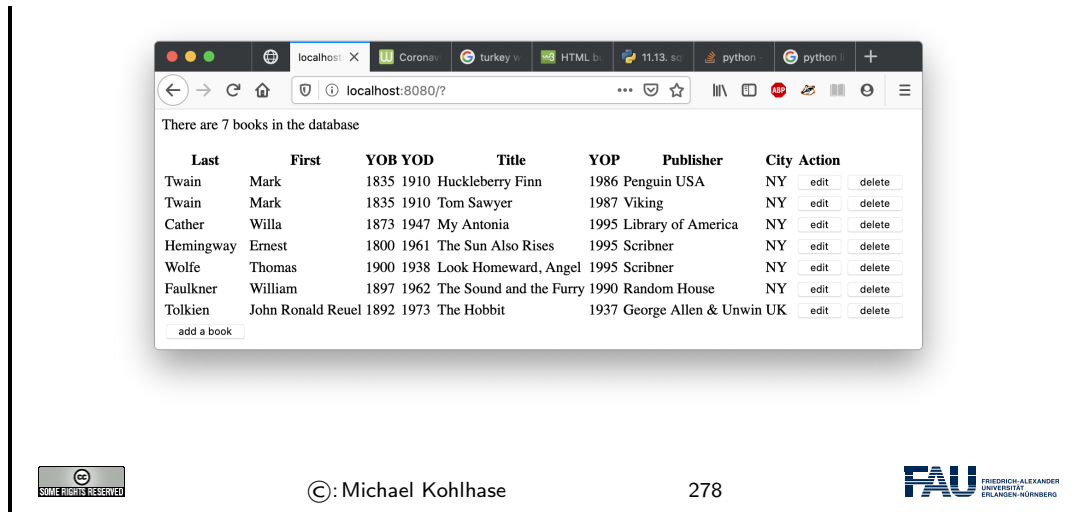
The backend of this is very simple we fire up a simple **SQL** query that selects all the records from the Books table. As we configured the database connection to return database records as **python** dictionaries, the variable `booklist` variable is a list of data dictionaries, which we can feed to the STPL template `books.tpl`, which creates the return page for `http://localhost:8080/`. This page consists of a paragraph which reports on the number of books in the database and then a table which is built up from

1. a table header which is simply imported from a template file `th.tpl`
2. a body, which is created by iterating over `booklist`, feeding each row – a **python** dictionary – to the template `book.tpl` as **keyword arguments** via the **double star operator**, and
3. a table row with a link to the `add` route for adding new books.

Before we show the nested templates, let us inspect the result:

## The Books Application Root: Result

▷ Here is the page of the books application in its initial state.



Indeed we have the report on the number of books and a table which ends in an “add a book” link. The table header and rows contain the seven data cells and two more for possible actions on the database records. The next two templates are responsible for that; they are called in the `books` template above.

### The Books Application Root: More Templates

- ▷ It inserts the table header from the template file `th.tpl`:

```
<tr>
  <th>Last</th><th>First</th><th>YOB</th><th>YOD</th>
  <th>Title</th><th>YOP</th><th>Publisher</th><th>City</th>
  <th rowspan="2">Action</th>
</tr>
```

- ▷ and iterates over the list of books, using the template file `book.tpl`:

```
<tr>
  <td>{{Last}}</td><td>{{First}}</td>
  <td>{{YOB}}</td><td>{{YOD}}</td>
  <td>{{Title}}</td><td>{{YOP}}</td>
  <td>{{Publisher}}</td><td>{{City}}</td>
  <td><a href="/edit/{{rowid}}"><button>edit</button></a></td>
  <td><a href="/delete/{{rowid}}"><button>delete</button></a></td>
</tr>
```

The first template is completely elementary, the second is called with **keyword arguments** whose values substituted for the `{{key}}` template variables. The last two columns in the table are the action links that point to the add and delete routes we present next.

The “add a book” functionality is distributed over two routes: a GET route for `/add/` and a POST route for the same path. The first is responsible for showing the input form, whereas the second parses the POST request generated by the first one and fills the database with the results. Let us look at the implementation in detail.

### The Books Application Routes: Adding Book Records

- ▷ We add a route for adding books record (for the add button)

```
@get('/add')
def add():
    return template('add')
```

Note that this is the route for the GET method on the path /add.

▷ This uses the template file add.tpl:

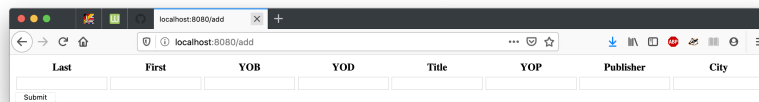
```
<form action="/add" method="post">
  <table>
    % include('th.tpl')
    <tr>
      <td><input type="text" name="Last"/></td>
      <td><input type="text" name="First"/></td>
      <td><input type="text" name="YOB"/></td>
      <td><input type="text" name="YOD"/></td>
      <td><input type="text" name="Title"/></td>
      <td><input type="text" name="YOP"/></td>
      <td><input type="text" name="Publisher"/></td>
      <td><input type="text" name="City"/></td>
    </tr>
  </table>
  <input type="submit" value="Submit"/>
</form>
```



The implementation is a rather straightforward application of a template that provides a HTML form. The only interesting thing is that we can reuse the template th.tpl from above for the table header. This not only saves effort, but also makes the user experience consistent over the various parts of the application.

## The Books Application Routes: Adding Book Records

▷ The result is



▷ The action in the HTML form is to POST to the path /add. Thus we need POST route for /add as well:

```
@post('/add')
def addResponse():
    data = parseResponse()
    ins = 'INSERT INTO Books VALUES (:Last,:First,:YOB,:YOD,:Title,:YOP,:Publisher,:City)'
    cursor.execute(ins,data)
    return template('response', data = data,
                    rowid = cursor.lastrowid,
                    text = 'New book record received')
```



The addResponse function that answers the POST route for the path /add/ just inserts a new database record in to the Books table. Note the use of the SQLite3 [parameter substitution](#) here. We substitute the parameters :«key» in the string ins with the corresponding values in the python

dictionary data which we obtain as the result of the `parseResponse` function, which we will look at next.

## The Books Application Routes: Adding Book Records

▷ this uses the function `parseResponse`, which we will reuse later.

```
def parseResponse ():
    data = {'Last': request.forms.get('Last'),
           'First': request.forms.get('First'),
           'YOB': request.forms.get('YOB'),
           'YOD': request.forms.get('YOD'),
           'Title': request.forms.get('Title'),
           'YOP': request.forms.get('YOP'),
           'Publisher': request.forms.get('Publisher'),
           'City': request.forms.get('City')}
    return data
```

▷ and the template `repnse.tpl`:

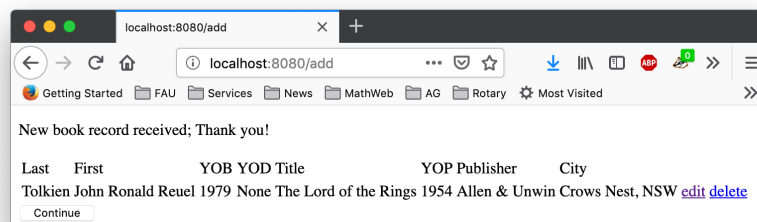
```
<form action="/">
  <p>{{text}}; Thank you!</p>
  <table>
    % include('th.tpl')
    % include('book.tpl',**data)
  </table>
  <input type="submit" value="Continue"/>
</form>
```



The `parseResponse` function is almost trivial, it just queries the response object that comes from the POST request for the various components via the `forms.get` method and packages the results in a python dictionary that feeds the `response.tpl` template. The latter creates a HTML form without input fields – we only use it to trigger a GET request to the path `/` (the application root that displays the updated book list). Note that we re-use the templates `th.tpl` and `books.tpl` from above again.

## The Books Application Routes: Adding Book Records

▷ Here is the result after filling in Tolkien's "*Lord of the Rings*":



The next relevant route is the "delete a book" functionality. Here we use another new feature: when

creating a database table in SQLite3, the system creates an additional primary key column rowid. In particular we have a rowid column in the Books table, which we make use of.

## The Books Application Routes: Deleting Book Records

- ▷ We add a route for deleting book records (for the add button)

```
@get('/delete/<id:int>')
def delete(id):
    cursor.execute('DELETE FROM Books WHERE rowid = ?',(id,))
    return template('delete')
```

Note that we have a [dynamic route](#) here: We use the [named wildcard](#) `<id:int>` to obtain the rowid of the record to be deleted.

- ▷ The template file delete.tpl does the obvious:

```
<form action="/">
  <p>Book record deleted ; Thank you!</p>
  <input type="submit" value="Continue"/>
</form>
```



Note that the link on the “delete” buttons in the books table root (see template book.tpl above) has the form `<button href="/edit/{rowid}">edit</button>`, i.e. it references the rowid column. This is picked up in the GET route for `/delete/<id:int>` path via the [named wildcard](#) `<id:int>`. This makes sure the right database record is deleted.

The routes for editing book records combine techniques from the ones for adding and deleting. From the former we use the layout into a GET and POST route, from the latter, we use the [dynamic route](#)

## The Books Application Routes: Editing Book Records

- ▷ [Idea](#): Combine techniques from the add and delete routes

```
@get('/edit/<id:int>')
def edit(id):
    cursor.execute('SELECT * FROM Books WHERE rowid = ?',(id,))
    return template('edit', cursor.fetchone(), id = id)

@post('/edit/<id:int>')
def editResponse(id):
    data = parseResponse()
    up = """UPDATE Books
        SET Last = :Last, First = :First, YOB = :YOB, YOD = :YOD,
            Title = :Title, YOP = :YOP, Publisher = :Publisher, City = :City
        WHERE rowid = :rowid"""
    data.update({'rowid': id})
    cursor.execute(up, data)
    return template('response', data = data, text = 'Updated book record')
```



In this case we have a small subtlety: the update instruction and the template edit.tpl need a rowid key/value pair. We solve this by updating the data dictionary suitably. Now we only have to give the template edit.tpl, which is rather straightforward.

## Books Application Routes: Editing Book Records (cont.)

- ▷ The template file `edit.tpl` is similar to `add.tpl` above, but pre-fills the input fields with the database record values.

```
<form action="/edit/{{id}}" method="post">
  <table>
    % include('th.tpl')
    <tr>
      <td><input type="text" name="Last" value="{{Last}}"/></td>
      <td><input type="text" name="First" value="{{First}}"/></td>
      <td><input type="text" name="YOB" value="{{YOB}}"/></td>
      <td><input type="text" name="YOD" value="{{YOD}}"/></td>
      <td><input type="text" name="Title" value="{{Title}}"/></td>
      <td><input type="text" name="YOP" value="{{YOP}}"/></td>
      <td><input type="text" name="Publisher" value="{{Publisher}}"/></td>
      <td><input type="text" name="City" value="{{City}}"/></td>
    </tr>
  </table>
  <input type="submit" value="Submit"/>
</form>
```



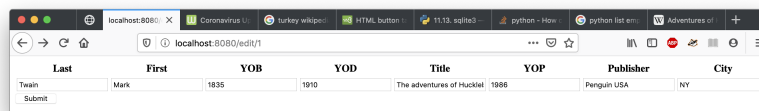
©: Michael Kohlhasse

286



## Books Application Routes: Editing Book Records (cont.)

- ▷ The result is



- ▷ Again, we use the template `response.tpl`, which we fill with a different message.



©: Michael Kohlhasse

287



The main message to take home from this experiment is that we can build a simple but complete web application with less than 100 lines of python code and less than 70 lines of HTML [template files](#).

## 11.2 Access Control and Management

Now that we have a basic web application running, we can start adding features. The most important one is [access control](#) to restrict who can access more critical functionalities of the web application, such as deleting or editing database entries.

There are many technologies for [access control](#), many use advanced features like browser [cookies](#). Here we want to introduce the simplest one: [HTTP basic authentication](#) is built into the fabric of the world wide web, das it is part of the [HTTP](#) protocol that drives it.

As [HTTP basic authentication](#) is unsafe (it sends user names and passwords over the network only lightly encoded), we also add a discussion on how to upgrade the web application to [HTTPS](#).

The full source is available at <https://gl.mathhub.info/MiKoMH/IWGS/blob/master/source/databases/ex/books-app-https.py>. The respective template files are siblings.

## Access Control and Management

- ▷ **Problem:** Anyone can write, edit, and delete records from the books database.
- ▷ **Solution:** Implement a password-based login procedure and restrict write/edit/delete access to logged-in agents.
- ▷ Let's fix some terminology before we continue
- ▷ **Definition 11.2.1** **Access control** is the selective restriction of access to a resource, **access management** describes the corresponding process.
- ▷ **Access management** usually comprises both **authentication** and **authorization**.
- ▷ **Definition 11.2.2** **Authorization** refers to a set of rules that determine who is allowed to do what.
- ▷ **For our books application** we need four things
  1. a browser interaction to query the user for username and password
  2. a way to transport them to the web application program
  3. a method for checking the username/password (**authentication**)
  4. a way the specify who can do what. (**authorization**)

**Realization:** 1./2. via **HTTP**, 4. via bottle basic auth, implement 3. directly.



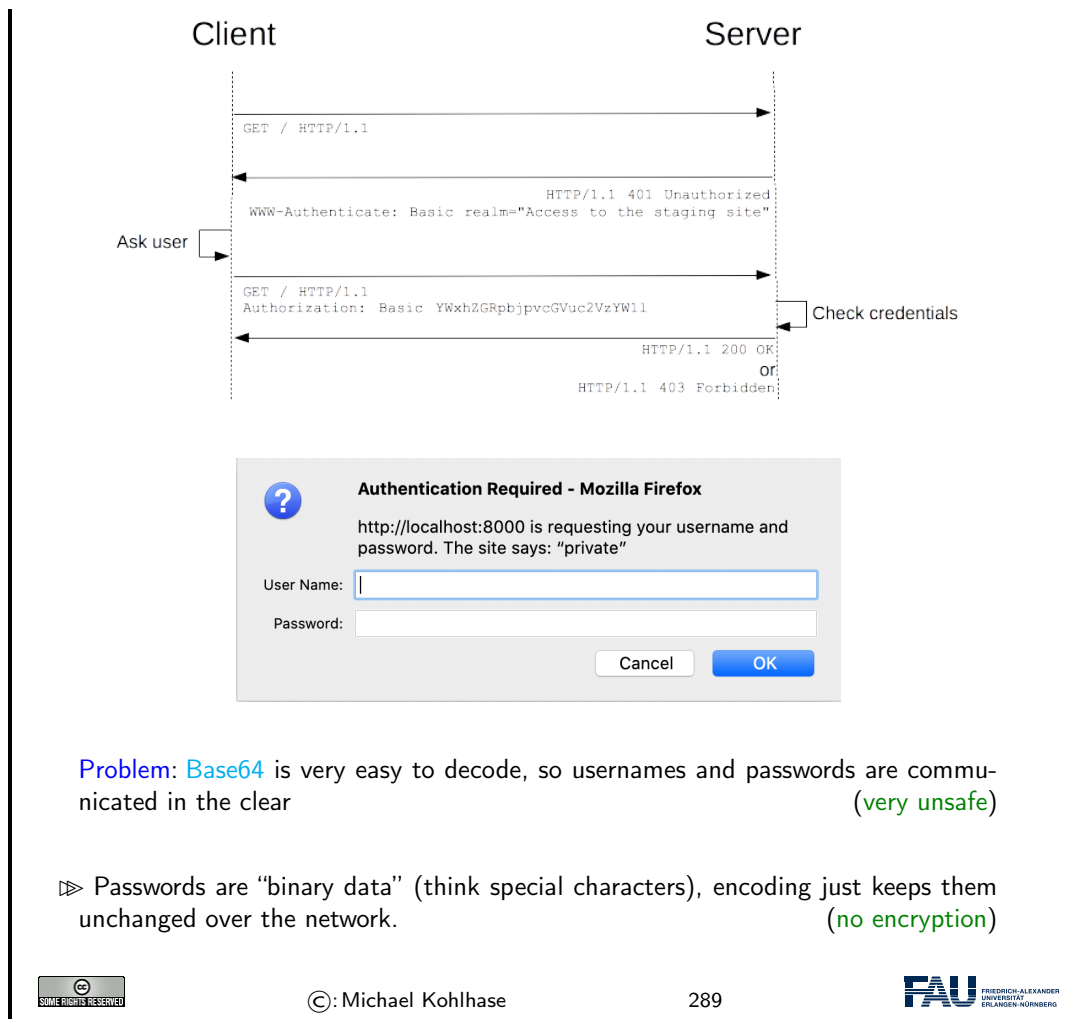
**HTTP basic authentication** is a simple mechanism in the **HTTP** protocol that standardizes the transmission of username/password information the “handshake” that leads to its acquisition.

## HTTP Basic Authentication

- ▷ **Recall** that **HTTP** is a plain-text protocol that passes around headers like this

```
GET /docs/index.html HTTP/1.1
Host: www.nowhere123.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
(blank line)
```

- ▷ **Idea:** For **authentication** extend the **HTTP** headers with support for username/-password pairs.
- ▷ **Definition 11.2.3** **HTTP basic authentication** introduces a **HTTP** header **Authorization** for **base64-encoded** pairs `⟨username⟩:⟨password⟩` and a couple of challenge/response messages.



The message sequence diagram in Definition 11.2.3 shows the basic handshake mechanism that establishes authentication and the delivery of restricted material to an authenticated user.

The diagram shows the details of the communication between client and server (symbolized by the two vertical lines). The top arrow is a normal **HTTP GET** request (without a **Authorization** field).

But – since the resource that is requested – the server does not just answer with a **HTTP** “200 OK” and the resource, the server answers with a **HTTP** “401 Unauthorized” code, which contains a description of the reason for the restriction.

When the browser receives the 401 code, it asks the user for a user name and password e.g. with a popup form like the one shown in Definition 11.2.3, possibly displaying the reason string<sup>4</sup>. This information is then sent to the server in a second **GET** request, this time with the user-name/password information in the **Authorization** request.

The server checks the user/password data and – depending on the result of the check – sends a **HTTP** response “200 OK” together with the resource or a “403 Forbidden” (without the resource).

One thing that we have not discussed here is that most browsers store the username/password information and supply it to the server – often directly in any outgoing requests – which makes it hard to test authentication and unauthenticated behavior in web application development. A useful trick here is – if you are logged into `http://example.org` – to address a **GET** request to `http://abc@example.org`. Background: **HTTP basic authentication** allows you to set user/-

<sup>4</sup>EdNOTE: MK: the reason string does not fit

password information directly by prepending `⟨user⟩:⟨pass⟩` to the [authority](#) of the [URI](#) used in a [HTTP](#) request.

Of course, [HTTP basic authentication](#) is supported by the bottle WSGI framework.

### Basic Auth in Bottle

- ▷ **Idea:** Support the server side of [HTTP basic authentication](#) in bottle web-apps.
- ▷ **Implementation:** New decorator `@auth_basic(⟨function⟩)` to mark a route as password-protected.
- ▷ **Usage:** Decorate every route we want to restrict access of with `@auth_basic(⟨function⟩)`, where `⟨function⟩` is a function that takes two string arguments (user name and password) and returns a Boolean for the authorization decision.



©: Michael Kohlhasse

290



What happens behind the scenes here is clear from the authentication handshake explained in Definition 11.2.3

### Basic Auth in Bottle: Minimal Viable Example

- ▷ **Example 11.2.4** A web application with restricted route.

```
from bottle import run, get, auth_basic

def check(user, password):
    return user == "miko" and password == "test"

@get("/")
@auth_basic(check)
def protected():
    return "authorized access granted!"

run(host="localhost", port=8000)
```

**Idea:** Mix restricted and open routes in a partially restricted application

- ▷ **Extension:** Use different check functions for different levels of restriction (user roles)



©: Michael Kohlhasse

291



This was easy enough. But one problem remains: in [HTTP basic authentication](#), user names and passwords are not confidential when they are transported over the network. The simplest way to ensure confidentiality is to layer encryption on top of [HTTP](#), which is just what the [HTTPS](#) protocol does.

### HTTPS: HTTP over TLS

- ▷ **Definition 11.2.5** [Hypertext Transfer Protocol Secure \(HTTPS\)](#) is an extension of the [Hypertext Transfer Protocol \(HTTP\)](#) for secure communication over

a computer network. **HTTPS** achieves this by running **HTTP** over a **TLS** connection.

- ▷ **Consequences for Web Applications:** We can use **HTTP** as usual, except
  - ▷ we gain communication privacy and server authentication
  - ▷ server and browser need to speak **HTTPS** (most do)
  - ▷ the server needs a **public key certificate** and a **private key**.
- ▷ in bottle, we can just swap out the **HTTP** server to one that can do **HTTPS**:
 

```
run(host='localhost',port='8888',server='gunicorn',keyfile='key.pem',certfile='cert.pem')
```

install it first with `pip install gunicorn`.
- ▷ **Problem:** Where to get the certificate file `cert.pem` and private key `key.pem`?
- ▷ **One Solution:** Self-sign one, e.g. using <https://www.selfsignedcertificate.com/> (adapt file names)
- ▷ **Remaining Problem:** Your browser force you to log an exception for `https://localhost:8888` (probably OK for development)



Self-signed **TLS** certificate are sufficient for web application development. But publically deploying a **HTTPS**-based web application we need real ones. Fortunately, there is a relatively simple way of obtaining them.

## Getting a Real TLS Certificate via Let's-Encrypt

- ▷ **HTTPS** is the new "regular **HTTP**" on the web
- ▷ **Observation 11.2.6** A self-signed certificate gives you communication privacy but not authentication  $\Leftarrow$  only you yourself vouch for the authenticity of the web site.
- ▷ **Definition 11.2.7** In a public key infrastructure, the TLS certificate is issued by a **certificate authority**, an organization chartered to verify identity and issue TLS certificates.
- ▷ **Certificate authorities** sell trust (for a lot of money)  
They certify e.g. that the `https://bmw.com` is under control of BMW AG.
- ▷ **Idea:** Finding out that you have control over a particular web site on the web can be automated, if you run a program on the server host.
- ▷ **Definition 11.2.8** **Let's Encrypt** is a not-for-profit **certificate authority** that does this and issues free TLS certificates. (to encourage **HTTPS** adoption)
- ▷ **Concretely:** on a linux server you need two steps
  1. install certbot (usually via your package manager)

2. then `sudo /usr/local/bin/certbot certonly --standalone` will generate certs.

Details at <https://letsencrypt.org>.

▷ **Success:**  $\geq 1.000.000.000$  TLS certificates, 200.000.000 sites since 2016



©: Michael Kohlhasse

293



We have only covered the basic ideas behind certificate authorities and [Let's Encrypt](#) here, but this should enable you to figure out the rest from the [Let's Encrypt](#) web site.

## 11.3 Deploying the Books Application as a Program

Now we address the fact that a web application is usually deployed on a unix server, by sysadmins who are accustomed the unix way of handling – configuring, starting, etc. – applications. We will first introduce a way to make python scripts as [shell](#) commands and give them arguments – optional and mandatory ones.

### Deploying The Books Application as a Program

- ▷ Having a python script `booksapp.py` you start with `python3 booksapp.py` is sufficient for development
- ▷ If you want to deploy it on a web server, you want more: The sysadmin you deliver your web application to wants to start – and manage – it like any other UNIX command.
- ▷ After all, your web server will be a UNIX (e.g. `linux`) computer
- ▷ In particular behavioral variants should be available via command line options.
- ▷ **Example 11.3.1** To run the books application without output (`-q` or `--quiet`) and initialized with the seven book records we want to run

```
booksapp -q --initbooks
```



©: Michael Kohlhasse

294



### Deploying The Books Application as a Program

- ▷ **Example 11.3.2** If we forget the options, we need help:

```
> booksapp --help
```

```
Usage: <yourscript> [options]
```

```
Options:
```

```
  -h, --help show this help message and exit
```

```
  -q, --quiet don't print status messages to stdout
```

```
  -l FILE, --log=FILE write log reports to FILE
```

```
  --initbooks initialize with seven book records
```

Im



## Deploying a python Script as a Shell Command/Executable

- ▷ We can make our a python script behave like a native **shell** command.
- ▷ The file extension .py is only used by convention, we can leave it out and simply call the file booksapp.

- ▷ Then we can add a special python comment in the first **line**

```
#!/usr/bin/python3
```

which the **shell** interprets as “call the program python3 on me”.

- ▷ Finally, we make the file hello executable, i.e. tell the **shell** the file should behave like a shell command by issuing

```
chmod u+x booksapp
```

in the directory where the file booksapp is stored.

- ▷ We add the **line**

```
export PATH="./:${PATH}"
```

to the file .bashrc. This tells the **shell** where to look for programs (here the respective current directory called .)



## Working with Options in python

- ▷ We have the optparse library for dealing with command line options (install with **pip3**)

- ▷ **Example 11.3.3 (Options in the Books Application)**

```
from optparse import OptionParser
parser = OptionParser()
parser.add_option("-l", "--log", dest="logfile",
                  help="write logs to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")
parser.add_option('--version', dest="version", default=1.0, type="float",
                  help="the version of the books application")

options, args = parser.parse_args()
# do something with the options and their args.
print ('VERSION :', options.version)
```





## Chapter 12

# Image Processing

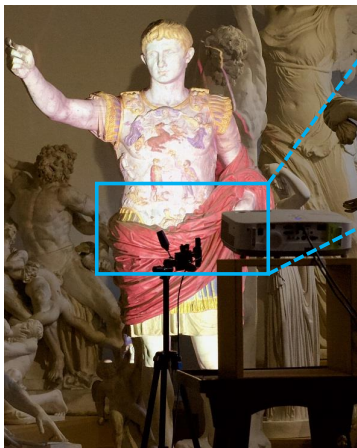
We will now begin a new topic on our way to a useful image database. In particular we will see how computer scientists think about images, how images are represented in computer memory and what we can do with them.

### Images

- ▷ **Example 12.0.1 (Zooming in on Augustus)** An image taken by a standard DSLR camera. Let's zoom in on it!

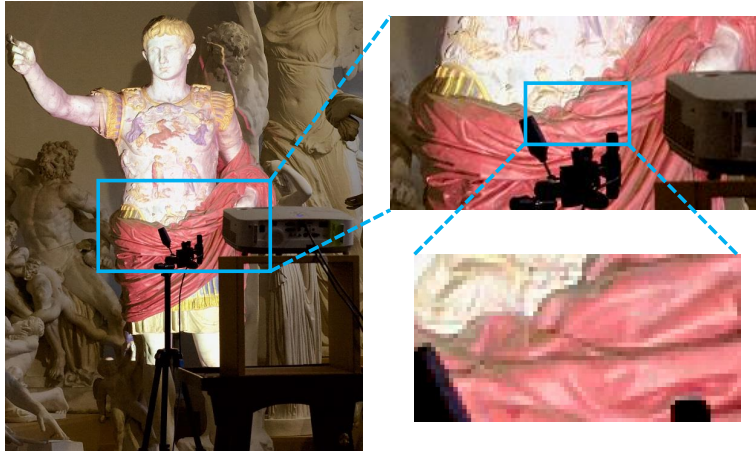


And a bit more



When zooming in on an image, we start to see blocks of colors, which are organized in a

regular grid.



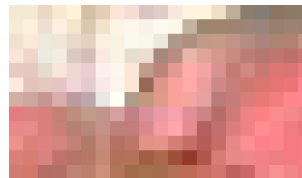
©: Michael Kohlhasse

298



## Images as Rasters of Pixels

- ▷ If we zoom in quite a bit more, we see
- ▷ **Observation:** The colors are arranged in a two-dimensional grid (raster).



- ▷ **Definition 12.0.2** We call the grid **raster** and each entry in it **pixel** (from “picture element”).

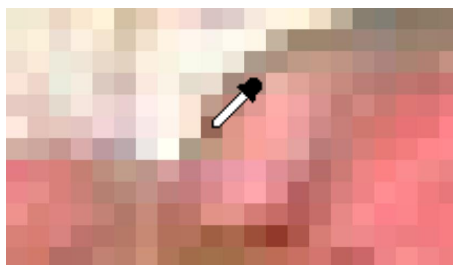


©: Michael Kohlhasse

299

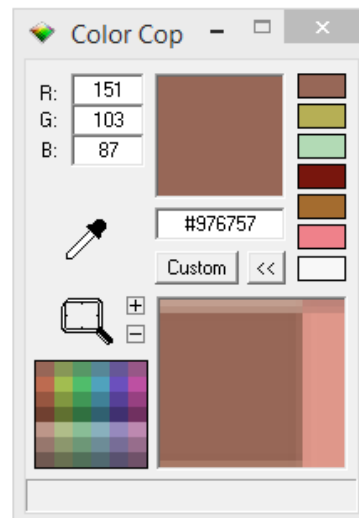


## Colors



- ▷ Colors are usually stored in  $\langle R, G, B \rangle$  format. (3 channels)  
 $R, G, B \in [0, 255] \leadsto$  One Byte per channel per pixel.

- ▷ Images in this format can store  $256 \cdot 256 \cdot 256 = 256^3$  (about 16 million) colors.





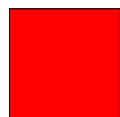
Each pixel stores color information. We can obtain the values stored in images using a color picker. Image processing programs like Microsoft Paint or Adobe Photoshop provide color pickers (pipettes), but there also exist standalone applications. In this example we are using Color Cop <sup>1</sup>.

According to the color picker, our pixel stores the value (151, 103, 87). Colors are organized in the so-called RGB format, meaning a color is composed from a mixture of red (R), green (G) and blue (B). We call these components **channels** or **bands**.

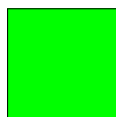
The value in each of these channels typically ranges from 0 to 255. This is because a single Byte can store exactly this value range and a Byte was deemed enough for most applications. We can deduce that a pixel has  $256 \times 256 \times 256$  distinct value combinations, which is just over 16 million colors an image in this format can display. You might have seen this number on product descriptions of computer monitors or cameras.

## Color Examples

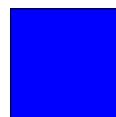
▷ **Example 12.0.3** A color can be represented by three numbers.



(255, 0, 0)  
Red



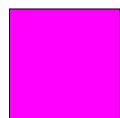
(0, 255, 0)  
Green



(0, 0, 255)  
Blue



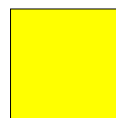
(255, 255, 255)  
White



(255, 0, 255)  
Magenta



(0, 255, 255)  
Cyan



(255, 255, 0)  
Yellow



(128, 128, 128)  
Gray

▷ **Definition 12.0.4** A color is called **grayscale**, iff  $R = G = B$



A channel value of 0 means no intensity in this channel, a value of 255 corresponds to full intensity. Thus, in order to create a pure red we set the R channel to 255 and the other two to 0 (no green or blue). Other colors are achieved in a similar fashion.

Secondary colors (magenta, cyan, yellow) are created by mixtures of red, green, and blue. For example, we create magenta by mixing red and blue.

Different shades of gray are obtained, when  $R=G=B$ . White is the brightest gray we can achieve, by setting all values to 255. Black on the other hand has all channels set to 0 (meaning no light/intensity).

When processing colors it is often beneficial to think about **normalized colors**. We normalize colors by dividing by 255 (the highest value). Resulting color values are now between 0 and 1.

## Normalized Color Values

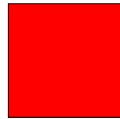
<sup>1</sup><http://colorcop.net/>

▷ Rather than thinking of a pixel value of being between 0 and 255, it is beneficial to think in terms of normalized color values, between 0 and 1.

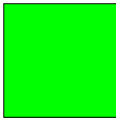
▷ Values are still stored as Bytes, but normalized before use:  $v' = v / 255$

▷

▷ **Example 12.0.5**



(1, 0, 0)  
Red



(0, 1, 0)  
Green



(0, 0, 1)  
Blue



(1, 1, 1)  
White



(1, 0, 1)  
Magenta



(0, 1, 1)  
Cyan



(1, 1, 0)  
Yellow



(0.5, 0.5, 0.5)  
Gray



©: Michael Kohlhase

302

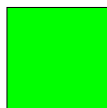


## HTML Color Codes

Shorthand notation for colors.  
Encode (R,G,B) as hexadecimal numbers.



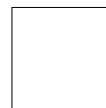
#FF0000  
Red



#00FF00  
Green



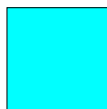
#0000FF  
Blue



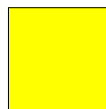
#FFFFFF  
White



#FF00FF  
Magenta



#00FFFF  
Cyan



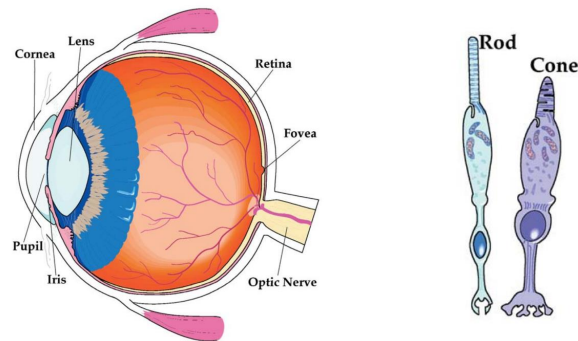
#FFFF00  
Yellow



#808080  
Gray

Recall from last semester: In HTML and CSS we often express colors in HTML color codes. This is the same principle as before, however the values are not expressed in decimal numbers but instead in hexadecimal.

## The Human Eye



M. D. Fairchild, *Color appearance models*, 2nd ed. Chichester, West Sussex, England; Hoboken, NJ: J. Wiley, 2005.

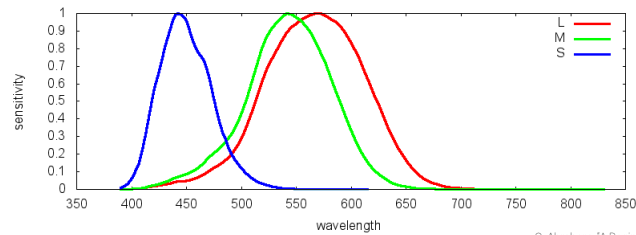
The human eye has **cones and rods**, which are responsible for **color and brightness** vision, respectively.

Slide 304

Quick detour into the real world: Let's explore where the RGB format comes from.

Light from our surroundings enters our eye through the lens and then hits the retina on the back of our eye. On the retina sit rods and cones, which are responsible for brightness and color vision, respectively. Since we are interested in colors here, we will ignore the rods for the purpose of this lecture.

### The Human Eye – Three Types of Cones



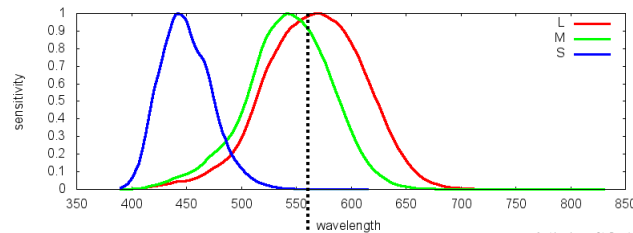
C. Abraham, 'A Beginner's Guide to (CIE) Colorimetry', *Hipster Color Science* 10-Sep-2016. Available: <https://medium.com/hipster-color-science/a-beginners-guide-to-colorimetry-401f1830b65a> [Accessed: 13-May-2019].

Slide 305

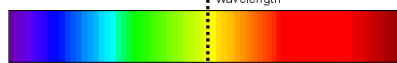
Light is an electromagnetic radiation. Only a small part of this radiation is visible to the human visual system (wavelengths around 380 to 740 nanometers).

There are three types of cones, which react to different areas in this spectrum. They roughly correspond to the wavelengths, which we perceive as red, green, and blue (or rather long, middle, and short wavelengths).

## The Human Eye – Three Types of Cones



C. Abraham, 'A Beginner's Guide to (CIE) Colorimetry', *Hipster Color Science* 10-Sep-2016. Available: <https://medium.com/hipster-color-science/a-beginners-guide-to-colorimetry-401f1830b65a> [Accessed: 13-May-2019].



Example: Yellow  
Both "red" and "green" cone are stimulated.

Eye cannot distinguish between yellow light and **mixture** of red and green! (both look yellow)

Slide 306

When we now see yellow light for example, the two cones responsible for long and medium length wavelengths are stimulated. Our brain converts this stimulus to yellow.

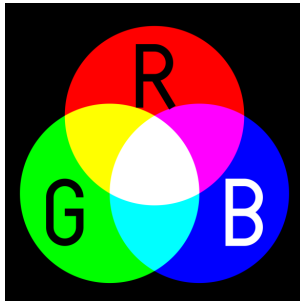
However, let's imagine we perceive a mixture from red and green light. In this case these two cones will be stimulated, too! Our brain is incapable of distinguishing between these two scenarios, since the physical stimulus on our eye is the exact same!

It turns out that we can create all colors as a mixture of red, green, and blue light.

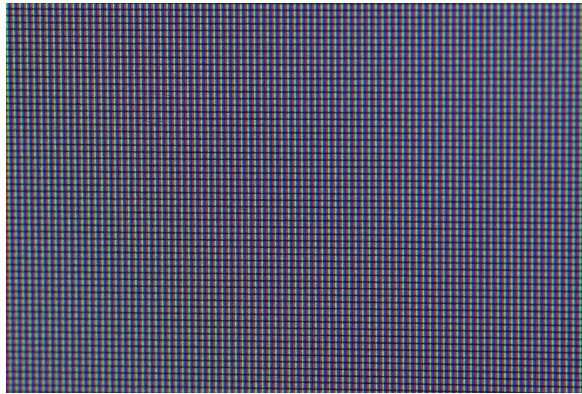
## Monitors

Monitors have pixels, too!

One **pixel** = red, green, blue **subpixel**!  
If the subpixels are small enough, it  
looks like a single color!



SharkD, "Additive color mixing", 2006 Available:  
<https://commons.wikimedia.org/wiki/File:AdditiveColor.svg> [Accessed 06-June-2019]



Devore, TFT Bildschirm RGB Pixel, 2012 Available:  
[https://commons.wikimedia.org/wiki/File:TFT\\_Bildschirm\\_RGB\\_Pixel.JPG](https://commons.wikimedia.org/wiki/File:TFT_Bildschirm_RGB_Pixel.JPG) [Accessed 06-June-2019]

Slide 307

Monitors take advantage of this, since they usually also have pixels. These pixels typically consist not of a single light source, but three distinct **subpixels**. If these subpixels are small enough and close together, our eye cannot see that the light actually comes from different points and thus perceives the mixture color.

End of detour!

## Image Size



Image: 1440 x 746 pixels

Image File Size Expectation:

**Width x Height x Channels:**  $1440 \times 746 \times 3 = 3,222,720$  Bytes  $\approx 3$  MB

However:

Augustus.jpg	4/30/2019 2:58 PM	JPEG image	404 KB
Augustus.png	6/3/2019 12:19 PM	PNG image	1,628 KB

On disk images are usually compressed (jpeg, png, gif, etc).  
Jpeg file size is smaller than png, but image quality is lost.

Slide 308

Take our Augustus image again. It is 1440 pixels wide and 746 pixels high. Since each pixel stores three channels, which each measure one Byte, we can calculate the image size:  $1440 \times 746 \times 3 = 3222720$  Bytes. On disk however images are usually smaller.

This is because images on disc are usually compressed and stored in a format like **.jpg** or **.png**. Be careful with JPEG compression! JPEG sacrifices image quality in order to achieve smaller file sizes!

## Jpeg Compression Artifacts

Here, the Augustus image is saved with a very **high jpeg compression**. The file size is tiny (27 KB, compare to 440 KB on previous slide). However, the **image quality suffers**.

Jpeg creates blocks of pixels, and approximates the colors in this block with as few bits as possible (according to compression ratio).



Slide 309

In this example we turned the JPEG compression very high, which leads to a tiny file size but strong artefacts in the image quality.

## Pillow

<https://pillow.readthedocs.io/en/stable/>

Install: `pip install Pillow`

We will use Pillow in IWGS.

Pillow is a fork (a version) of the old Python module PIL (Python Image Library).

```
from PIL import Image

# load image
im = Image.open('image.jpg')
im.show()

# access color at pixel (x, y)
x = 15
y = 300
r, g, b = im.getpixel((x, y))
```

When processing images in code, we have to load them from disc and then perform operations on them. In IWGS we will use **Pillow** for this task. The example shows how images are loaded from disc.

Loading here means that the file is read, and that the compression is reversed, i.e. the image is decompressed. This means that the image which was before stored in JPEG compression is now present in main memory (RAM). You can think about the loaded image as a long Python list of pixel values, i.e. one pixel after the other.

## Pillow

<https://pillow.readthedocs.io/en/stable/>

Install: `pip install Pillow`

We will use Pillow in IWGS.

Pillow is a fork (a version) of the old Python module PIL (Python Image Library).

```
from PIL import Image

# load image
im = Image.open('image.jpg')
im.show()

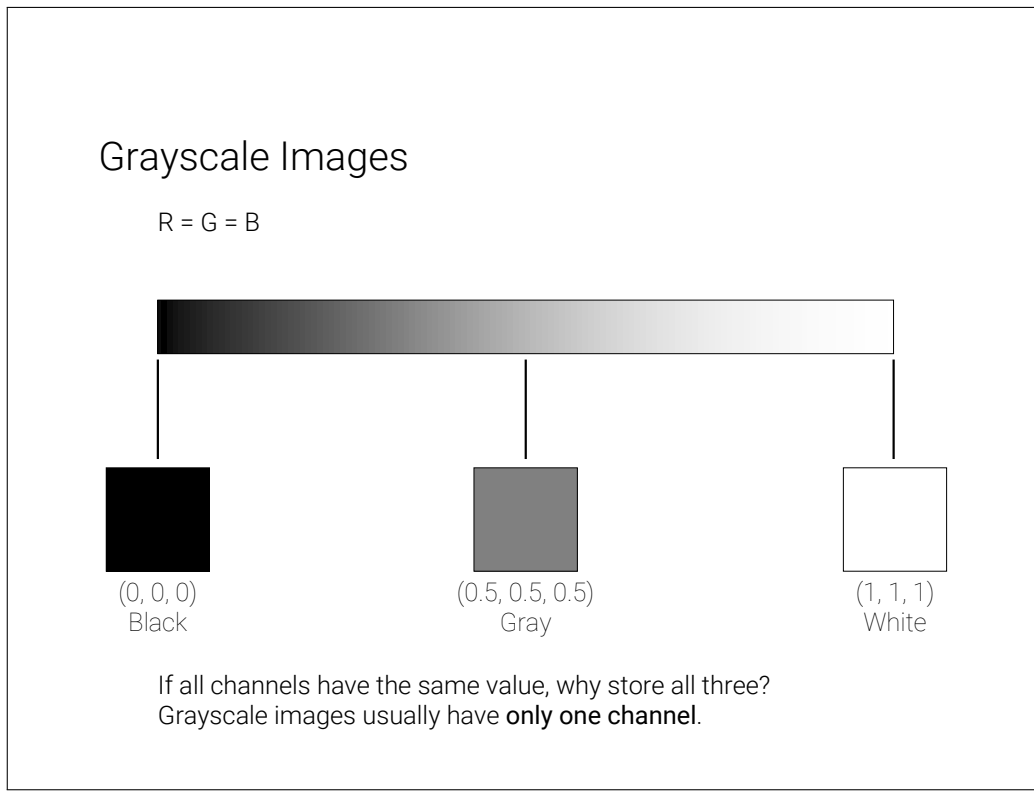
# access color at pixel (x, y)
x = 15
y = 300
r, g, b = im.getpixel((x, y))
```

```
from PIL import Image

# load image
im = Image.open('image.jpg')
im
```

↖  
In Jupyter

In Jupyter Notebooks we instead simply write the variable name, which then shows the image in a new cell.



Slide 312

We said before that in colors, which represent shades of gray, all channels have the same value. If this is true for all colors in an image, we call them **grayscale images**.

Since it is pointless to store each value three times, grayscale images usually only store one value per pixel, which is then tripled before display.

## Color to Grayscale Conversion



$$\text{Gray} = 0.21 \times \text{R} + 0.71 \times \text{G} + 0.08 \times \text{B}$$

Humans are very sensitive to green.  
Green is therefore weighted higher than red and blue.

Slide 313

Conversion from color to grayscale images is a common operation, which most image processing tools (Photoshop etc.) support. It serves as a first example of what we can do with images.

Grayscale conversion is a *weighted sum* of the three channel values. This means, each channel value is multiplied with a factor and then the values are added to form a single value. Since humans are very sensitive to green, the G channel has the highest weight.

### Some more Image Operations



Original



Grayscale



Sepia



Inverse

Each pixel is  
processed separately!



Threshold

Red Channel  
Extraction

Slide 314

Displayed here are some more image operations. All of these process each pixel separately. Implementation of these operations is very simple in Python. Since we store all our pixels in a large list, we can simply create a for-loop over this list, do our calculation and store the result in a new image at the same pixel coordinate.

## Image Operations in Pillow

```
from PIL import Image, ImageOps

im = Image.open ('image.jpg')

# convert to grayscale
gray = ImageOps.grayscale(im)

# invert image
inverse = ImageOps.invert(im)
```

Complete List:

<https://pillow.readthedocs.io/en/stable/reference/ImageOps.html>

Slide 315

**Pillow** supports many image operations. This slide displays two examples. Refer to the documentation for a complete list.

## Transparency

Sometimes we want to overlay images -> **Layers**  
We need a notion of how transparent a pixel is.

We introduce a **fourth channel**: A (for alpha).  
Alpha is the **Opacity** (inverse of transparency).  
A pixel is now (R,G,B,A).

Order of layers is important here! The Augustus image is **below** the other image!  
The Augustus image has NO transparency, the second image does!




Transparency is an important operation. In this example we want to layer two images on top of each other. We thus need to store for each pixel a measure of how transparent it is.


We expand our RGB notion to RGBA, by introducing a fourth channel A. A stands for alpha and corresponds to the opacity of a pixel, i.e. a value of 0 means zero opacity (fully transparent), a value of 1 (normalized) means fully opaque (no transparency).

### Transparency


$(R,G,B,A) = (0, 0, 0, 0)$   
Full transparent






$(R,G,B,A) = (0.6, 0.0, 1.0, 0.5)$   
Half transparent purple



$(R,G,B,A) = (1, 1, 0, 1)$   
Full yellow




+

=


$$R_{\text{target}} = (1-A) \times R_{\text{augustus}} + A \times R_{\text{purple,yellow}}$$

$$G_{\text{target}} = (1-A) \times G_{\text{augustus}} + A \times G_{\text{purple,yellow}}$$

$$B_{\text{target}} = (1-A) \times B_{\text{augustus}} + A \times B_{\text{purple,yellow}}$$

See examples for the opacity here. Fully transparent regions (visualized by the checkerboard), have an alpha value of 0. Fully opaque regions have a value of 1. Intermediate values are possible which correspond to partial transparency.

The final image is then composed by deciding for each pixel how much color from each source image should contribute.

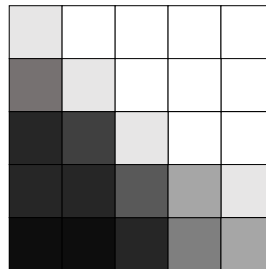
Note that this is again a per-pixel operation, which can easily be implemented with a simple for-loop.

## Edge Detection

Goal: Find interesting parts of image (features).

Example: Find **edges**, i.e. sections, where **color changes rapidly**.

Example Image:



Slide 318

We will now look at more interesting image operations. A typical example especially important for object recognition in images is to find **features**. Features are areas in the image, which are recognizable.

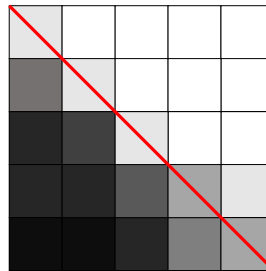
For example, let's say we want to find so-called **edges** in our image, i.e. areas where the color changes rapidly. Edges often correspond to object outlines. We will see an example later.

## Edge Detection

Goal: Find interesting parts of image (features).

Example: Find **edges**, i.e. sections, where **color changes rapidly**.

Example Image:



Clearly there is an edge in this image.  
How do we detect it automatically?

Slide 319

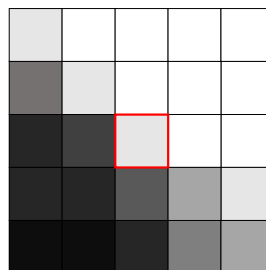
In this (admittedly simple) example image, we can clearly see, that there is an edge present, where the color shifts fast from dark to light. We will now explore, how we can detect such an edge automatically.

## Edge Detection

Goal: Find interesting parts of image (features).

Example: Find **edges**, i.e. sections, where **color changes rapidly**.

Example Image:



Decide for each pixel, if it is an edge.  
Here: Is marked pixel an edge pixel?

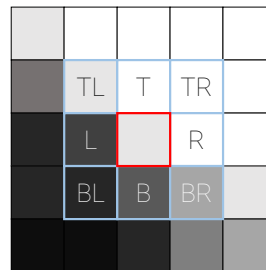
The idea is to decide for each pixel if it is part of an edge or not (binary decision, yes or no). Let's take the marked pixel as example, but remember that the following operations are performed on each pixel in the image.

## Edge Detection

Goal: Find interesting parts of image (features).

Example: Find **edges**, i.e. sections, where **color changes rapidly**.

Example Image:



T = Top  
B = Bottom  
L = Left  
R = Right

Inspect neighbor pixels.

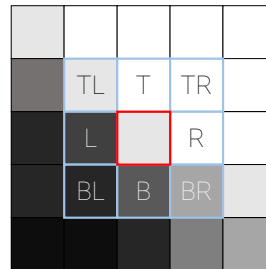
Let's consider the neighbors of our marked pixel.

## Edge Detection

Goal: Find interesting parts of image (features).

Example: Find **edges**, i.e. sections, where **color changes rapidly**.

Example Image:



T = Top  
B = Bottom  
L = Left  
R = Right

Horizontal edge, if:

$$[I_B - I_T] + [I_{BL} - I_{TL}] + [I_{BR} - I_{TR}] > \text{Threshold}$$

Vertical edge, if:

$$[I_R - I_L] + [I_{TR} - I_{TL}] + [I_{BR} - I_{BL}] > \text{Threshold}$$

Slide 322

The idea for this edge detection algorithm is to compare the pixel column left to our marked pixel to the column to the right. If the difference between the two columns is large, we know that we are observing a vertical edge.

Analogous we can do the same for horizontal edges, by comparing the row above to the row below our marked pixel.

We could perform this operation using only the pixels marked by L, R, B, and T, so only the direct neighbors. By taking the diagonal pixels into consideration, too, we make sure we only detect larger features.

## Edge Detection

Usually the center row or column is more important and is thus higher weighted.

Algorithm: Get pixel value of each neighbor in 3x3 window, multiply with following weights and add everything up.

Horizontal edge test:

	-1	-2	-1	
	0	0	0	
	1	2	1	

Vertical edge test:

	-1	0	1	
	-2	0	2	
	-1	0	1	

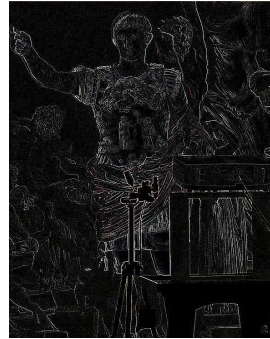
Slide 323

The operation we described here is called **Sobel filter**<sup>2</sup>, named after Irwin Sobel.

Usually the direct neighbors are deemed more important than the diagonal neighbors. The pixel values of the neighbor pixels are thus weighted, such that the direct neighbors contribute more.

<sup>2</sup>[https://en.wikipedia.org/wiki/Sobel\\_operator](https://en.wikipedia.org/wiki/Sobel_operator)

## Edge Detection



```
from PIL import Image, ImageFilter

im = Image.open('augustus.jpg')
edges = im.filter(ImageFilter.FIND_EDGES)

edges.show() # or just edges in Jupyter
```

Slide 324

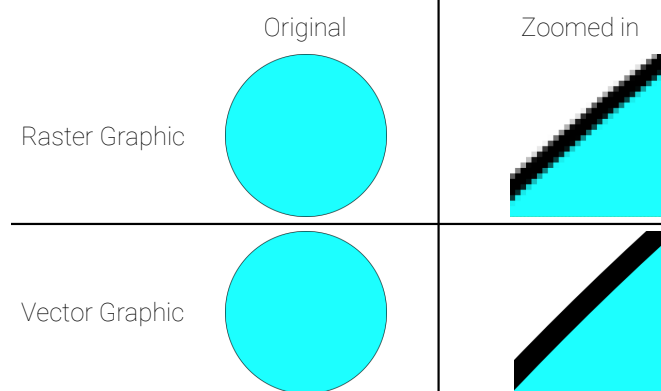
Here we see an example of edge detection. White pixels in the right image are pixels, which were classified as edge pixels, i.e. pixels where large changes in color are present. Black pixels are no edges.

**Pillow** provides this operation as showcased in the code example.

## Vector Graphics

Raster Graphics store colors in pixel grid.  
Quality deteriorates when image is zoomed into.

Vector Graphics solve this problem!



Slide 325

The images we talked about so far store colors in a large grid of pixels (a raster). A common problem with these types of images is that we cannot zoom in on them as far as we want, without losing quality. At a certain point we start to see the individual pixels.

Vector graphics are an alternative way of storing image data, which solve this problem.

## Vector Graphics

Instead of individual pixels, vector graphics store **shape information**.

Example: For circle, just store

- center
- radius
- line width
- line color
- fill color

For line, store

- start and end point
- line width
- line color

For display, vector graphics usually have to be **rasterized**  
(monitors only support raster graphics)!

Slide 326

The idea of vector graphics is fundamentally different than the idea of raster graphics. Instead of storing pixels, we now store shape information!

For example, for a circle we don't store a color for each pixel, but we rather just store where the circle is, along with its radius, color, etc.

## Vector Graphics Display

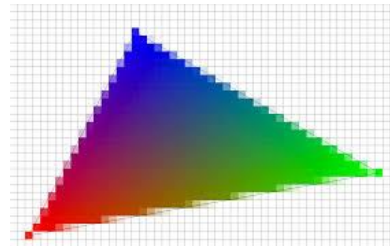
There exist monitors to directly display vector graphics.

*Autopilot, A free software Asteroids-like video game played on an oscilloscope configured in X-Y mode. 2013.  
Available: [https://commons.wikimedia.org/wiki/File:Space\\_Rocks\\_\(game\).jpg](https://commons.wikimedia.org/wiki/File:Space_Rocks_(game).jpg)  
[Accessed: 06-June-2019].*



However, with common displays, vector graphics are **rasterized** before display.

*John P. Hess, "Rasterization - The Most Basic Rendering Technique,"  
FilmmakerIQ.com 07-Apr-2017. Available:  
<https://filmmakeriq.com/lessons/rasterization/>  
[Accessed: 06-June-2019].*



Slide 327

Note that most monitors cannot display vector graphics. There are vector monitors, but they are not common.

The monitor displayed here does not have pixels. It instead moves a laser and traces a polygon (the asteroids and spaceship). The laser stimulates a phosphor layer, which then glows.

Common monitors work with pixels. Vector graphics are thus **rasterized** (i.e. turned into raster graphics) just before being displayed. The rasterizer decides for each pixel, whether it is inside or outside the shape.

## SVG

Scalable Vector Graphics.  
SVG is one type of vector graphics.  
XML-based!

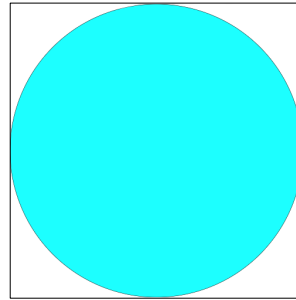
Example for circle:

```
<svg width="100" height="100" xmlns="http://www.w3.org/2000/svg">
  <circle cx="50" cy="50" r="50" style="fill:#1cffff; stroke:#000000; stroke-width:0.1" />
</svg>
```

**<svg>** tag starts document.  
**width, height** declares size.

**<circle>** starts circle.  
**cx, cy** is the center point.  
**r** is the radius.  
**style** describes how the circle looks.

Since the SVG size is 100x100 and the circle is at (50,50) with radius 50,  
it is centered and fills the whole region.



Slide 328

SVG is one image format for vector graphics. Since it is XML-based we are able to read it. As described above, we can create circles by specifying a position, radius, and style (color etc).

## More SVG Primitives

Rectangle:

```
<rect x="..." y="..." width="..." height="..." style="..." />
```

Ellipse:

```
<ellipse cx="..." cy="..." rx="..." ry="..." style="..." />
```

Line:

```
<line x1="..." y1="..." x2="..." y2="..." style="..." />
```

Text:

```
<text x="..." y="..." style="...">This is my text!</text>
```

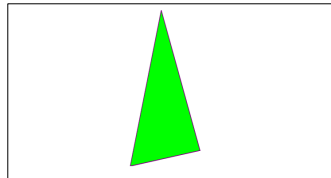
Images:

```
<image xlink:href="..." x="..." y="..." width="..." height="..." />
```

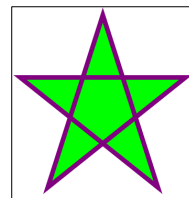
Here are some examples of SVG primitives.

## SVG Polygons

```
<svg height="210" width="500" xmlns="http://www.w3.org/2000/svg">  
  <polygon points="200,10 250,190 160,210" style="fill:lime;stroke:purple;stroke-width:1" />  
</svg>
```



```
<svg height="210" width="210" xmlns="http://www.w3.org/2000/svg">  
  <polygon points="100,10 40,198 190,78 10,78 160,198" style="fill:lime;stroke:purple;stroke-width:5;fill-rule:nonzero;" />  
</svg>
```



We can draw arbitrary polygons by specifying a list of coordinates.

## SVG in HTML

SVG can be used in dedicated files (.svg file ending).  
It can however also be written **directly in HTML** files.

Triangle from last slide embedded in HTML file:

```
<html>
<body>

<svg height="210" width="500" xmlns="http://www.w3.org/2000/svg">
  <polygon points="200,10 250,190 160,210" style="fill:lime;stroke:purple;stroke-width:1" />
</svg>

</body>
</html>
```

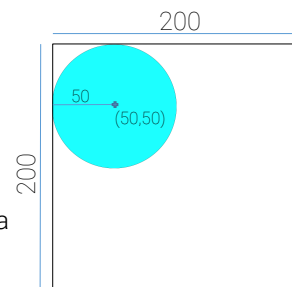
Slide 331

SVG can directly be embedded in HTML!

## The SVG **viewBox** Attribute

```
<svg width="200" height="200" xmlns="...">
  <circle cx="50" cy="50" r="50" style="..." />
</svg>
```

In this example the width and height are scaled by a factor of 2 to give us a little more room.  
Sometimes we want to specify a larger image, but only display a section of it.

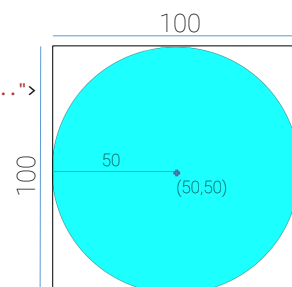


Introducing **viewBox**:

```
<svg viewBox="0 0 100 100" width="200" height="200" xmlns="...">
  <circle cx="50" cy="50" r="50" style="..." />
</svg>
```

**viewBox** specifies a region inside our canvas. Only things inside this region are drawn. The resulting image is then stretched to the canvas size (**zoom effect**).

<https://www.sarasoueidan.com/blog/svg-coordinate-systems/>



Slide 332

We now explore a useful attribute of SVG called `viewBox`. We said that we can zoom in onto vector graphics as far as we want without losing quality, so let's give ourselves the possibility to do so.

The top example shows a 200 by 200 units large SVG canvas. In the top left quadrant we draw a circle.

The second code snippet employs the `viewBox` attribute, which specifies an area of the image we want to display. In this example we give it a region from (0,0) to (100,100), meaning we specify exactly this upper left quadrant.

`viewBox` now does two things: First, it only draws objects inside this region, i.e. it discards everything outside. Second, it stretches this region to the whole SVG canvas. This means, that our final image is still 200 by 200 units (pixels) in size, but we only see a region of our original image. This gives a zoom effect.

## Annotations in HTML

In the exercise, we will augment our web server by an annotation tool.

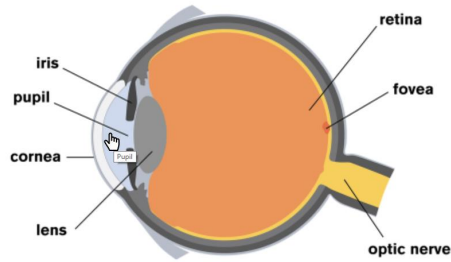
Goal:

- Mark interesting areas and provide meta data.
- Display annotated information.

Our goal for the image database is to be able to highlight interesting areas in the image and display this information to the user.

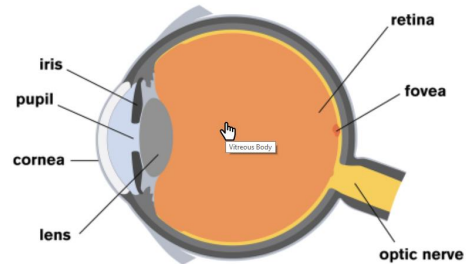
## Image Maps

Image maps allow you to mark regions in an image and assign links to them.  
Example:



Clicking on the pupil leads to:

<https://en.wikipedia.org/wiki/Pupil>



Clicking on the vitreous body leads to:

[https://en.wikipedia.org/wiki/Vitreous\\_body](https://en.wikipedia.org/wiki/Vitreous_body)

Tobii AB, "The human eye," <https://www.tobii.com>, 2019.  
Available: [https://www.tobii.com/imagevault/publishedmedia/4a2hi9p5cq7422k8cndr8/Structures\\_Of\\_The\\_Human\\_Eye.png](https://www.tobii.com/imagevault/publishedmedia/4a2hi9p5cq7422k8cndr8/Structures_Of_The_Human_Eye.png)  
[Accessed: 06-June-2019].

To this end we will first explore HTML image maps. Image maps provide a way to mark areas in an image. These areas act as links, i.e. clicking on them leads to different URLs. For example in this case there are two regions in the image (pupil and vitreous body), which - when clicked on - direct your browser to the respective Wikipedia articles.

## Image Maps in HTML

```
<html>
<body>

  <map name="image-map">
    <area title="Pupil"
          href="https://en.wikipedia.org/wiki/Pupil" coords="102,117,143,219" shape="rect" >
    <area title="Vitreous Body"
          href="https://en.wikipedia.org/wiki/Vitreous_body" coords="242,166,107" shape="circle" >
  </map>

</body>
</html>
```

**<img>** tag specifies image, **usemap** attribute specifies an image map with a name (here **image-map**).

**<map>** (with the same name!) then includes **<area>**s, which have a title (shown on hover) and a link (**<href>**).

Areas are defined by a shape (**rect**, **circle**, **poly**) and some **coords**.

Easy creation of image maps: <https://www.image-map.net/>

Slide 335

We add a new attribute to our **<img>** tag, called **usemap**. This specifies an image map to use. It does so by giving the name of the map.

The map itself is defined just under the image. Note that its name is the same we provided in the **usemap** attribute. Inside the map we define our areas for the two parts of the eye we want to annotate. In this example we use a rectangle for the pupil and a circle for the vitreous body. The **coords** attribute gives information about the shape, i.e. for the rect the upper left and bottom right corner and for the circle the position and radius.

## Problems with HTML Image Maps

HTML image maps suffer from one big problem:

`<area>` does not allow CSS **hover** attribute. This makes it hard to highlight regions on mouse-over (only with JavaScript).

Slide 336

Image maps are useful for certain tasks, but aren't quite what we want here. They are somewhat difficult to work with, especially if you want the areas to react to your mouse.

## SVG Image Maps

Goal: Build an annotation system, which displays information on hover.



George Washington

Abraham Lincoln

Michael Moll, "Die Präsidenten am Mount Rushmore," <https://www.dieweltenbummler.de/> 2017.  
Available: <https://www.dieweltenbummler.de/wp-content/uploads/2017/05/Mount-Rushmore-von-unten-1536x1024.jpg>  
[Accessed: 11-June-2019]

We therefore go a different route, by using SVG. Displayed here is our goal, which we will pursue on the following slides. The rectangles mark certain parts of our image and react to the mouse being moved over them. On the one hand the area is highlighted by the white rectangles. Additionally descriptive text is displayed below the image (in this case the name of the respective president).

## SVG Annotation Implementation – First Steps

```
<html>
<body>

  <svg xmlns="http://www.w3.org/2000/svg" width="1536" height="1024" >

    <!-- Image -->
    <image width="1536" height="1024" xlink:href="mount_rushmore.jpg" />

  </svg>

</body>
</html>
```

SVG, which includes a raster `<image>`.

Let's start simple by creating the standard HTML code skeleton. We also include a raster graphic (our image). Note again, that the image is **not** a vector image. Even though it is embedded in a SVG environment, it will not have the benefits of vector graphics, i.e. it will lose quality when zoomed in on.

## SVG Annotation Implementation – First Steps



Michael Moll, "Die Präsidenten am Mount Rushmore," <https://www.dieweltenbummler.de/> 2017.  
Available: <https://www.dieweltenbummler.de/wp-content/uploads/2017/05/Mount-Rushmore-vons unten-1536x1024.jpg>  
[Accessed: 11-June-2019]

Slide 339

This is the result of code so far. As expected we see our image, not more, not less.

## SVG Annotation Implementation – Areas

```
<html>
<body>

<svg xmlns="http://www.w3.org/2000/svg" width="1536" height="1024" >

  <!-- Image -->
  <image width="1536" height="1024" xlink:href="mount_rushmore.jpg" />

  <!-- Areas in image as rects. -->
  <rect x="300" y="125" width="250" height="300" />
  <rect x="550" y="225" width="200" height="300" />
  <rect x="750" y="375" width="200" height="300" />
  <rect x="999" y="375" width="200" height="300" />

</svg>

</body>
</html>
```

Add four **<rect>**s (one for each president).

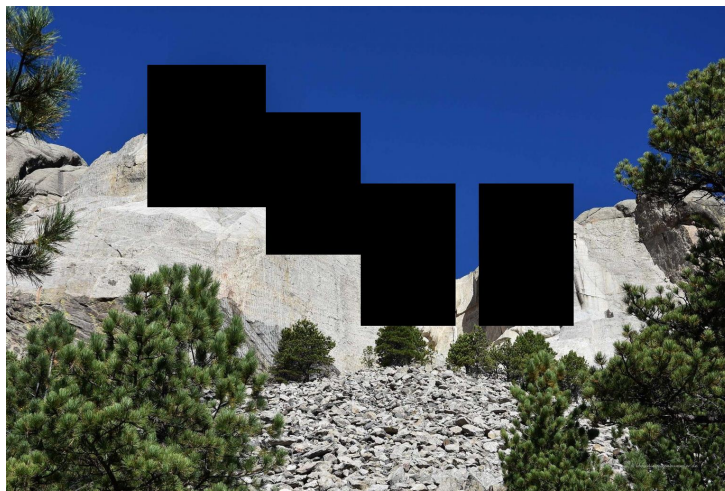
Slide 340

Let's add the rectangles for the annotation. Coordinates of the rectangles can be read from any image processing tool like Microsoft Paint or GIMP.

Note that the order of elements in our SVG matters! Here the `<rect>` tags are specified **after** the image. SVG draws the elements from top to bottom. The rectangles are therefore drawn on top of the image.

Swapping this order would lead to the image being drawn on top of the rectangles. This means, that the rectangles would not be visible!

## SVG Annotation Implementation – Areas



Michael Moll, "Die Präsidenten am Mount Rushmore," <https://www.dieweltenbummler.de/2017>.  
Available: <https://www.dieweltenbummler.de/wp-content/uploads/2017/05/Mount-Rushmore-von-unten-1536x1024.jpg>  
[Accessed: 11-June-2019]

The rectangles are now visible in our SVG. Their color defaults to black, so let's fix this next, so that we can actually see our image again.

## SVG Annotation Implementation – Adding CSS

```
<html>
<head>
  <link rel="stylesheet" href="SVGImageMap.css">
</head>

<body>

  <svg xmlns="http://www.w3.org/2000/svg" width="1536" height="1024" >

    <!-- Image -->
    <image width="1536" height="1024" xlink:href="mount_rushmore.jpg" />

    <!-- Areas in image as rects. -->
    <rect x="300" y="125" width="250" height="300" />
    <rect x="550" y="225" width="200" height="300" />
    <rect x="750" y="375" width="200" height="300" />
    <rect x="999" y="375" width="200" height="300" />

  </svg>

</body>
</html>
```

Add CSS stylesheet. In this case the CSS in a separate file, but you can also embed it directly in the HTML.

Slide 342

We add a CSS stylesheet to our site. This can either be defined in a separate file (like in this example), or be specified directly in the HTML inside of `<style>` tags.

## SVG Annotation Implementation – Adding CSS

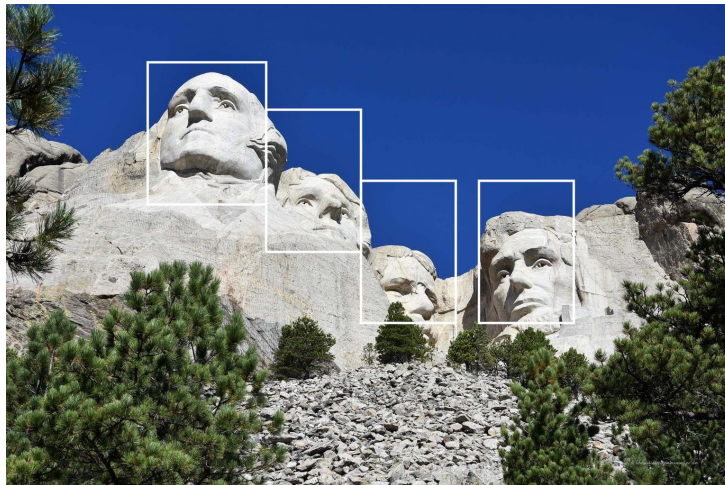
```
rect {
  fill-opacity: 0;
  stroke: white;
  stroke-opacity: 1;
  stroke-width: 5px;
}
```

Simple CSS stylesheet. `<rect>`s are given no fill, and a white stroke.

We define our CSS. Our goal is to give the rectangles a solid white border, but no inner color. We thus change the stroke (border) parameters.

The fill opacity is set to zero, in order to make it completely transparent.

## SVG Annotation Implementation – Adding CSS



Michael Moll, "Die Präsidenten am Mount Rushmore," <https://www.dieweltenbummler.de/2017>.  
Available: <https://www.dieweltenbummler.de/wp-content/uploads/2017/05/Mount-Rushmore-von-unten-1536x1024.jpg>  
[Accessed: 11-June-2019]

Our rectangles are now white and since we set the inner part to transparent, we see the presidents' heads again. However, the rectangles are always visible and do not react to our mouse input. We will fix this next.

## SVG Annotation Implementation – Hover Effect

```
rect {  
  fill-opacity: 0;  
  stroke: white;  
  stroke-opacity: 0;  
  stroke-width: 5px;  
}  
  
rect:hover {  
  stroke-opacity: 1;  
}
```

Set **<rect>** stroke to zero opacity (fully transparent). This makes it invisible. Instead make it opaque on hover.

Slide 345

Since we want the rectangles to be invisible by default, let's start by setting the stroke opacity to zero. Now the areas are never visible.

Next, we give the rectangles a hover selector. This specifies the rectangles' style, whenever the mouse is over the element. This allows us to specialize the appearance for this case.

For the hover-case we set the opacity back to one, meaning full visibility.

## SVG Annotation Implementation – Hover Effect



Michael Moll, "Die Präsidenten am Mount Rushmore," <https://www.dieweltenbummler.de/> 2017.  
Available: <https://www.dieweltenbummler.de/wp-content/uploads/2017/05/Mount-Rushmore-vons unten-1536x1024.jpg>  
[Accessed: 11-June-2019]

Slide 346

The rectangles are now invisible, except when hovered over by the mouse.

## SVG Annotation Implementation – Annotation Text

```
<html>
<head>
  <link rel="stylesheet" href="SVGImageMap.css">
</head>

<body>

  <svg xmlns="http://www.w3.org/2000/svg" width="1536" height="1224" >

    <!-- Image -->
    <image width="1536" height="1024" xlink:href="mount_rushmore.jpg" />

    <!-- Areas in image as rects. -->
    <rect x="300" y="125" width="250" height="300" />
    <text x="100" y="1200">George Washington</text>

    <rect x="550" y="225" width="200" height="300" />
    <text x="100" y="1200">Thomas Jefferson</text>

    <rect x="750" y="375" width="200" height="300" />
    <text x="100" y="1200">Theodore Roosevelt</text>

    <rect x="999" y="375" width="200" height="300" />
    <text x="100" y="1200">Abraham Lincoln</text>
  </svg>

</body>
</html>
```

Let's give ourselves a bit more room at the bottom.  
Increase the height of the SVG.

Add annotation text per element. Note that all **<text>**s have the same position at the bottom of our SVG.

Slide 347

We will now add the description text to each of our annotation areas. Since our text should appear below the image, let's start by giving ourselves a bit more room in the SVG canvas. We thus increase the SVG height by a bit. Note, that this does not impact the image (because it has an own height).

We then add the text. Note, that all text elements have the exact same position below the image. They only differ in the text they display (the name of the president).

We write each text element directly below the corresponding rectangle tag, for reasons we will explain in a bit!

## SVG Annotation Implementation – Annotation Text

```
rect {  
  fill-opacity: 0;  
  stroke: white;  
  stroke-opacity: 0;  
  stroke-width: 5px;  
}  
  
rect:hover {  
  stroke-opacity: 1;  
}  
  
text {  
  fill: black;  
  opacity: 1;  
  font-size: 100px;  
}
```

CSS for text. Set color, opacity and size.

Slide 348

Let's also give our text a style. The text color is specified by the fill attribute. This is the default, so it's not really necessary to specify this. However, oftentimes it is advisable to be as verbose as possible with certain attributes, because this more clearly shows our intention.

## SVG Annotation Implementation – Annotation Text



**Abraham Lincoln**

Michael Moll, "Die Präsidenten am Mount Rushmore," <https://www.dieweltenbummler.de/> 2017.  
Available: <https://www.dieweltenbummler.de/wp-content/uploads/2017/05/Mount-Rushmore-vons unten-1536x1024.jpg>  
[Accessed: 11-June-2019]

Slide 349

We have text! It is not particularly pretty, mainly because all texts are right above each other, but this is expected so far, since we specified all text tags to have the same position. Our main problem is, that the text does not react to our mouse input yet. Remember: Our goal is that each text element is only displayed, when the corresponding rectangle in the image is hovered by the mouse.

## SVG Annotation Implementation – Hover Annotation

```
rect {
  fill-opacity: 0;
  stroke: white;
  stroke-opacity: 0;
  stroke-width: 5px;
}

rect:hover {
  stroke-opacity: 1;
}

text {
  fill: black;
  opacity: 0;
  font-size: 100px;
}

rect:hover + text {
  opacity: 1;
}
```

Add CSS hover effect for `<rect>`s, which effects the `<text>`.

Syntax:

`rect:hover + text {<rules>}`

Selector      Sibling operator      Target

Note, that the `+` operator only affects **siblings** (same level), which are **directly after** the selector element. The order of elements in the HTML is therefore important!

Slide 350

Our approach is analogous to the hovering of the rectangles we did previously. Let's give our text a default opacity of zero, and a hover opacity of one.

Remember though, that the hover selector always influences the element it is specified on, i.e. when writing `text:hover`, and then changing the opacity, this changes the opacity when we hover over the text, **not** when we hover the rectangle. We thus introduce the CSS sibling operator, `+`.

Using the sibling operator, it is possible to change another element's style when a certain element is hovered (or interacted with in a different way). In this case, we give the **rectangle** a hover selector, which then influences the **text**.

The sibling operator influences the **next** element of the specified type (in our case text) in the HTML/SVG. This is why earlier we put the text elements always directly after the rectangle.

This way, when a rectangle is hovered over, the next text element is always the corresponding description and will thus become visible.

## SVG Annotation Implementation – Hover Annotation



George Washington

Abraham Lincoln

Michael Moll, "Die Präsidenten am Mount Rushmore," <https://www.dieweltenbummler.de/2017>.  
Available: <https://www.dieweltenbummler.de/wp-content/uploads/2017/05/Mount-Rushmore-von-s unten-1536x1024.jpg>  
[Accessed: 11-June-2019]

Slide 351

Now our annotation tool is working as expected!

## CSS Image Filters

Goal: Apply image effects (grayscale etc.) directly in CSS.

```
<html>
<body>

<style>

  img {
    filter: grayscale(100%);
  }

</style>



</body>
</html>
```

Demo: <https://codepen.io/rss/pen/ftnDd>

Slide 352

Let's explore more capabilities of CSS. CSS is able to apply operations to images. In this example we make an image gray, by specifying a grayscale filter attribute. The argument of the filter gives us the possibility to make the image only a little gray. Since it is set to 100% in this example, the image is converted to perfect grayscale.

## Some more CSS Filters

The argument values are of course only examples.

```
.blur      { filter: blur(4px); }  
.brightness { filter: brightness(0.30); }  
.contrast  { filter: contrast(180%); }  
.grayscale { filter: grayscale(100%); }  
.huerotate { filter: hue-rotate(180deg); }  
.invert    { filter: invert(100%); }  
.opacity   { filter: opacity(50%); }  
.saturate  { filter: saturate(7); }  
.sepia     { filter: sepia(100%); }  
.shadow    { filter: drop-shadow(8px 8px 10px green); }
```

Slide 353

Here are more examples of image filters. The CSS selectors here start with dots. This makes them influence HTML elements of the respective class name, i.e. the selector `.shadow` gives the HTML element with class `shadow` a drop shadow.

## CSS Blur

```
<html>
<body>

  <style>
    img { filter: blur(4px); }
  </style>

</body>
</html>
```



Slide 354

Blur is an image operation, which mixes each pixel's color with the colors of its neighbor. The operation is thus similar to our edge detection example from earlier, but with different weights per neighbor pixel.

Also, for blur it is possible to specify larger neighborhoods. In this case the radius of our neighborhood is 4 pixels, meaning that we mix the colors of a region with radius 4.

## CSS Contrast

```
<html>
<body>

<style>
  img { filter: contrast(180%); }
</style>



</body>
</html>
```



Slide 355

Contrast makes dark colors darker and light colors lighter for arguments over 100%. This increases the range between the darkest and lightest pixel.

For arguments under 100%, the contrast shrinks.

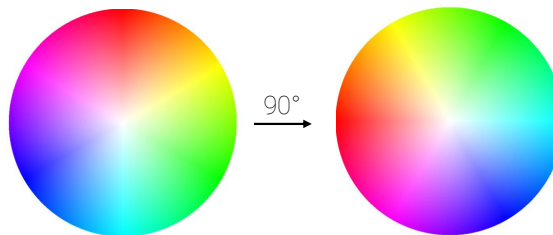
## CSS Hue Rotate

```
<html>
<body>

<style>
  img { filter: hue-rotate(90deg); }
</style>



</body>
</html>
```



The color wheel at the top might look familiar to you. It is a standard way of displaying colors. The outer ring is roughly equivalent with the colors of the rainbow (with some exceptions; purple for example is not a rainbow color).

The hue-rotate filter rotates this color wheel, such that each color lands in a different spot. In our example (90deg), red becomes green. This effect can be observed on Augustus' cloak.

## CSS Filters

CSS filters do not just apply to images!  
(Almost) everything can be filtered.

```
<p class="blur">Text</p>
```

Filters can be combined!

```
.combination {
  filter: blur(4px) grayscale(100%);
}
```

**Disadvantage for image:** Original image is delivered to client. When user saves the image, they get the original!

Images are not the only HTML element which can be filtered. It turns out that you can apply filters to nearly everything in HTML, for example text. Note that here we are using the `blur` class from earlier.

Another useful thing is the combination of CSS filters. For example you can blur an image and then convert it to grayscale, as showcased in the example.

Note that the order is important. Changing the order of these filters yields different results.

One extremely important thing to keep in mind is that CSS is executed on the client (the user's browser). The original image or text is delivered to the client, where the filter is applied. You can try this out by right-clicking a filtered image on a website and saving it to your hard drive. Note, that the original image is saved!

The implication here is, that for certain content it is best to perform the filter on the server and then deliver the filtered content to the user, so that he or she does not even have the possibility to get the original. This however also means more computation on the server, which might be expensive.

As a rule of thumb: perform as much as possible on the client side (CSS and JavaScript) and as much as necessary on the server (for example Python in Bottle).

## CSS Animations

```
img {  
  animation: invertAnimation 1s forwards;  
}  
  
@keyframes invertAnimation {  
  from {  
    filter: none;  
  }  
  to {  
    filter: invert(100%);  
  }  
}
```

Slide 358

A fun thing to play around with are CSS animations. Animations allow you to change state of an object over time. In this case we define an animation called *invertAnimation* which applies an inversion-filter. The syntax specifies that at the beginning of the animation, no filter should be applied and in the end we want the image to be completely inverted.

We then apply the animation to all elements of tag `<img>`. We declare that the animation should run one second (1s), so the image is inverted after one second.

The last attribute specifies what should happen after the animation is completed. **forwards** means that the element should simply stay how it is, so it stays inverted after the one second.

## SVG Filters

```
<svg xmlns="http://www.w3.org/2000/svg" width="1536" height="1024">

  <style>
    image {
      filter: url(#myCustomFilter);
    }
  </style>

  <image width="1536" height="1024" xlink:href="mount_rushmore.jpg" />

  <!-- Image filter -->
  <filter id="myCustomFilter">
    <feGaussianBlur stdDeviation="5" />
  </filter>

</svg>
```

Slide 359

Unfortunately in SVG the filtering works differently. In this example we define a filter at the bottom. We give it a name (*myCustomFilter*), which we can then reference in the CSS snippet above. With the `url` function we can apply a filter with the given name to all images.

The *Gaussian Blur* filter here is similar to the *blur* filter in CSS.

## SVG Filters

```
<svg xmlns="http://www.w3.org/2000/svg" width="1536" height="1024">

  <style>
    image {
      filter: url(#myCustomFilter);
    }
  </style>

  <image width="1536" height="1024" xlink:href="mount_rushmore.jpg" />

  <!-- Image filter -->
  <filter id="myCustomFilter">
    <feGaussianBlur stdDeviation="5" />
    <feColorMatrix type="saturate" values="0.1" />
  </filter>

</svg>
```



Slide 360

Similarly to HTML, we can combine filters. In this case we apply a saturation filter after the blur. This is similar to a grayscale filter.

## Chapter 13

# Legal Foundations of Information Technology

In this Chapter, we cover a topic that is a very important secondary aspect of our work as knowledge workers that – at best – create immaterial things: the legal foundations that regulate how the fruits of our labor are appreciated (and – importantly – recompensated), and what we have to do to respect people’s personal data.

 **Caveat** : The content of this Chapter are about legal matters, but are written by a computer scientist, i.e. not a legal expert. They should be considered as an introduction of the fundamental concepts involved, and definitely not as legal advice. For that, contact an [intellectual property lawyer](#).

That being said, we expect that understanding the concepts covered in this Chapter will help you with getting most out of this conversation.

### 13.1 Intellectual Property

The first complex of questions centers around the assessment of the products of work of knowledge/information workers, which are largely intangible, and about questions of recompensation for such work.

#### Intellectual Property: Concept

- ▷ **Question:** Intellectual labour creates (intangible) objects, can they be [owned](#)?
- ▷ **Answer:** Yes: in certain circumstances they are [property](#) like tangible objects.
- ▷ **Definition 13.1.1** The concept of [intellectual property](#) motivates a set of laws that regulate [property rights](#) on intangible objects, in particular
  - ▷ [Patents](#) grant exploitation rights on original ideas.
  - ▷ [Copyrights](#) grant personal and exploitation rights on expressions of ideas.
  - ▷ [Industrial Design Rights](#) protect the visual design of objects beyond their function.
  - ▷ [Trademarks](#) protect the signs that identify a legal entity or its products to establish brand recognition.

- ▷ **Intent:** Property-like treatment of intangibles will foster innovation by giving individuals and organizations material incentives.



To understand **intellectual property** better, let us recap the concepts of **property** and **ownership** in general.

### Background: Property and Ownership in General

- ▷ **Definition 13.1.2** **Ownership** is the state or fact of exclusive rights and control over **property**, which may be a physical object, land/real estate or intangible object.
- ▷ **Definition 13.1.3** **Ownership** involves multiple rights (the **property rights**), which may be separated and held by different parties.
- ▷ **Definition 13.1.4** There are various legal entities (e.g. persons, states, companies, associations, ...) that can have **ownership** over a **property**  $p$ . We call them the **owners** of  $p$ .
- ▷ **Remark 13.1.5** Depending on the nature of the **property**, an owner of property has the right to consume, alter, share, redefine, rent, mortgage, pawn, sell, exchange, transfer, give away or destroy it, or to exclude others from doing these things, as well as to perhaps abandon it.
- ▷ **Remark 13.1.6** The process and mechanics of **ownership** are fairly complex: one can gain, transfer, and lose **ownership** of property in a number of ways.



These concepts are the basis for many other concepts such as money, trade, debt, bankruptcy, and the criminality of theft. **Ownership** is the key building block in the development of the capitalist socio-economic system, must influentially developed in Adam Smith's book *An Inquiry into the Nature and Causes of the Wealth of Nations* [Smi76] from 1776.

Naturally, many of the concepts are hotly debated. Especially due to the fact that intuitions and legal systems about property have evolved around the more tangible forms of properties that cannot be simply duplicated and indeed multiplied by copying them. In particular, other intangibles like physical laws or mathematical theorems cannot be property.

### Intellectual Property: Problems

- ▷ **Delineation Problems:** How can we distinguish the product of human work, from "discoveries", of e.g. algorithms, facts, genome, algorithms. (not property)
- ▷ **Philosophical Problems:** The implied analogy with physical property (like land or an automobile) fails because physical property is generally rivalrous while intellectual works are non-rivalrous (the enjoyment of the copy does not prevent enjoyment of the original).

- ▷ **Practical Problems:** There is widespread criticism of the concept of intellectual property in general and the respective laws in particular.
  - ▷ (software) patents are often used to stifle innovation in practice. (patent trolls)
  - ▷ copyright is seen to help big corporations and to hurt the innovating individuals



©: Michael Kohlhase

363



We will not go into the philosophical debates around intellectual property here, but concentrate on the legal foundations that are in force now and regulate IP issues. We will see that groups holding alternative views of intellectual properties have learned to use current IP laws to their advantage and have built systems and even whole sections of the software economy on this basis.

Many of the concepts we will discuss here are regulated by laws, which are (ultimately) subject to national legislative and judicative systems. Therefore, none of them can be discussed without an understanding of the different jurisdictions. Of course, we cannot go into particulars here, therefore we will make use of the classification of jurisdictions into two large legal traditions to get an overview. For any concrete decisions, the details of the particular jurisdiction have to be checked.

### Legal Traditions

- ▷ The various legal systems of the world can be grouped into “traditions”.
- ▷ **Definition 13.1.7** Legal systems in the **common law tradition** are usually based on case law, they are often derived from the British system.
- ▷ **Definition 13.1.8** Legal systems in the **civil law tradition** are usually based on explicitly codified laws (civil codes).
- ▷ As a rule of thumb all English-speaking countries have systems in the **common law tradition**, whereas the rest of the world follows a **civil law tradition**.



©: Michael Kohlhase

364



Another prerequisite for understanding **intellectual property** concepts is the historical development of the legal frameworks and the practice how intellectual property law is synchronized internationally.

### Historic/International Aspects of Intellectual Property Law

- ▷ **Early History:** In **late antiquity** and the **middle ages** IP matters were regulated by royal privileges
- ▷ **History of Patent Laws:** First in Venice 1474, Statutes of Monopolies in England 1624, US/France 1790/1...
- ▷ **History of Copyright Laws:** Statue of Anne 1762, France: 1793, ...
- ▷ **Problem:** In an increasingly globalized world, national **IP** laws are not enough.
- ▷ **Definition 13.1.9** The **Berne convention** process is a series of international treaties that try to harmonize international **IP** laws. It started with the original

Berne convention 1886 and went through revision in 1896, 1908, 1914, 1928, 1948, 1967, 1971, and 1979.

- ▷ The World Intellectual Property Organization Copyright Treaty was adopted in 1996 to address the issues raised by information technology and the Internet, which were not addressed by the Berne Convention.
- ▷ **Definition 13.1.10** The **Anti-Counterfeiting Trade Agreement** (ACTA) is a multinational treaty on international standards for intellectual property rights enforcement.
- ▷ With its focus on enforcement **ACTA** is seen by many to break fundamental human information rights, criminalize **FLOSS**.



## 13.2 Copyright

In this Section, we go into more detail about a central concept of **intellectual property** law: copyright is the component most of IP law applicable to the individual computer scientist. Therefore a basic understanding should be part of any CS education. We start with a definition of what works can be copyrighted, and then progress to the rights this affords to the copyright holder.

### Copyrightable Works

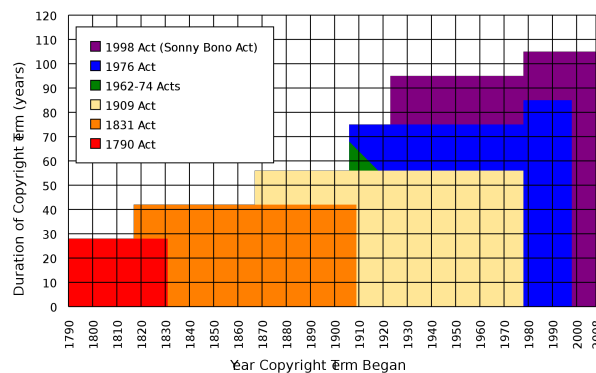
- ▷ **Definition 13.2.1** A **copyrightable work** is any artefact of human labor that fits into one of the following eight categories:
  - ▷ **Literary works**: Any work expressed in letters, numbers, or symbols, regardless of medium. (Computer source code is also considered to be a literary work.)
  - ▷ **Musical works**: Original musical compositions.
  - ▷ **Sound recordings** of musical works. (different licensing)
  - ▷ **Dramatic works**: literary works that direct a performance through written instructions.
  - ▷ **Choreographic works** must be “fixed,” either through notation or video recording.
  - ▷ **Pictorial, graphic and sculptural work** (**PGS works**): Any two-dimensional or three-dimensional art work
  - ▷ **Audiovisual works**: work that combines audio and visual components. (e.g. films, television programs)
  - ▷ **Architectural works** (copyright only extends to aesthetics)
- ▷ The categories are interpreted quite liberally (e.g. for computer code).
- ▷ There are various requirements to make a work copyrightable: it has to
  - ▷ exhibit a certain originality (Schöpfungshöhe)
  - ▷ require a certain amount of labor and diligence (“sweat of the brow” doctrine)



In short almost all products of intellectual work are **copyrightable**, but this does not mean copyright applies to all those works. Indeed there is a large body of works that are “out of copyright”, and can be used by everyone. Indeed it is one of the intentions of **intellectual property** laws to increase the body of intellectual resources a society a draw upon to create wealth. Therefore copyright is limited by regulations that limit the duration of copyright and exempts some classes of works from copyright (e.g. because they have already been paid for by society).

### Limitations of Copyrightability: The Public Domain

- ▷ **Definition 13.2.2** A work is said to be in the **public domain**, if no copyright applies, otherwise it is called **copyrighted**.
- ▷ **Example 13.2.3** Works made by US government employees (in their work time) are in the public domain directly (Rationale: taxpayer already paid for them)
- ▷ **Copyright expires**: usually 70 years after the death of the creator
- ▷ **Example 13.2.4 (US Copyright Terms)** Some people claim that US copyright terms are extended, whenever Disney’s Mickey Mouse would become **public domain**.



Now that we have established, which works are **copyrighted** — i.e. to which works are **intellectual property**, we now turn to the rights owning such a property entails.

### Rights under Copyright Law

- ▷ **Definition 13.2.5** The **copyright** is a collection of rights on a **copyrighted** work;
  - ▷ **personal rights**: the copyright holder may
    - ▷ determine whether and how the work is published (right to publish)
    - ▷ determine whether and how her authorship is acknowledged. (right of attribution)

- ▷ to object to any distortion, mutilation or other modification of the work, which would be prejudicial to his honor or reputation (**droit de respect**)
- ▷ **exploitation rights**: the owner of a copyright has the exclusive right to do, or authorize to do any of the following:
  - ▷ to reproduce the copyrighted work in copies (or phonorecords);
  - ▷ to prepare derivative works based upon the copyrighted work;
  - ▷ to distribute copies of the work to the public by sale, rental, lease, or lending;
  - ▷ to perform the copyrighted work publicly;
  - ▷ to display the copyrighted work publicly; and
  - ▷ to perform the copyrighted work publicly by means of a digital-audio transmission.
- ▷ **Remark 13.2.6** Formally, it is not the **copyrightable work** that can be owned itself, but the **copyright**.
- ▷ **Definition 13.2.7** The use of a **copyrighted** material, by anyone other than the owner of the **copyright**, amounts to **copyright infringement** only when the use is such that it conflicts with any one or more of the exclusive rights conferred to the owner of the copyright.



Initially, and by default the **copyright** of an intellectual work is owned by the creator. But – as with any property – **copyrights** can be transferred. We will now

## Copyright Holder

- ▷ **Definition 13.2.8** The **copyright holder** is the legal entity that owns the **copyright** to a **copyrighted** work.
- ▷ By default, the original creator of a copyrightable work holds the copyright.
- ▷ In most jurisdictions, no registration or declaration is necessary (**but copyright ownership may be difficult to prove**)
- ▷ copyright is considered **intellectual property**, and can be transferred to others (**e.g. sold to a publisher or bequeathed**)
- ▷ **Definition 13.2.9 (Work for Hire)** A **work made for hire (WFH)** is a work created by an employee as part of his or her job, or under the explicit guidance or under the terms of a contract.
- ▷ In jurisdictions from the **common law tradition**, the copyright holder of a **WFH** is the employer, in jurisdictions from the **civil law tradition**, the author, unless the respective contract regulates it otherwise.



Again, the rights of the **copyright holder** are mediated by usage rights of society; recall that **intellectual property** laws are originally designed to increase the intellectual resources available to society.

### Limitations of Copyright (Citation/Fair Use)

- ▷ There are limitations to the exclusivity of rights of the **copyright holder** (some things cannot be forbidden)
- ▷ **Citation Rights:** **Civil law jurisdictions** allow citations of (extracts of) copyrighted works for scientific or artistic discussions. (note that the right of attribution still applies)
- ▷ In the **civil law tradition**, there are similar rights:
- ▷ **Definition 13.2.10 (Fair Use/Fair Dealing Doctrines)** Case law in **common law traditions** has established a **fair use doctrine**, which allows e.g.
  - ▷ making safety copies of software and audiovisual data
  - ▷ lending of books in public libraries
  - ▷ citing for scientific and educational purposes
  - ▷ excerpts in search engine

Fair use is established in court on a case-by-case taking into account the purpose (commercial/educational), the nature of the work the amount of the excerpt, the effect on the marketability of the work.



©: Michael Kohlhase

370



## 13.3 Licensing

Given that **intellectual property** law grants a set of exclusive rights to the owner, we will now look at ways and mechanisms how usage rights can be bestowed on others. This process is called licensing, and it has enormous effects on the way software is produced, marketed, and consumed. Again, we will focus on copyright issues and how innovative license agreements have created the open source movement and economy.

### Licensing: the Transfer of Rights

- ▷ **Remember:** the **copyright holder** has **exclusive rights** to a **copyrighted** work.
- ▷ **In particular:** all others have only **fair-use rights** (but we can transfer rights)
- ▷ **Definition 13.3.1** A **license** is an authorization (by the **licensor**) to use the licensed material (by the **licensee**).
- ▷ **Note:** a **license** is a regular contract (about **intellectual property**) that is handled just like any other contract. (it can stipulate anything the licensor and licensees agree on) in particular a license may
  - ▷ involve **term**, **territory**, or **renewal** provisions
  - ▷ require paying a fee and/or proving a capability.

▷ require to keep the licensor informed on a type of activity, and to give them the opportunity to set conditions and limitations.

- ▷ **Mass Licensing of Computer Software:** Software vendors usually license software under extensive **end-user license agreement** (EULA) entered into upon the installation of that software on a computer. The license authorizes the user to install the software on a limited number of computers.



©: Michael Kohlhase

371



Copyright law was originally designed to give authors of literary works — e.g. novelists and playwrights — revenue streams and regulate how publishers and theatre companies can distribute and display them so that society can enjoy more of their work.

With the inclusion of software as “**literary works**” under copyright law the basic parameters of the system changed considerably:

- modern software development is much more a collaborative and diversified effort than literary writing,
- re-use of software components is a decisive factor in software,
- software can be distributed in compiled form to be executable which limits inspection and re-use, and
- distribution costs for digital media are negligible compared to printing.

As a consequence, much software development has been industrialized by large enterprises, who become **copyright holder** as the software was created as **work for hire**. This has led to software quasi-monopolies, which are prone to stifling innovation and thus counteract the intentions of intellectual property laws.

The **Free/Open Source Software** movement attempts to use the **intellectual property** laws themselves to counteract their negative side effects on innovation and collaboration and the (perceived) freedom of the programmer.

### Free/Libre/Open-Source Licenses

- ▷ **Recall:** Software is treated as literary works wrt. copyright law.
- ▷ **But:** Software is different from literary works wrt. distribution channels (**and that is what copyright law regulates**)
- ▷ **In particular:** When literary works are distributed, you get all there is, software is usually distributed in binary format, you cannot understand/cite/modify/fix it.
- ▷ **So:** Compilation can be seen as a technical means to enforce copyright. (**seen as an impediment to freedom of fair use**)
- ▷ **Recall:** IP laws (in particular patent law) was introduced explicitly for two things
- ▷ incentivize innovation (by **granting exclusive exploitation rights**)
  - ▷ spread innovation (by **publishing ideas and processes**)
- Compilation breaks the second tenet (and may thus stifle innovation)

- ▷ **Idea:** We should create a public domain of source code
- ▷ **Definition 13.3.2** **Free/Libre/Open-Source Software (FLOSS or just open source)** is software that is and licensed via **licenses** that ensure that its source code is available.
- ▷ Almost all of the Internet infrastructure is (now) **FLOSS**; so are the Linux and Android operating systems and applications like OpenOffice and The GIMP.



©: Michael Kohlhase

372



The relatively complex name **Free/Libre/Open Source** comes from the fact that the English<sup>1</sup> word “free” has two meanings: free as in “freedom” and free as in “free beer”. The initial name “free software” confused issues and thus led to problems in public perception of the movement. Indeed Richard Stallman’s initial motivation was to ensure the freedom of the programmer to create software, and only used cost-free software to expand the software public domain. To disambiguate some people started using the French “libre” which only had the “freedom” reading of “free”. The term “open source” was eventually adopted in 1998 to have a politically less loaded label.

The main tool in brining about a **public domain** of **open-source software** was the use of licenses that are cleverly crafted to guarantee usage rights to the public and inspire programmers to license their works as open-source systems. The most influential license here is the GNU public license which we cover as a paradigmatic example.

### GPL/Copyleft: Creating a FLOSS Public Domain?

- ▷ **Problem:** How do we get people to contribute source code to the **FLOSS** public domain?
- ▷ **Idea:** Use special licenses to:
  - ▷ allow others to use/fix/modify our source code (derivative works)
  - ▷ require them to release their modifications to the **FLOSS** public domain if they do.
- ▷ **Definition 13.3.3** A **copyleft** license is a license which requires that allows derivative works, but requires that they be licensed with the same license.
- ▷ **Definition 13.3.4** The **General Public License (GPL)** is a **copyleft** license for **FLOSS** software originally written by Richard Stallman in 1989. It requires that the source code of GPL-licensed software be made available.
- ▷ The GPL was the first copyleft license to see extensive use, and continues to dominate the licensing of **FLOSS** software.
- ▷ **FLOSS** based development can reduce development and testing costs (but community involvement must be managed)
- ▷ Various software companies have developed successful business models based on **FLOSS** licensing models. (e.g. Red Hat, Mozilla, IBM, ...)



©: Michael Kohlhase

373



<sup>1</sup>the movement originated in the USA

**Note:** that the GPL does not make any restrictions on possible uses of the software. In particular, it does not restrict commercial use of the copyrighted software. Indeed it tries to allow commercial use without restricting the freedom of programmers. If the unencumbered distribution of source code makes some business models (which are considered as “extortion” by the open-source proponents) intractable, this needs to be compensated by new, innovative business models. Indeed, such business models have been developed, and have led to an “open-source economy” which now constitutes a non-trivial part of the software industry.

With the great success of **open-source software**, the central ideas have been adapted to other classes of copyrightable works; again to create and enlarge a public domain of resources that allow re-use, derived works, and distribution.

### Open Content/Data via Open Licenses

- ▷ **Recall:** **FLOSS** licenses have created a vibrant public domain for software.
- ▷ **How about:** (not so different from software)
  - ▷ other copyrightable works: music, video, literature, technical documents
  - ▷ data (including **research data**)
- ▷ **Idea:** Adapt the **FLOSS license** ideas to the particular domain  $X \leadsto$  **open  $X$** .
  - ▷ **open content:** pictures, music, video, documents, ...  $\leadsto$  **Creative Commons**
  - ▷ **open data:** data from science, government, and organizations, ...  
 $\leadsto$  **Open Data Commons [OpenDataCommons:on]**.
  - ▷ **open licenses** for many other domains  $X$ .
- ▷ **Open  $X$  Incentives (why open  $X$  communities grow):** Open  $X$ 
  - ▷ incentivize other authors to **extend/improve the  $X$**   
 $\leadsto$  more/better  $X$  can be generate at a lower cost.
  - ▷ generate **attention** to the  $X$  and **recognition** for authors  
 $\leadsto$  this gives alternative revenue models for authors.
- ▷ **Open  $X$  Slogan:** Publish  $X$  early, publish  $X$  often!



©: Michael Kohlhase

374

**FAU**  
 FRIEDRICH-ALEXANDER  
 UNIVERSITÄT  
 ERLANGEN-NÜRNBERG

### Creative Commons a System of Open Content Licenses

**Definition 13.3.5** The **Creative Commons licenses** are

- ▷ a **common legal vocabulary** for sharing content
- ▷ to create a kind of “public domain” using licensing
- ▷ presented in three layers (human/lawyer/machine)-readable
- ▷ Creative Commons license provisions (<http://www.creativecommons.org>)



- ▷ **author retains copyright** on each module/course
- ▷ **author licenses** material to the world with requirements
  - +/- **attribution** (must reference the author)
  - +/- **commercial use** (can be restricted)
  - +/- **derivative works** (can allow modification)
  - +/- **share alike** (copyleft) (modifications must be donated back)



## 13.4 Information Privacy

The last big topic in this chapter is information privacy. This affects us in IWGS in a different way than the previous ones. As providers of information systems we are subject to regulations that require us to keep user's **personally identifiable information** (PII) private to the extent possible and keep inform users informed of what happens to it.

### Information/Data Privacy

- ▷ **Definition 13.4.1** The principle of **information privacy** comprises the idea that humans have the right to control who can access their personal **data**.
- ▷ **Information privacy** concerns exist wherever personally identifiable information is collected and stored – in digital form or otherwise. In particular in the following contexts
  - ▷ Healthcare records
  - ▷ Criminal justice investigations and proceedings
  - ▷ Financial institutions and transactions
  - ▷ Biological traits, such as ethnicity or genetic material
  - ▷ Residence and geographic records
- ▷ **Information privacy** is becoming a growing concern with the advent of the **Internet** and **web search engines** that make access to information easy and efficient.
- ▷ The “reasonable expectation of privacy” is regulated by special laws.
- ▷ **Intuition:** Acquisition and storage of personal data is only legal for the purposes of the respective transaction, must be minimized, and distribution of personal data is generally forbidden with few exceptions. Users have to be informed about collection of personal data.



The

### The General Data Protection Regulation (GDPR)

▷ **Definition 13.4.2** The **General Data Protection Regulation (GDPR)** is a **EU regulation** created in 2016 to harmonize **information privacy** regulations within Europe.

The **GDPR** applies to **data controllers**, i.e. organizations that process personal data of EU citizens (the **data subjects**).

It sanctions violations to **GDPR** mandates with substantial punishments – up to 20M€ or 4% of annual worldwide turnover.

▷ **Remark 13.4.3** As an **EU regulation**, the **GDPR** is directly effective in all EU member countries. (enforced since 2018)

▷ The **GDPR** applies to **data controllers** outside the EU, **iff** they

1. offer goods or services to EU citizens, or
2. monitor their behavior



## Organizational Measures or Information Privacy (GDPR)

▷ **Physical access control**: Unauthorized persons may not be granted physical access to data processing equipment that process personal data. (↪ **locks, access control systems**)

▷ **System access control**: Unauthorized users may not use systems that process personal data (↪ **passwords, firewalls, ...**)

▷ **Information access control**: Users may only access those data they are authorized to access. (↪ **access control lists, safe boxes for storage media, encryption**)

▷ **Data transfer control**: Personal data may not be copied during transmission between systems (↪ **encryption**)

▷ **Input control**: It must be possible to review retroactively who entered, changed, or deleted personal data. (↪ **authentication, journaling**)

▷ **Availability control**: Personal data have to be protected against loss and accidental destruction (↪ **physical/building safety, backups**)

▷ **Obligation of separation**: Personal data that was acquired for separate purposes has to be processed separately.



## Personally Identifiable Information (GDPR)

▷ **Definition 13.4.4** **Personally identifiable information (PII)** is information that, when used alone or with other relevant data, can identify an individual.

PII may contain **direct identifiers** (e.g., passport information) that can identify a person uniquely, or **quasi-identifiers** (e.g., race) that can be combined with other **quasi-identifiers** (e.g., date of birth) to successfully recognize an individual.

- ▷ Under the **GDPR**, any PII a site collects must be either **anonymized**, i.e. PII deleted, or **pseudonymized** (with the consumer's identity replaced with a pseudonym).
- ▷ With **pseudonymization** companies can still do data analysis that would be impossible with anonymization.



## Customer-Service Requirements (GDPR)

- ▷ Visitors must be notified of data the site collects from them and explicitly consent to that information-gathering (This site uses cookies ~ Agree)
- ▷ **data controllers** must notify **data subjects** in a timely way (72h) if any of their personal data held by the site is breached.
- ▷ The **data controller** needs to specify a data-protection officer (DPO).
- ▷ **data subjects** have the right to have their presence on the site erased
- ▷ **data subjects** can request the disclosure all data the **data controller** collected on them. (if the request is in writing, the answer must be on paper)





## Chapter 14

# Ontologies, Semantic Web for Cultural Heritage

In the last Chapter IWGS, we will discuss a virtual research environment for [cultural heritage](#). Before we present the system itself, we take a close look at the underlying technology: ontologies, semantic web technologies, and linked open data.

### 14.1 Documenting our Cultural Heritage

Before we even start talking about the WissKI system, we should become clear on the concepts involved. We start out with the notion of [cultural heritage](#) itself.

#### Documenting our Cultural Heritage

- ▷ **Definition 14.1.1** [Cultural heritage](#) is the legacy of physical artifacts – [cultural artefacts](#) – and practices, representations, expressions, knowledges, or skills – [intangible cultural heritage \(ICH\)](#) – of a group or society that is inherited from past generations.
- ▷ **Problem:** How can we understand, conserve, and learn from our [cultural heritage](#)?
- ▷ **Traditional Answer:** We collect [cultural artefacts](#), study them carefully, relate them to other [artefacts](#), discuss the findings, and publish the results. We display the [artefacts](#) in museums and galleries, and educate the next generation.
- ▷ **DigHumS Answer:** In “Digital Humanities and Social Sciences”, we want to represent our [cultural heritage](#) digitally, and utilize computational tools to do so.
- ▷ **Practical Question:** What are the best representation formats and tools?



©: Michael Kohlhase

381



There is another context in which we want to understand the WissKI system: that of [research data](#). We will introduce the basic concepts now.

## Research Data in a Nutshell

- ▷ **Definition 14.1.2** **Research data** is any **information** that has been collected, observed, generated or created to validate original research findings. Although usually digital, research data also includes non-digital formats such as laboratory notebooks and diaries.
- ▷ **Types of research data:**
  - ▷ documents, spreadsheets, laboratory notebooks, field notebooks, diaries,
  - ▷ questionnaires, transcripts, codebooks, test responses,
  - ▷ audiotapes, videotapes, photographs, films,
  - ▷ **cultural artefacts**, specimens, samples,
  - ▷ data files, database contents (video, audio, text, images), digital outputs,
  - ▷ models, algorithms, scripts,
  - ▷ contents of an application (input, output, logfiles, schemata),
  - ▷ methodologies and workflows, standard operating procedures, and protocols,
- ▷ **Non-digital Research Data** such as **cultural artefacts**, laboratory notebooks, ice-core samples, or sketchbooks is often unique. Materials could be digitized, but this may not be possible for all types of **data**.



The very idea of **research data** is they are retained to justify the published research: in particular just publishing tables of results and experiment descriptions in journals is not enough.

In the past, this has led to the practice of keeping meticulous lab books in the experimental sciences, and in recent times to the practice of publishing original data together with the results, so that experiments can be replicated and derived results can be re-calculated. This being pushed through the scientific organizations in the last decades.

But publishing raw data is also insufficient: experiments can only be replicated and derivations can only be checked if the underlying data can be obtained in practice, are complete and correct, and can be interpreted by the reader. This has

## FAIR Research Data: The Next Big Thing

- ▷ **Principle:** Scientific experiments must be replicated, and derivations must be checkable to be trustworthy. (consensus of scientific community)
- ▷ **Intuition:** **Research data** must be retained for justification, shared for synergies!
- ▷ **Consequence:** Virtually all scientific funding agencies now require some kind of **research data** strategy in proposals. (tendency: getting stricter)
- ▷ **Problem:** Not all forms of **data** are actually useable in practice.
- ▷ **Definition 14.1.3 (Gold Standard Criteria)** **Research data** should be **FAIR**:

- ▷ **Findable**: easy to identify and find for both humans and computers, e.g. with metadata that facilitate searching for specific datasets,
- ▷ **Accessible**: stored for long term so that they can easily be accessed and/or downloaded with well-defined access conditions, whether at the level of meta-data, or at the level of the actual data,
- ▷ **Interoperable**: ready to be combined with other datasets by humans or computers, without ambiguities in the meanings of terms and values,
- ▷ **Reusable**: ready to be used for future research and to be further processed using computational methods.

Consensus in the [research data](#) community; for details see [FAIR18; Wil+16].

- ▷ **Open Question**: How can we achieve [FAIR](#)-ness in for a discipline in practice?



©: Michael Kohlhase

383



After these general considerations about [research data](#), let us come back our primary concern in IWGS: [research data](#) in the humanities and social sciences.

If we look at the categories of [research data](#) we can expect in the humanities and social sciences, then we can categorize them into four broad categories. And we can see that we have already learned about many of them in IWGS.

### Categories of Data in DigiHumS and their Formats

- ▷ We distinguish four broad categories of [data](#) in DigiHumS.
- ▷ **Concrete data**: digital representations of [artefacts](#) in terms of simple data,
  - ▷ e.g. images as pixel arrays in JPEG. (see Chapter 12)
  - ▷ e.g. books identified by author/title/publisher/pubyear. (see Chapter 9)
- ▷ **Narrative data**: documents and text fragments used for communicating knowledge to humans.
  - ▷ e.g. [plain text](#) and [formatted text](#) with [markup codes](#) (see Chapter 4)
- ▷ **Symbolic data**: descriptions of object and facts in a formal language
  - ▷ e.g. `3+5` in python (see Chapter 2)
- ▷ **Metadata**: “data about data”, e.g. who has created these facts, images, or documents, how do they relate to each other? (not covered yet)
- ▷ **Metadata** are the resources, DigiHumS results are made of (→ support that)  
The other categories digitize [artefacts](#) and auxiliary data.
- ▷ We will need all of these – and their combinations – to do DigiHumS.



©: Michael Kohlhase

384



The last kind – [metadata](#) – is arguably the most important kind in the it concerns the relations between [artefacts](#), which are usually digitized into [concrete data](#).

### WissKI: a Virtual Research Env. for Cultural Heritage

- ▷ **Definition 14.1.4** WissKI is a virtual research environment (VRE) for managing scholarly data and documenting cultural heritage.
- ▷ **Requirements:** For a virtual research environment for cultural heritage, we need
  - ▷ scientific communication about and documentation of the cultural heritage
  - ▷ networking knowledge from different disciplines (transdisciplinarity)
  - ▷ high-quality data acquisition and analysis
  - ▷ safeguarding authorship, authenticity, persistence
  - ▷ support of scientific publication
- ▷ WissKI was developed by the research group of Prof. Günther Görtz at FAU Erlangen-Nürnberg and is now used in hundreds of DH projects across Germany.
- ▷ FAU supports cultural heritage research by providing hosted WissKI instances.
  - ▷ See <https://wisski.agfd.fau.de> for details
  - ▷ We will use an instance for the Kirmes paintings in the homework assignments



©: Michael Kohlhasse

385



This leads to the following plan for the rest of the chapter.

### Documenting Cultural Heritage: Current State/Preview

- ▷ Pre-DH State of cultural heritage documentation:
  - ▷ scientific communication/documentation by journal articles/books
  - ▷ persistence: paper records, file cards, databases (like our KirmesDB)
  - ▷ Analysis: manual examination of artefacts in museums/archives.
- ▷ **Idea:** Use more technology to do better.
- ▷ **Preview:** WissKI uses Semantic Web technologies to do just that. We will now
  - ▷ Motivate the Semantic Web (why do we need more than the WWW)
  - ▷ introduce ontologies, linked open data and their technology stacks
  - ▷ show off WissKI and offer a little project based on Kirmes corpus.



©: Michael Kohlhasse

386



## 14.2 Systems for Documenting the Cultural Heritage

Let us now have a look at how we can use digital systems to document the cultural heritage. This is the backdrop against which we need to position the WissKI system.

The traditional methods of documenting **cultural artefacts** is in form of – often handwritten – ledgers that inventory the collections of museums.

### Documenting Cultural Artefacts: Inventory Books

▷ **Definition 14.2.1** An **inventory book** is a ledger that identifies, describes, and records provenance of the **artefacts** in the collection of a museum.

▷ **Example 14.2.2 (An Inventory Book)**

INVENTAR JAHR NR.	KUNSTLER	GEGENSTAND, BESCHREIBUNG, BEZEICHNUNG	TECHNIK, VERSTOFF	MAASSE	EDWERTHUNG	ANKAUF'S PREIS	DOZIRUNG'S PREIS	BEMERKUNGEN
1799/99	Prissum	Raffaello	Frische und Blau- und Weisse	H. 11 1/2 B. 12 1/2 L. 12 1/2	Ein Stein Klein- tafel	2.15		
85	Tischlein	Wappenstein Klein- tafel	1/2, sehr Fein- auf- gezeichnet auf Papier mit einer Weisse Frische	H. 11 1/2 B. 12 1/2 L. 12 1/2	Ein Stein Klein- tafel			
86	Frische	Ein Stein Klein- tafel mit einer Weisse Frische	Ein Stein Klein- tafel	H. 11 1/2 B. 12 1/2 L. 12 1/2	Ein Stein Klein- tafel	8.-		1799/99
87	Kallmann	Ein Stein Klein- tafel mit einer Weisse Frische	Ein Stein Klein- tafel	H. 11 1/2 B. 12 1/2 L. 12 1/2	Ein Stein Klein- tafel	54.-		1799/99
88	Dick	Ein Stein Klein- tafel mit einer Weisse Frische	Ein Stein Klein- tafel	H. 11 1/2 B. 12 1/2 L. 12 1/2	Ein Stein Klein- tafel	42.-		1799/99
89	Kallmann	Ein Stein Klein- tafel mit einer Weisse Frische	Ein Stein Klein- tafel	H. 11 1/2 B. 12 1/2 L. 12 1/2	Ein Stein Klein- tafel			1799/99

**Problems:** non-digital, only single-user access, institution-local, no querying, ...



©: Michael Kohlhasse

387

FAU  
FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG

If we want to improve on – or just digitize **inventory books**, the most obvious idea – at least with what we have learned in IWGS – is to put the data into a database for persistence and use a web application for the user interface. Instead of surveying the multitude existing systems we want to improve on, let us briefly show an example.

### ▷ Cultural Artefacts in Databases: Example

▷ **Example 14.2.3** A typical database for **cultural artefacts**: (HiDa/MIDAS)

HiDa/MIDAS-Datenbank  
Projekt zur Nürnberger Goldschmiedekunst

Freitext: unerschlossene Information

©: Michael Kohlhase 388

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

The system we see above is an instance of the HiDa/MIDAS system, which is in use in many museums for managing their collections. HiDa [HiDa] is a conventional (and commercial) **relational database** with a sophisticated user interface for data acquisition, reporting, exporting, and publication. Database schemata can be chosen from a set of options; here we see the MIDAS schema [BHK16].

This the HiDa/MIDAS system is by no means the only one on the marked, but the architecture is typical for the state of the art and living in most cultural institutions worldwide.

## Cultural Artefacts in Databases: Pro/Con

### ▷ Databases of Cultural Artefacts – Advantages:

- ▷ persistence, multi-user access, structured data,
- ▷ web/catalog publication, standardized exports,
- ▷ standardized performant query language.

### ▷ Databases of Cultural Artefacts – Problems:

- ▷ identifiers are database-local  $\leadsto$  no trans-database relations,
- ▷ database schemata are inflexible  $\Leftarrow$  we need extensions in practice,
- ▷ free text as an un-structured, untapped resource.

- ▷ **Idea:** **Relational databases** impose structure, let's try something very unstructured: the **world wide web**. (up next)



Let us see whether this idea has merit.

## Using the Web for the Cultural Heritage

- ▷ **Idea:** Why not use the [world wide web](#) as a tool?
  - ▷ it is inherently distributed and networked,
  - ▷ the data formats HTML and XML are highly flexible,
  - ▷ gives us instantaneous access to information/images/...,
  - ▷ allows collaboration and discussion. (wikis, fora, blogs)



©: Michael Kohlhase

390



Again, an example is in order to help understand the issues at hand.

## Cultural Artefacts on the Web

- ▷ **Example 14.2.4** A text about a [cultural artefact](#) (an etching by Dürer)

The screenshot shows the Wikipedia page for 'Melencolia I'. The article text describes the engraving as a 1514 work by Albrecht Dürer, depicting a melancholic figure surrounded by various symbolic objects like a magic square, a ladder, and a rainbow. It also mentions the artist's association with melancholia and the work's historical importance.

**Question:** Just how does the etching discussed here relate to Albrecht Dürer?



©: Michael Kohlhase

391



We collect the properties of the various approaches to documenting [cultural artefacts](#) to see how to proceed.

## ▷ Using the Web for Cultural Heritage

- ▷ **Problems:** with using the [Web](#) as a resource
  - ▷ Information is often of dubious quality (imprecise, typos, incomplete, ...)
  - ▷ Information is primarily written for human consumption
    - ▷ ~> not machine-actionable, but full text search works (e.g. Google)

- ▷ sometimes we can use established structures (e.g. Infobox in Wikipedia)
- ▷ **Evaluation:** The web is complementary to databases on the structure-vs-flexibility tradeoff scale for cultural heritage systems. (we need both)
- ▷ **Idea:** Use the semantic web for cultural heritage
  - ▷ **Goal:** Make information accessible for humans and machines
  - ▷ meaning capture by reference to real-world objects
  - ▷ globally unique identifiers of cultural artefacts ( $\cong$  URIs)
  - ▷ inference (get out more than you put in!)



## 14.3 The Semantic Web

In this Section we will introduce the “Semantic Web”. That tries to transform the “World Wide Web” from a human-understandable web of multimedia documents into a “web of machine-understandable data”. In this context, “machine-understandable” means that machines can draw inferences from data they have access to, so that they can make use of the knowledge that is implicit – i.e. not explicitly stated, but can be derived from other information (by humans) – in the web.

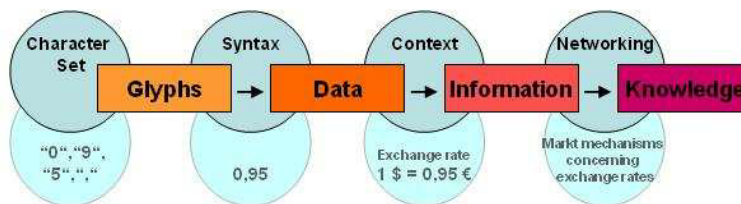
We will now define the term **semantic web** and discuss the pertinent ideas involved. There are two central ones, we will cover here:

- Information and data come in different levels of explicitness; this is usually visualized by a “ladder” of information.
- if information is sufficiently machine-understandable, then we can automate drawing conclusions

### The Semantic Web

- ▷ **Definition 14.3.1** The **semantic web** is a collaborative movement led by the W3C that promotes the inclusion of semantic content in web pages with the aim of converting the current web, dominated by unstructured and semi-structured documents into a machine-understandable “web of data”.

- ▷ **Idea:** Move web content up the ladder, use inference to make connections.



- ▷ **Example 14.3.2** Information that is not explicitly represented (in one place)

**Query:** *Who was US president when Barak Obama was born?*

**Google:** ...*BIRTH DATE: August 04, 1961...*

**Query:** *Who was US president in 1961?*

**Google:** *President: Dwight D. Eisenhower [...] John F. Kennedy (starting January 20)*

Humans understand the text and combine the information to get the answer.  
Machines need more than just text  $\leadsto$  [semantic web](#) technology.



©: Michael Kohlhase

393



The term “Semantic Web” was coined by Tim Berners Lee in analogy to [semantic networks](#), only applied to the world wide web. And as for [semantic networks](#), where we have inference processes that allow us the recover information that is not explicitly represented from the network (here the world-wide-web).

To see that problems have to be solved, to arrive at the “Semantic Web”, we will now look at a concrete example about the “semantics” in web pages. Here is one that looks typical enough.

### What is the Information a User sees?

*WWW2002*

*The eleventh International World Wide Web Conference*

*Sheraton Waikiki Hotel*

*Honolulu, Hawaii, USA*

*7-11 May 2002*

*Registered participants coming from*

*Australia, Canada, Chile Denmark, France, Germany, Ghana, Hong Kong, India,*

*Ireland, Italy, Japan, Malta, New Zealand, The Netherlands, Norway, Singapore, Switzerland, the United Kingdom, the United States, Vietnam, Zaire*

*On the 7th May Honolulu will provide the backdrop of the eleventh International World Wide Web Conference.*

*Speakers confirmed*

*Tim Berners-Lee: Tim is the well known inventor of the Web,*

*Ian Foster: Ian is the pioneer of the Grid, the next generation internet.*



©: Michael Kohlhase

394



But as for semantic networks, what you as a human can see (“understand” really) is deceptive, so let us obfuscate the document to confuse your “semantic processor”. This gives an impression of what the computer “sees”.

### What the machine sees

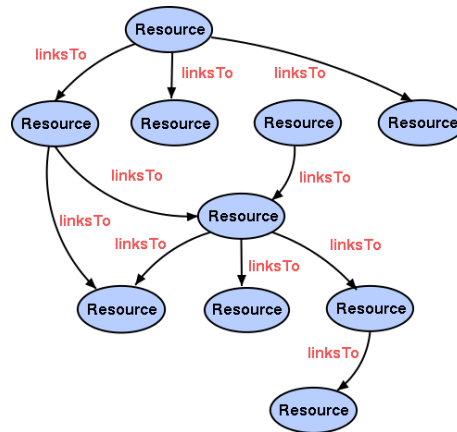
WWW€€€





## The Current Web

- ▷ **Resources:** identified by URI's, untyped
- ▷ **Links:** href, src, ... limited, non-descriptive
- ▷ **User:** Exciting world - semantics of the resource, however, gleaned from content
- ▷ **Machine:** Very little information available - significance of the links only evident from the context around the anchor.



©: Michael Kohlhase

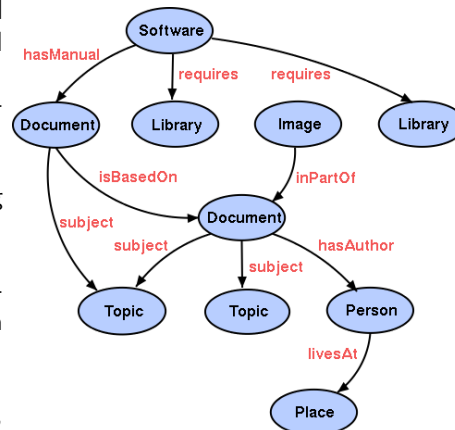
399



Let us now contrast this with the envisioned semantic web.

## The Semantic Web

- ▷ **Resources:** Globally Identified by URI's or Locally scoped (Blank), Extensible, Relational
- ▷ **Links:** Identified by URI's, Extensible, Relational
- ▷ **User:** Even more exciting world, richer user experience
- ▷ **Machine:** More processable information is available (Data Web)
- ▷ **Computers and people:** Work, learn and exchange knowledge effectively



©: Michael Kohlhase

400



Essentially, to make the web more machine-processable, we need to classify the resources by the concepts they represent and give the links a meaning in a way, that we can do inference with that.

The ideas presented here gave rise to a set of technologies jointly called the “semantic web”, which we will now summarize before we return to our logical investigations of knowledge representation techniques.

## Towards a “Machine-Actionable Web”

- ▷ **Recall:** We need external agreement on meaning of annotation tags.

- ▷ **Idea:** standardize them in a community process (e.g. DIN or ISO)
  - ▷ **Problem:** Inflexible, Limited number of things can be expressed
  - ▷ **Better:** Use Ontologies to specify meaning of annotations
    - ▷ Ontologies provide a vocabulary of terms
    - ▷ New terms can be formed by combining existing ones
    - ▷ Meaning (semantics) of such terms is formally specified
    - ▷ Can also specify relationships between terms in multiple ontologies
  - ▷ Inference with annotations and ontologies (get out more than you put in!)
    - ▷ Standardize annotations in RDF [KC04] or RDFa [Her+13b] and ontologies on OWL [OWL09]
    - ▷ Harvest RDF and RDFa in to a triplestore or OWL reasoner.
    - ▷ Query that for implied knowledge (e.g. chaining multiple facts from Wikipedia)
- SPARQL:** Who was US President when Barack Obama was Born?  
**DBPedia:** John F. Kennedy (was president in August 1961)



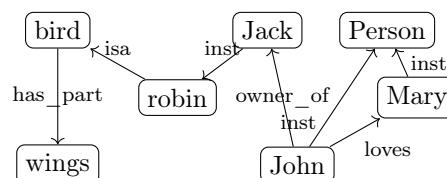
## 14.4 Semantic Networks and Ontologies

To get a feeling for ontologies and how they enable the “machine-actionable web” and how that helps us in DH, we take a look at “semantic networks”, which are an early form of ontologies. They allow us to explain many of the basic functionalities of the “semantic web” without getting too much into details of the technologies involved. We will preview that at the end of this section and go into details in Section 14.6.

Semantic networks are a very simple way of arranging concepts and their relations in a graph.

### Semantic Networks [CQ69]

- ▷ **Definition 14.4.1** A **semantic network** is a **directed graph** for representing knowledge:
  - ▷ **nodes** represent **concepts**, i.e. classes of individuals/objects (e.g. bird, John, robin)
  - ▷ **links** represent relations between these (isa, father\_of, belongs\_to)
- ▷ **Example 14.4.2** A semantic net for birds and persons:



**Problem:** how do we do inference from such a network?

▷ **Idea:** encode taxonomic information about concepts and individuals

- ▷ in “isa” links (inclusion of concepts)
- ▷ in “inst” links (concept memberships)
- ▷ use property inheritance along “isa” and “inst” in the process model

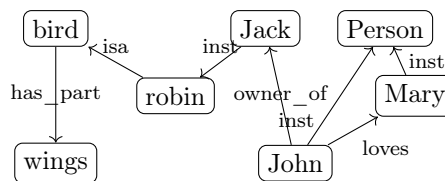


Even though the network in Example 14.4.2 is very intuitive (we immediately understand the concepts depicted), it is unclear how we (and more importantly a machine that does not associate meaning with the labels of the nodes and edges) can draw inferences from the “knowledge” represented.

### Deriving Knowledge Implicit in Semantic Networks

▷ **Observation 14.4.3** *There is more knowledge in a semantic network than is explicitly written down.*

▷ **Example 14.4.4** In the network below, we “know” that *robins have wings* and in particular, *Jack has wings*.



**Idea:** “isa” and “inst” links are special: they propagate properties encoded by other links.



We now make the idea of “propagating properties” rigorous by defining the notion of **derived relations**, i.e. the relations that are left implicit in the network, but can be added without changing its meaning.

### ▷ Deriving Knowledge Semantic Networks

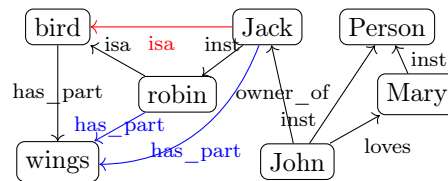
▷ **Definition 14.4.5 (Inference in Semantic Networks)** We call all link labels except “inst” and “isa” in a semantic network **relations**.

Let  $N$  be a semantic network and  $R$  a relation in  $N$  such that  $A \xrightarrow{\text{isa}} B \xrightarrow{R} C$  or  $A \xrightarrow{\text{inst}} B \xrightarrow{R} C$ , then we can **derive** a relation  $A \xrightarrow{R} C$  in  $N$ .

The process of **deriving** new **concepts** and **relations** from existing ones is called **inference** and **conceptss/relations** that are only available via **inference implicit** (in a semantic network).

▷ **Intuition:** **Derived relations** represent knowledge that is implicit in the network; they could be added, but usually are not to avoid clutter.

## ▷ Example 14.4.6



**Slogan:** Get out more knowledge from a semantic networks than you put in.



©: Michael Kohlhase

404



Note that Definition 14.4.5 does not quite allow to **derive** that *Jack is a bird* (did you spot that?), even though we know it is true in the world. This shows us that we that **inference** in semantic networks has to be very carefully defined and may not be “complete”, i.e. there are things that are true in the real world that our **inference** procedure does not capture.

Dually, if we are not careful, then the **inference** procedure might **derive** properties that are not true in the real world – even if all the properties explicitly put into the network are. We call such an **inference** procedure “unsound” or “incorrect”.

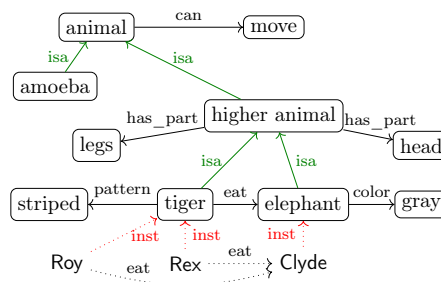
These are two general phenomena we have to keep an eye on.

Another problem is that semantic nets (e.g. in in Example 14.4.2) confuse two kinds of concepts: individuals (represented by proper names like *John* and *Jack*) and concepts (nouns like *robin* and *bird*). Even though the “isa” and “inst” links already acknowledge this distinction, the “has\_part” and “loves” relations are at different levels entirely, but not distinguished in the networks.

## ▷ Terminologies and Assertions

▷ **Remark 14.4.7** We should keep the “inst” and “isa” links apart – and distinguish **concepts** from individuals/objects.

▷ **Example 14.4.8** From the network



infer that *elephants* have *legs* and that *Clyde* is *gray*.

▷ **Definition 14.4.9** We call the subgraph of a semantic network  $N$  spanned by the “isa” relations the **terminology** (or **TBox**, or the famous **Isa-Hierarchy**) and the subgraph spanned by the “inst” relation the **assertions** (or **ABox**) of  $N$ .



©: Michael Kohlhase

405



But there are severe shortcomings of semantic networks: the suggestive shape and node names give (humans) a false sense of meaning, and the inference rules are only given in the process model (the implementation of the semantic network processing system).

This makes it very difficult to assess the strength of the inference system and make assertions e.g. about completeness.

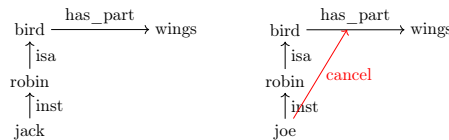
### Limitations of Semantic Networks

▷ What is the meaning of a link?

- ▷ link names are very suggestive (misleading for humans)
- ▷ meaning of link types defined in the process model (no denotational semantics)

**Problem:** No distinction of optional and defining traits

⇒ **Example 14.4.10** Consider a robin that has lost its wings in an accident



Cancel-links have been proposed, but their status and process model are debatable.



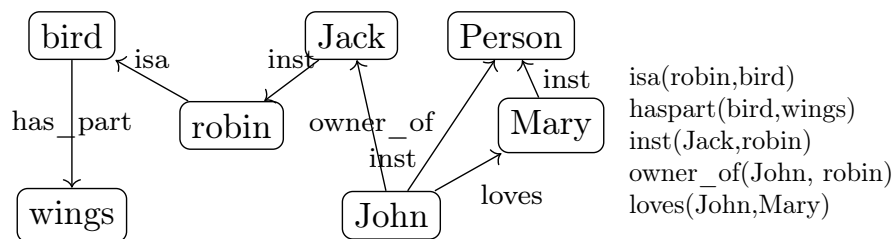
To alleviate the perceived drawbacks of semantic networks, we can contemplate another notation that is more linear and thus more easily implemented: function/argument notation.

### Another Notation for Semantic Networks

▷ **Definition 14.4.11** Function/argument notation for semantic networks

- ▷ interprets node as arguments (reification to individuals)
- ▷ interprets link as functions (logical relations)

▷ **Example 14.4.12**



**Evaluation:**

- ▷ + linear notation (equivalent, but better to implement on a computer)
- + easy to give process model by deduction (e.g. in ProLog)

– worse locality properties	(networks are associative)
©: Michael Kohlhase	407

Indeed the function/argument notation is the immediate idea how one would naturally represent semantic networks for implementation.

This notation has been also characterized as subject/predicate/object triples, alluding to simple (English) sentences. This will play a role in the “semantic web” later.

The next slide is a bit outside of the scope of IWGS, but we want to go into this anyway.

We have been talking about the “procedural model” of a [semantic network](#), which essentially specifies the inference algorithm that [derives](#) new knowledge in a network. There is an alternative to this: we can map the network language – [function/argument notation](#) for networks is an essential step for this – in to a known language with an inference system. We call this kind of a mapping a “denotational semantics”, here into a language called first-order logic.

Building on the function/argument notation from above, we can now give a formal semantics for semantic networks: we translate into first-order logic and use the semantics of that.

### A Denotational Semantics for Semantic Networks

▷ **Extension:** take isa/inst concept/individual distinction into account

$robin \subseteq bird$   
 $haspart(bird, wings)$   
 $Jack \in robin$   
 $owner\_of(John, Jack)$   
 $loves(John, Mary)$

▷ **Observation:** this looks like first-order logic, if we take

- ▷  $a \in S$  to mean  $S(a)$
- ▷  $A \subseteq B$  to mean  $\forall X. A(X) \Rightarrow B(X)$
- ▷  $haspart(A, B)$  to mean  $\forall X. A(X) \Rightarrow (\exists Y. B(Y) \wedge part\_of(X, Y))$

▷ **Idea:** Take first-order deduction as process model (gives inheritance for free)

Indeed, the semantics induced by the translation to first-order logic, gives the intuitive meaning to the semantic networks. Note that this only holds only for the features of semantic networks that are representable in this way, e.g. the cancel links shown above are not (and that is a feature, not a bug).

But even more importantly, the translation to first-order logic gives a first process model: we can use first-order inference to compute the set of inferences that can be drawn from a semantic network.

Based on the intuitions from [semantic networks](#) we can now come to general (Semantic Web) ontologies.

## What is an Ontology

▷ **Definition 14.4.13** An **ontology** is a formal model of (an aspect of) the world. It

- ▷ introduces a **vocabulary** for the **objects**, **concepts**, and **relations** of a given domain,
- ▷ specifies intended meaning of **vocabulary** in a **description logic** using
  - ▷ a set of **axioms** describing structure of the model
  - ▷ a set of **facts** describing some particular concrete situation

The **vocabulary** together with the collection of **axioms** is often called a **terminology** (or **TBox**) and the collection of facts an **ABox** (**assertions**).

In addition to the **represented axioms** and **facts**, the **description logic** determines a number of **derived** ones.

▷ **Definition 14.4.14** A **vocabulary** often includes names for **classes** and **relationships** (also called **concepts**, and **properties**).

▷ **Remark 14.4.15** If the **description logic** has a reasoner, we can automatically

- ▷ detect inconsistent axiom systems
- ▷ compute class membership and taxonomies.



There is a whole collection of standardized languages and interoperable systems that facilitate dealing with (very large) ontologies in practice. We will only give a summary preview here, leaving the detailed discussion to Section 14.6.

## Semantic Web Technology in a Nutshell

▷ **Ontologies** have become one of the standard devices for representing information about the **Web** and the world.

▷ This is facilitated and standardized by the **semantic web technology stack**:

- ▷ **URIs** for representing **objects**,
- ▷ **RDF triples** for representing **facts**,
- ▷ **RDFa** for annotating **RDF triples** in XML documents,
- ▷ **OWL** for representing **TBoxes**,
- ▷ **triple stores** for storing (lots of) **RDF triples**,
- ▷ **SPARQL** for querying **ontologies**,
- ▷ **description logic reasoners** for deciding ontology consistency and concept subsumption,
- ▷ **Protégé** for authoring and maintaining **ontologies**,

▷ Details in Section 14.6.



Indeed, this list can be read as a technology roadmap for the WissKI system. We have already seen the most of the concepts in Section 14.4, we will discuss the technologies in Section 14.6, but first we will have a look at the **CIDOC CRM** ontology that is used in WissKI.

## 14.5 CIDOC CRM: An Ontology for Cultural Heritage

We have seen that databases are not the only choice for representing data about **cultural heritage**. Indeed, the WissKI system chooses **ontologies** as a basis for representation and querying.

To ensure interoperability, WissKI is based on the ISO-standardized **CIDOC CRM** ontology, which we will now introduce and explore.

Now, we can instantiate what we have learned about ontology-based information systems to **cultural heritage** disciplines. We collect all the bits and pieces and hint at the technologies (details in Section 14.6).

### Ontologies for Cultural Artefacts

- ▷ **Idea:** Use **ontologies** for documenting **cultural heritage**.

- ▷ flexible schemata (OWL)
- ▷ easy data sharing
- ▷ open standards, free tools
- ▷ semantic querying via SPARQL

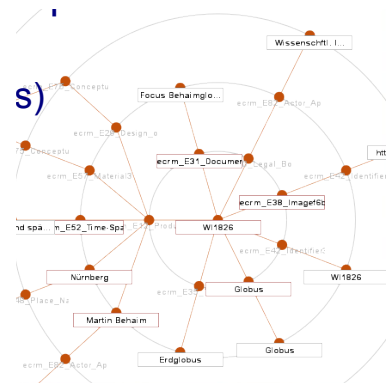
- ▷ **Idea:** We can use RDF like a Mindmap: RDF can

- ▷ represent relations between objects
- ▷ classify objects (web resources)

RDFa for document annotation

- ▷ Reference **ontologies** for interoperability:

- ▷ SUMO (Suggested Upper Model Ontology) [SUMO] for common knowledge,
- ▷ FOAF (Friend-of-a-Friend) [FOAF14] for persons and relations,
- ▷ **CIDOC CRM** for documentation of **cultural heritage**. (up next)



So let us look at the **CIDOC CRM** ontology in more detail. It has been developed by the Documentation Committee of the ICOM (International Council of Museums) over more than 20 years and has been standardized by the ISO. Even more importantly for our purposes here, the **CIDOC CRM** has been implemented in the OWL format, which gives us the use of the **semantic web technology stack**.

## CIDOC CRM (Conceptual Reference Model)

- ▷ **Definition 14.5.1** **CIDOC CRM** provides an extensible ontology for concepts and information in cultural heritage and museum documentation. It is the international standard (ISO 21127:2014) for the controlled exchange of **cultural heritage** information. The central classes include
  - ▷ **space-time** specified by title/identifier, place, era/period, time-span, and relationship to persistent items
  - ▷ **events** specified by title/identifier, beginning/ending of existence, participants (people, either individually or in groups), creation/modification of things (physical or conceptional), and relationship to persistent items
  - ▷ **material things** specified by title/identifier, place, the information object the material thing carries, part-of relationships, and relationship to persistent items
  - ▷ **immaterial things** specified by title/identifier, information objects (propositional or symbolic), conceptional things, and part-of relationships
- ▷ **Definition 14.5.2** **Erlangen CRM/OWL** implements **CIDOC CRM** in OWL

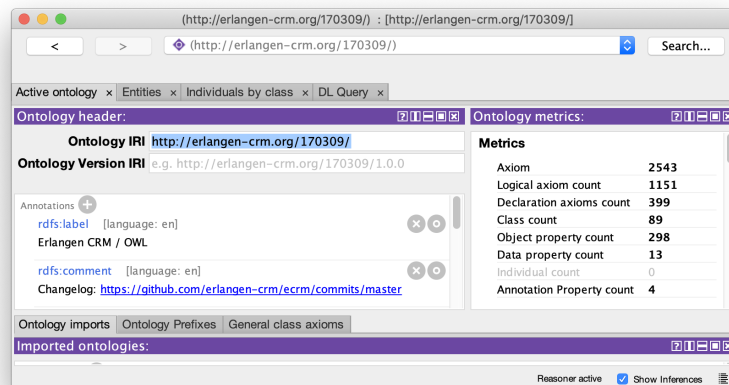
Details about **CIDOC CRM** can be found at [CC] and about **Erlangen CRM/OWL** at [ECRMB; ECRMa].



One of the advantages of having **CIDOC CRM** in OWL is that we can use semantic web technologies to deal with it. Here we use one of the practically most important tools: Protégé.

## ▷ Protege, an IDE for Ontology Development

- ▷ **Definition 14.5.3** Protégé [Pro] is an **integrated development environment** for **ontologies** represented in the OWL family. It comprises
  - ▷ a visual user interface for exploring and editing ontologies,
  - ▷ a **inference** component to ensure **ontology** consistency and minimality,
  - ▷ a facility for querying the loaded ontologies.
- ▷ **Example 14.5.4** (**CIDOC CRM** in Protege)



©: Michael Kohlhase

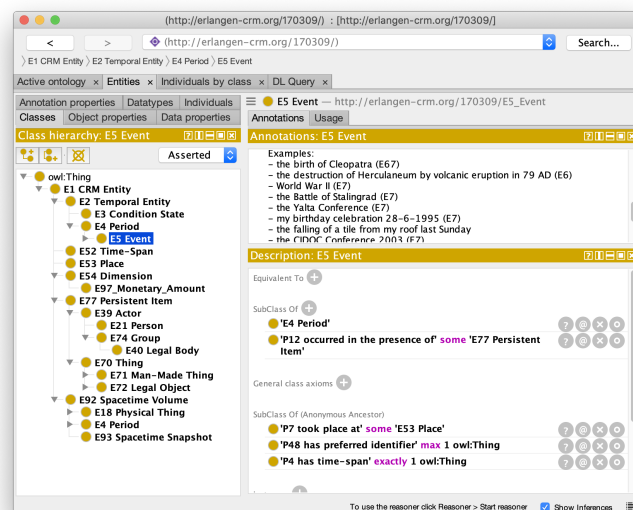
413



The backbone of the [CIDOC CRM](#) ontology is formed by the [concepts](#) (called “classes” in OWL). They form an inheritance hierarchy – of which the top part is shown on the left of the Protégé window below. The ontology provides – usually relatively abstract classes for all objects related to [cultural artefacts](#), their properties, and provenance.

## CIDOC CRM Explored (Classes)

- ▷ [Idea](#): Use semantic web technology to explore [Erlangen CRM/OWL](#).
- ▷ [CIDOC CRM Classes](#): [concept](#)  $\hat{=}$  OWL “Class” (shown in [Protege](#))



©: Michael Kohlhase

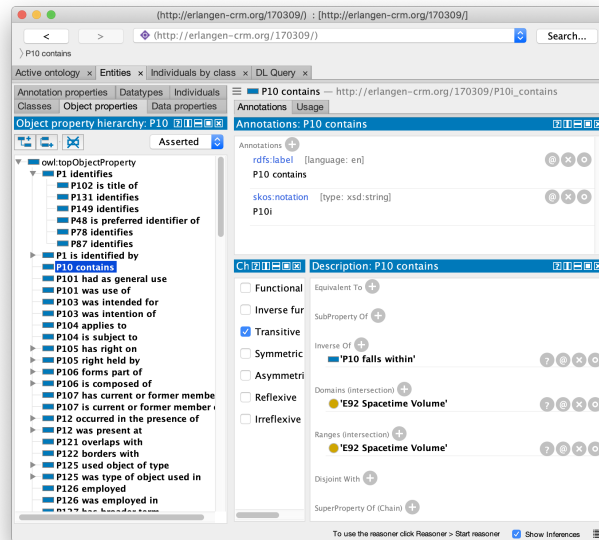
414



The concepts are complemented by the [relations](#) – called “object properties” in OWL.

## CIDOC CRM Explored (Relations)

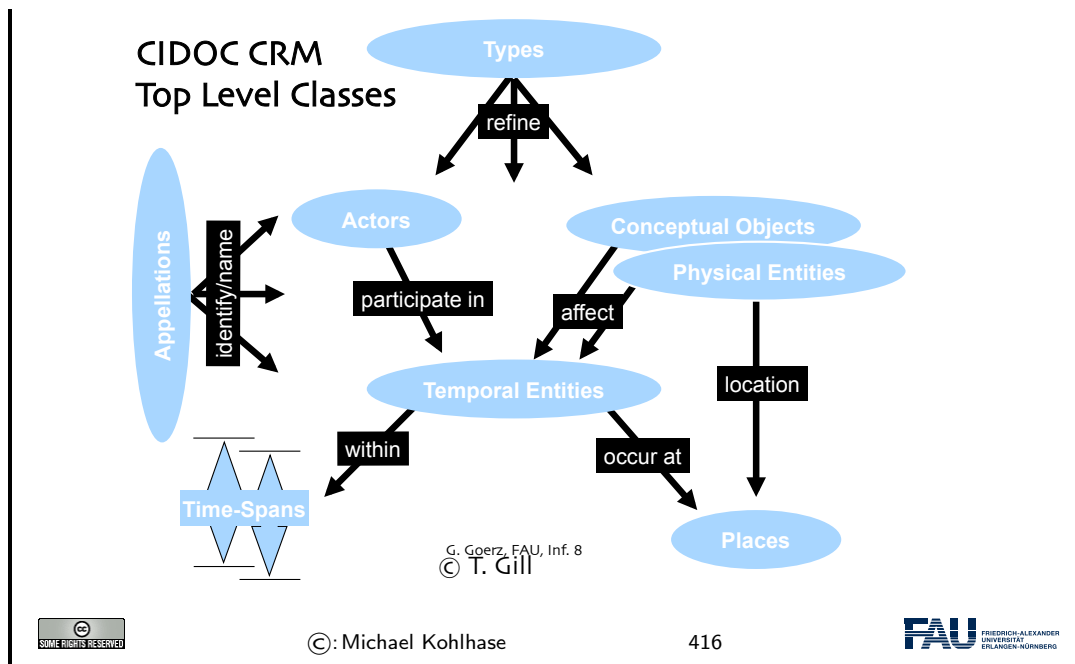
▷ CIDOC CRM Relations: **relation**  $\hat{=}$  OWL “Object Property” (shown in Protege)



There are also a small number of “data properties”, i.e. properties whose values are **concrete data** like numbers, dates, or strings. They are less interesting structurally, but important in practice as Note that there are also “ we will see.

We can summarize the structure of the **CIDOC CRM ontology** in the following diagram.

## CIDOC CRM Structure (Overview)



Now that we understand the [CIDOC CRM ontology](#), we look into the process of modeling [cultural artefacts](#).

## CIDOC-CRM Modeling

▷ This is all good and dandy but how do I concretely model [cultural artefacts](#)?

▷ Answer: [CIDOC CRM](#) is only a [TBox](#), we add an [ABox](#) of [objects](#) and [facts](#).

▷ **Example 14.5.5** *Albrecht Dürer painted Melencolia 1 in Nürnberg*  
We have two units of information here:

1. Albrecht Dürer painted Melencolia 1
2. this happened in the city of Nürnberg

▷ [CIDOC CRM](#) modeling decisions; we start with 1. *AD painted M 1*

1. A painting *m* is an "Information Carrier" (E84)
2. It was created in an "Production Event" *q* (E12)
3. *m* is related to *q* via the "was produced by" relation (P108i)
4. *q* was "carried out by" a "person" *d* (P14 E21)
5. *d* "is identified by" an "actor appellation" *a* (P131 E82)
6. *a* "has note" the string "Albrecht Dürer". (P3)

▷ [CIDOC CRM](#) modeling decisions; continuing with 2. *this happened in N*

1. A painting *m* is an "Information Carrier" (E84)
2. It was created in an "Production Event" *q* (E12)
3. *m* is related to *q* via the "produced by" relation (P108i)
4. *q* was "took place at" a "place" *p* (P7 E53)
5. *p* "is identified by" an "place name" *n* (P48 E3)

6.  $n$  "has note" the string "Nürnberg".

(P3)



©: Michael Kohlhase

417



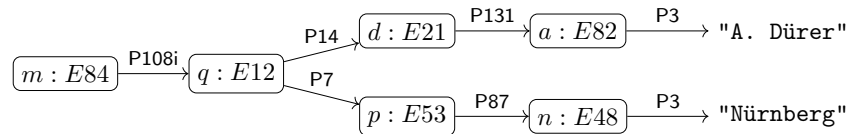
If we look more closely at the objects and relations in Example 14.5.5, we see that

- a typical information unit results in a whole chain of objects connected by ontology relations
- parts of these chains are shared between information units

We address this now and introduce the concept of **ontology groups** and **ontology paths** for that.

### CIDOC CRM Modelling (Ontology Paths)

▷ Modeling *Albrecht Dürer painted Melencolia 1 in Nürnberg* in CIDOC CRM



Note that we need to create the intermediary **objects**  $q$ ,  $d$ ,  $a$ , and  $n$ .

- ▷ **Problem:** That is a lot of work for something very simple.
- ▷ **Definition 14.5.6** We call sequence of facts  $s_i \xrightarrow{p_i} o_i$ , where  $s_i = o_{i-1}$  an **ontology path** and any subtree an **ontology group**.
- ▷ **Problem Reformulated:** A simple statement like *Albrecht Dürer painted Melencolia 1* becomes a whole **ontology path** in CIDOC CRM.
- ▷ **But:** we can reuse intermediary **objects** and **facts**, and need fine-grained models for flexibility.
- ▷ **Idea:** Maybe systems can take some of the pain out of modeling. ( $\leadsto$  WissKI)



©: Michael Kohlhase

418



In Example 14.5.5, we have already seen one of the peculiarities of modeling complex situations in **ontologies**: the use of events as intermediate objects. This is a general phenomenon when modeling with ontologies, which we have to get used. to

### Event-Oriented Modeling in CIDOC CRM

- ▷ **Observation 14.5.7** **Ontologies** make it easy to model facts with transitive verbs, e.g. *Albrecht Dürer created Melencolia 1* (**binary relation**)
- ▷ **Problem:** What about more complex situations with more arguments? E.g.
  1. *Albrecht Dürer created Melencolia 1 with an etching needle* (**ternary**)

2. *Albrecht Dürer* created *Melencolia 1* with an etching needle in *Nürnberg*  
(four arguments)
3. *Albrecht Dürer* created *Melencolia 1* with an etching needle in *Nürnberg*  
out of *boredom* (five)

▷ **Standard Solution:** Introduce “events” tied to the verb and describe those

▷ **Example 14.5.8** There was a creation event  $e$  with

1. *Albrecht Dürer* as the agent,
2. *Melencolia 1* as the product,
3. *an etching needle* as the means,
4. *boredom* as the reason,

**Consequence:** More than 1/3 of **CIDOC CRM** classes are events of some kind.



©: Michael Kohlhase

419



This “event-oriented” thinking is unfamiliar at first and takes practice to become natural. As a rule of thumb one should proceed as in the Melencolia example above. We first identify the “participants” in the situation, if these are more than two, we need to introduce an appropriate event (select from the ones provided by **CIDOC CRM**) and then connect the event to the object currently under consideration, and all the “participants” to the event.

## 14.6 The Semantic Web Technology Stack

In this Section we discuss how we can apply description logics in the real world, in particular, as a conceptual and algorithmic basis of the “Semantic Web”. That tries to transform the “World Wide Web” from a human-understandable web of multimedia documents into a “web of machine-understandable data”. In this context, “machine-understandable” means that machines can draw inferences from data they have access to.

Note that the discussion in this digression is not a full-blown introduction to RDF and OWL, we leave that to [SR14; Her+13a; Hit+12] and the respective W3C recommendations. Instead we introduce the ideas behind the mappings from a perspective of the description logics we have discussed above.

The most important component of the “Semantic Web” is a standardized language that can represent “data” about information on the Web in a machine-oriented way.

### ▷ Resource Description Framework

▷ **Definition 14.6.1** The **Resource Description Framework** (RDF) is a framework for describing resources on the web. It is an XML vocabulary developed by the W3C.

▷ **Note:** RDF is designed to be read and understood by computers, not to be being displayed to people. (it shows)

▷ **Example 14.6.2** RDF can be used for describing (all “objects on the WWW”) (all “objects on the WWW”)

- ▷ properties for shopping items, such as price and availability
- ▷ time schedules for web events
- ▷ information about web pages (content, author, created and modified date)
- ▷ content and rating for web pictures
- ▷ content for search engines
- ▷ electronic libraries



Note that all these examples have in common that they are about “objects on the Web”, which is an aspect we will come to now.

“Objects on the Web” are traditionally called “resources”, rather than defining them by their intrinsic properties – which would be ambitious and prone to change – we take an external property to define them: everything that has a URI is a web resource. This has repercussions on the design of RDF.

### Resources and URIs

- ▷ RDF describes resources with properties and property values.
- ▷ RDF uses Web identifiers (URIs) to identify resources.
- ▷ **Definition 14.6.3** A **resource** is anything that can have a URI, such as `http://www.fau.de`
- ▷ **Definition 14.6.4** A **property** is a resource that has a name, such as *author* or *homepage*, and a **property value** is the value of a property, such as *Michael Kohlhase* or `http://kwarc.info/kohlhase` (a property value can be another resource)
- ▷ **Definition 14.6.5** A RDF **statement**  $s$  (also known as a **triple**) consists of a resource (the **subject** of  $s$ ), a **property** (the **predicate** of  $s$ ), and a **property value** (the **object** of  $s$ ). A set of RDF **triples** is called an **RDF graph**.
- ▷ **Example 14.6.6** Statement: *[This slide]<sup>subj</sup> has been [author]<sup>pred</sup>ed by [Michael Kohlhase]<sup>obj</sup>*



The crucial observation here is that if we map “subjects” and “objects” to “individuals”, and “predicates” to “relations”, the RDF statements are just relational ABox statements of description logics. As a consequence, the techniques we developed apply.

We now come to the concrete syntax of RDF. This is a relatively conventional XML syntax that combines RDF statements with a common subject into a single “description” of that resource.

### XML Syntax for RDF

- ▷ RDF is a concrete XML vocabulary for writing statements
- ▷ **Example 14.6.7** The following RDF document could describe the slides as a resource

```

<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description about="https://.../CompLog/kr/en/rdf.tex">
    <dc:creator>Michael Kohlhase</dc:creator>
    <dc:source>http://www.w3schools.com/rdf</dc:source>
  </rdf:Description>
</rdf:RDF>

```

This RDF document makes two statements:

- ▷ The subject of both is given in the `about` attribute of the `rdf:Description` element
- ▷ The predicates are given by the element names of its children
- ▷ The objects are given in the elements as URIs or literal content.

**Intuitively:** RDF is a web-scalable way to write down ABox information.



©: Michael Kohlhase

422



Note that XML namespaces play a crucial role in using element to encode the predicate URIs. Recall that an element name is a qualified name that consists of a namespace URI and a proper element name (without a colon character). Concatenating them gives a URI in our example the predicate URI induced by the `dc:creator` element is `http://purl.org/dc/elements/1.1/creator`. Note that as URIs go RDF URIs do not have to be URLs, but this one is and it references (is redirected to) the relevant part of the Dublin Core elements specification [DCM12].

RDF was deliberately designed as a standoff markup format, where URIs are used to annotate web resources by pointing to them, so that it can be used to give information about web resources without having to change them. But this also creates maintenance problems, since web resources may change or be deleted without warning.

RDFa gives authors a way to embed RDF triples into web resources and make keeping RDF statements about them more in sync.

### ▷ RDFa as an Inline RDF Markup Format

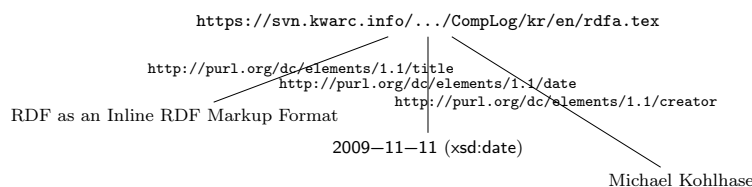
- ▷ **Problem:** RDF is a standoff markup format (annotate by URIs pointing into other files)

#### ▷ Example 14.6.8

```

<div xmlns:dc="http://purl.org/dc/elements/1.1/" id="address">
  <h2 about="#address" property="dc:title">RDF as an Inline RDF Markup Format</h2>
  <h3 about="#address" property="dc:creator">Michael Kohlhase</h3>
  <em about="#address" property="dc:date" datatype="xsd:date"
    content="2009-11-11">November 11., 2009</em>
</div>

```





In the example above, the `about` and `property` attribute are reserved by RDFa and specify the subject and predicate of the RDF statement. The object consists of the body of the element, unless otherwise specified e.g. by the `resource` attribute.

Let us now come back to the fact that RDF is just an XML syntax for ABox statements.

## RDF as an ABox Language for the Semantic Web

▷ **Idea:** RDF triples are ABox entries  $hRs$  or  $h : \varphi$ .

▷ **Example 14.6.9**  $h$  is the resource for Ian Horrocks,  $s$  is the resource for Ulrike Sattler,  $R$  is the relation “hasColleague”, and  $\varphi$  is the class `foaf:Person`

```
<rdf:Description about="some.uri/person/ian_horrocks">
  <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
  <hasColleague resource="some.uri/person/uli_sattler"/>
</rdf:Description>
```

**Idea:** Now, we need a similar language for TBoxes

(based on *ALC*)



In this situation, we want a standardized representation language for TBox information; OWL does just that: it standardizes a set of knowledge representation primitives and specifies a variety of concrete syntaxes for them. OWL is designed to be compatible with RDF, so that the two together can form an ontology language for the web.

## ▷ OWL as an Ontology Language for the Semantic Web

▷ **Task:** Complement RDF (ABox) with a TBox language.

▷ **Idea:** Make use of resources that are values in `rdf:type` (called *Classes*)

▷ **Definition 14.6.10** OWL (the *ontology web language*) is a language for encoding TBox information about RDF classes.

▷ **Example 14.6.11 (A concept definition for “Mother”)**

Mother = Woman  $\sqcap$  Parent is represented as

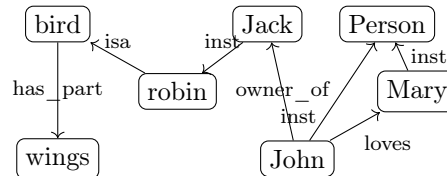
XML Syntax	Functional Syntax
<pre>&lt;EquivalentClasses&gt;   &lt;Class IRI="Mother"/&gt;   &lt;ObjectIntersectionOf&gt;     &lt;Class IRI="Woman"/&gt;     &lt;Class IRI="Parent"/&gt;   &lt;/ObjectIntersectionOf&gt; &lt;/EquivalentClasses&gt;</pre>	<pre>EquivalentClasses(   :Mother   ObjectIntersectionOf(     :Woman     :Parent   ) )</pre>



But there are also other syntaxes in regular use. We show the **functional syntax** which is inspired by the mathematical notation of relations.

### Extended OWL Example in Functional Syntax

- ▷ **Example 14.6.12** The **semantic network** from Example 14.4.4 can be expressed (in **functional syntax**) in OWL



```
ClassAssertion ( :Jack :robin)
ClassAssertion( :John :person)
ClassAssertion ( :Mary :person)
ObjectPropertyAssertion( :loves :John :Mary)
ObjectPropertyAssertion( :owner :John :Jack)
SubClassOf( :robin :bird)
SubClassOf ( :bird ObjectSomeValuesFrom( :hasPart :wing))
```

- ▷ ClassAssertion formalizes the “inst” relation,
- ▷ ObjectPropertyAssertion formalizes **relations**,
- ▷ SubClassOf formalizes the “isa” relation,
- ▷ for the “has\_part” relation, we have to specify that *all birds have a part that is a wing* or equivalently *the class of birds is a subclass of all objects that have some wing*.



We have introduced the ideas behind using description logics as the basis of a “machine-oriented web of data”. While the first OWL specification (2004) had three sublanguages “OWL Lite”, “OWL DL” and “OWL Full”, of which only the middle was based on description logics, with the OWL2 Recommendation from 2009, the foundation in description logics was nearly universally accepted.

The Semantic Web hype is by now nearly over, the technology has reached the “plateau of productivity” with many applications being pursued in academia and industry. We will not go into these, but briefly introduce one of the tools that make this work.

### SPARQL an RDF Query language

- ▷ **Definition 14.6.13** SPARQL, the “SPARQL Protocol and RDF Query Language” is an RDF query language, able to retrieve and manipulate data stored in RDF. The SPARQL language was standardized by the World Wide Web Consortium in 2008 [PS08].
- ▷ SPARQL is pronounced like the word “sparkle”.
- ▷ **Definition 14.6.14** A system is called a **SPARQL endpoint**, iff it answers

SPARQL queries.

▷ **Example 14.6.15**

Query for person names and their e-mails from a triple store with FOAF data.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?email
WHERE {
  ?person a foaf:Person.
  ?person foaf:name ?name.
  ?person foaf:mbox ?email.
}
```



©: Michael Kohlhase

427



SPARQL end-points can be used to build interesting applications, if fed with the appropriate data. An interesting – and by now paradigmatic – example is the DBpedia project, which builds a large ontology by analyzing Wikipedia fact boxes. These are in a standard HTML form which can be analyzed e.g. by regular expressions, and their entries are essentially already in triple form: The **subject** is the Wikipedia page they are on, the **predicate** is the key, and the object is either the URI on the object value (if it carries a link) or the value itself.

## SPARQL Applications: DBpedia


- ▷ **Typical Application:** DBpedia screen-scrapes Wikipedia fact boxes for RDF triples and uses SPARQL for querying the induced triple store.

▷ **Example 14.6.16 (DBpedia Query)**

People who were born in Erlangen before 1900  
(<http://dbpedia.org/snorql>)

```
SELECT ?name ?birth ?death ?person WHERE {
  ?person dbo:birthPlace :Erlangen .
  ?person dbo:birthDate ?birth .
  ?person foaf:name ?name .
  ?person dbo:deathDate ?death .
  FILTER (?birth < "1900-01-01"^^xsd:date)
}
ORDER BY ?name
```

- ▷ The answers include Emmy Noether and Georg Simon Ohm.

Emmy Noether	
	
<b>Born</b>	Amalie Emmy Noether 23 March 1882 <a href="#">Erlangen, Bavaria, German Empire</a>
<b>Died</b>	14 April 1935 (aged 53) <a href="#">Bryn Mawr, Pennsylvania, United States</a>
<b>Nationality</b>	German
<b>Alma mater</b>	<a href="#">University of Erlangen</a>
<b>Known for</b>	<a href="#">Abstract algebra</a> <a href="#">Theoretical physics</a> <a href="#">Noether's theorem</a>



©: Michael Kohlhase

428



## A more complex DBpedia Query

- ▷ **Demo:** DBpedia <http://dbpedia.org/snorql/>

**Query:** Soccer players born in a country with more than 10 M inhabitants, who play as goalie in a club that has a stadium with more than 30.000 seats.

**Answer:** computed by DBpedia from a SPARQL query

```

SELECT distinct ?soccerplayer ?countryOfBirth ?team ?countryOfTeam ?stadiumcapacity
{
  ?soccerplayer a dbo:SoccerPlayer ;
    dbo:position [dbp:position <http://dbpedia.org/resource/Goalkeeper_(association_football)> ;
    dbo:birthPlace/dbo:country* ?countryOfBirth ;
    #dbo:number 13 ;
    dbo:team ?team .
    ?team dbo:capacity ?stadiumcapacity ; dbo:ground ?countryOfTeam .
    ?countryOfBirth a dbo:Country ; dbo:populationTotal ?population .
    ?countryOfTeam a dbo:Country .
  FILTER (?countryOfTeam != ?countryOfBirth)
  FILTER (?stadiumcapacity > 30000)
  FILTER (?population > 1000000)
}
order by ?soccerplayer

```

Results:

SPARQL results:

soccerplayer	countryOfBirth	team	countryOfTeam	stadiumcapacity
Abdellam_Benabdellah	Algeria	Wydad_Casablanca	Morocco	67000
Airton_Moraes_Michelson	Brazil	FC_Red_Bull_Salzburg	Austria	31000
Alain_Gouaméné	Ivory_Coast	Raja_Casablanca	Morocco	67000
Allan_McGregor	United_Kingdom	Beşiktaş_J.K.	Turkey	41903
Anthony_Scribe	France	FC_Dinamo_Tbilisi	Georgia_(country)	54549
Brahim_Zaari	Netherlands	Raja_Casablanca	Morocco	67000
Bréiner_Castillo	Colombia	Deportivo_Táchira	Venezuela	38755
Carlos_Luis_Morales	Ecuador	Club_Atlético_Independiente	Argentina	48069
Carlos_Navarro_Montoya	Colombia	Club_Atlético_Independiente	Argentina	48069
Cristián_Muñoz	Argentina	Colo-Colo	Chile	47000
Daniel_Ferreira	Argentina	FBC_Melgar	Peru	60000
David_Bičik	Czech_Republic	Karşıyaka_S.K.	Turkey	51295
David_Loria	Kazakhstan	Karşıyaka_S.K.	Turkey	51295
Denys_Boyko	Ukraine	Beşiktaş_J.K.	Turkey	41903
Eddie_Gustafsson	United_States	FC_Red_Bull_Salzburg	Austria	31000
Emilian_Dolha	Romania	Lech_Poznań	Poland	43269
Eusebio_Acasuzo	Peru	Club_Bolivar	Bolivia	42000
Faryd_Mondragón	Colombia	Real_Zaragoza	Spain	34596
Faryd_Mondragón	Colombia	Club_Atlético_Independiente	Argentina	48069
Federico_Vilar	Argentina	Club_Atlas	Mexico	54500
Fernando_Martinuzzi	Argentina	Real_Garcilaso	Peru	45000
Fábio_André_da_Silva	Portugal	Servette_FC	Switzerland	30084
Gerhard_Tremmel	Germany	FC_Red_Bull_Salzburg	Austria	31000
Gift_Muzadzi	United_Kingdom	Lech_Poznań	Poland	43269
Günay_Güvenç	Germany	Beşiktaş_J.K.	Turkey	41903
Hugo_Marques	Portugal	C.D._Primeiro_de_Agosto	Angola	48500
Héctor_Landazuri	Colombia	La_Paz_F.C.	Bolivia	42000

We conclude our survey of the [semantic web technology stack](#) with the notion of a [triplestore](#), which refers to the database component, which stores vast collections of ABox [triples](#).

## Triple Stores: the Semantic Web Databases

▷ **Definition 14.6.17** A [triplestore](#) or [RDF store](#) is a purpose-built database for the storage [RDF graphs](#) and retrieval of [RDF triples](#) usually through variants of SPARQL.

▷ Common [triple stores](#) include

- ▷ Virtuoso: <https://virtuoso.openlinksw.com/> (used in DBpedia)
- ▷ GraphDB: <http://graphdb.ontotext.com/> (often used in WissKI)
- ▷ blazegraph: <https://blazegraph.com/> (open source; used in WikiData)

## 14.7 Ontologies vs. Databases

To understand [ontologies](#) better and contrast them to [database systems](#) to understand their respective possible role in documenting [cultural artefacts](#). We start off with a definition of the concept and components of an [ontology](#).

We will still keep our presentation of the material at a general level without committing to a particular ontology language or system.

We now consolidate our understanding of all these concepts with an example. We build an ontology by first constructing a [TBox](#) and then a corresponding [ABox](#).

### Example: Hogwarts Ontology

- ▷ **Example 14.7.1** [Axioms](#) describe the structure of the world,

Class HogwartsStudent = Student and attendsSchool Hogwarts  
 Class: HogwartsStudent  $\sqsubseteq$  hasPet only (Owl or Cat or Toad)  
 ObjectProperty: hasPet Inverses: isPetOf  
 Class: Phoenix  $\sqsubseteq$  isPetOf only Wizard

- ▷ **Example 14.7.2** [Facts](#) describe some particular concrete situation,

Individual: Hedwig  
 Types: Owl  
 Individual: HarryPotter  
 Types: HowgwartsStudent  
 Facts: hasPet Hedwig  
 Individual: Fawkes  
 Types: Phoenix  
 Facts: isPetOf Dumbledore



It is very instructive to compare [ontologies](#) to [databases](#). There are some similarities induced by the joint intention to represent structured data, but also some important differences, which will play a crucial role in our discussion later on.

### Ontologies vs. Databases

- ▷ **Obvious Analogy:** In an [ontology](#):

- ▷ [axioms](#) analogous to DB schema (structure and constraints on data)
- ▷ [facts](#) analogous to DB data
  - ▷ data instantiates schema, is consistent with schema constraints

- ▷ **But there are also important differences:**

Database:	Ontology:
<ul style="list-style-type: none"> <li>▷ <b>Closed world assumption (CWA)</b> <ul style="list-style-type: none"> <li>▷ Missing information treated as false</li> </ul> </li> <li>▷ <b>Unique name assumption (UNA)</b> <ul style="list-style-type: none"> <li>▷ Each individual has a single, unique name</li> </ul> </li> <li>▷ Schema behaves as constraints on structure of data</li> <li>▷ Define legal database states</li> </ul>	<ul style="list-style-type: none"> <li>▷ <b>Open world assumption (OWA)</b> <ul style="list-style-type: none"> <li>▷ Missing information treated as unknown</li> </ul> </li> <li>▷ <b>No UNA</b> <ul style="list-style-type: none"> <li>▷ Individuals may have more than one name</li> </ul> </li> <li>▷ Ontology axioms behave like implications (inference rules)</li> <li>▷ Entail implicit information</li> </ul>



©: Michael Kohlhase

432



Let us elucidate these quite abstract concepts and differences using a simple example, which we again take from the Hogwarts ontology (see fallback=above).

### DB vs. Ontology by Example (Querying)

#### ▷ Given the Ontology:

Individual: HarryPotter

Facts: hasFriend RonWeasley

hasFriend HermioneGranger

hasPet Hedwig

Individual: Draco Malfoy

#### ▷ Query: Is Draco Malfoy a friend of HarryPotter?

▷ DB: No

▷ Ontology: Don't Know (OWA: didn't say Draco was not Harry's friend)

#### ▷ Counting Query: How many friends does Harry Potter have?

▷ DB: 2

▷ Ontology: at least 1 (No UNA: Ron and Hermione may be 2 names for same person)

#### ▷ How about: if we add

DifferentIndividuals: RonWeasley HermioneGranger

▷ Ontology: at least 2 (OWA: Harry may have more friends we didn't mention yet)

#### ▷ And: if we also add

Individual: HarryPotter

Types: hasFriend only RonWeasley or HermioneGranger

- ▷ Ontology: 2



We continue our example with the behavior if we insert new information to the Hogwarts ontology. Again, databases and ontology systems react differently.

### DB vs. Ontology by Example (Insertion)

- ▷ **Given:** the ontology from Example 14.7.1 and Example 14.7.2 insert

Individual: Dumbledore

Individual: Fawkes

Types: Phoenix

Facts: isPetOf Dumbledore

- ▷ **System Response:**

- ▷ DB: Update rejected: constraint violation

- ▷ Range of hasPet is Human; Dumbledore is not (**CWA**)

- ▷ Ontology Reasoner:

- ▷ Infer that Dumbledore is Human

- ▷ Also infer that Dumbledore is a Wizard (**only a Wizard can have a phoenix as a pet**)



Finally, we come to one of the central disciplines in which to compare databases and ontology-based information systems: query answering. Here we see a crucial difference: ontology queries are **semantic**, i.e. they take both **axioms** and **facts** into account.

### DB vs. Ontology by Example: Query Answering

- ▷ DB schema plays no role in query answering (**efficiently implementable**)

- ▷ Ontology axioms play a powerful and crucial role in QA

- ▷ Answer may include implicitly derived facts

- ▷ Can answer conceptual as well as extensional queries

E.g., **Can a Muggle have a Phoenix for a pet?**

- ▷ May have very high worst case complexity (**≅ terrible runtimes**)  
Implementations may still behave well in typical cases.

- ▷ **Definition 14.7.3** We call a query language **semantic**, iff query answering involves **derived axioms** and **facts**.

- ▷ **Observation 14.7.4** *Ontology queries are **semantic**, while database queries are not.*



We will now summarize what we have learned about ontology-based information systems.

### Summary: Ontology Based Information Systems

- ▷ Analogous to relational database management systems  
Ontology  $\hat{=}$  schema; instances  $\hat{=}$  data
- ▷ Some important (dis)advantages
  - + (Relatively) easy to maintain and update schema.
    - ▷ Schema plus data are integrated in a logical theory.
  - + Query answers reflect both schema and data
  - + Can deal with incomplete information
  - + Able to answer both intensional and extensional queries
  - Semantics may be counter-intuitive or even inappropriate
    - ▷ Open -vs- closed world; axioms -vs- constraints.
  - Query answering much more difficult. (based on logical entailment)
    - ▷ Can lead to scalability problems.

In a nutshell they deliver more valuable answers at cost of efficiency.





## Chapter 15

# The WissKI System: A Virtual Research Environment for Cultural Heritage

We will now come to the WissKI system itself, which positions itself as a virtual research environment for cultural heritage. Indeed it is a comprehensive, ontology-based information system for documenting, studying, and presenting our [cultural heritage](#).

Before we go into the technicalities of the WissKI system itself, let us recall the requirements and motivations.

### ▷ WissKI: a Virtual Research Env. for Cultural Heritage

▷ **Definition 15.0.1** WissKI is a virtual research environment (VRE) for managing scholarly data and documenting [cultural heritage](#).

▷ **Requirements:** For a virtual research environment for [cultural heritage](#), we need

- ▷ scientific communication about and documentation of the [cultural heritage](#)
- ▷ networking knowledge from different disciplines (transdisciplinarity)
- ▷ high-quality data acquisition and analysis
- ▷ safeguarding authorship, authenticity, persistence
- ▷ support of scientific publication

▷ WissKI was developed by the research group of Prof. Günther Görtz at FAU Erlangen-Nürnberg and is now used in hundreds of DH projects across Germany.

▷ FAU supports [cultural heritage](#) research by providing hosted WissKI instances.

- ▷ See <https://wisski.agfd.fau.de> for details
- ▷ We will use an instance for the Kirmes paintings in the homework assignments

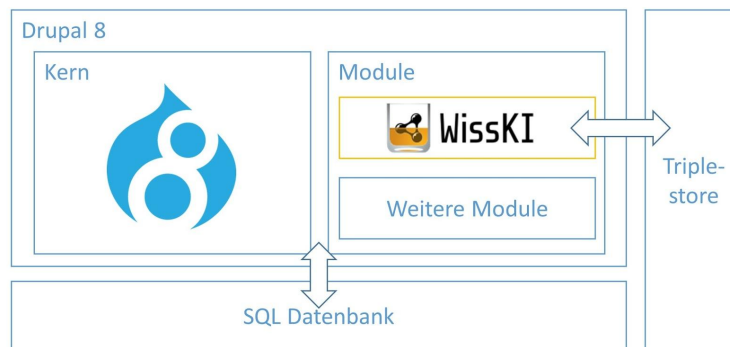


## 15.1 WissKI extends Drupal

The first thing about the WissKI system is that it is realized as an extension of the [Drupal web content management system](#), which already provides many of the features (e.g. user management, web authoring, collaboration, ...) a VRE needs to implement.

### WissKI System Architecture

- ▷ Software basis: [Drupal CMS](#) ([content management system](#))
  - ▷ large, active community, extensible by [Drupal modules](#)
  - ▷ provides much of the functionality of a VRE out of the box.

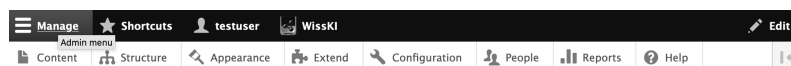


We now give a general overview of the [Drupal](#) system, and introduce the concepts we need for understanding WissKI system. Naturally, this does now do the [Drupal WCMS](#) justice. For an introduction we refer readers to [Gla17; Tom17] and the Drupal web site [Dru].

### Drupal: A Web Content Management Framework

- ▷ **Definition 15.1.1** [Drupal](#) is an [open source web content management application](#). It combines [CMS](#) functionality with knowledge management via [RDF](#).
- ▷ **Definition 15.1.2** [Drupal](#) allows to configure web pages modularly from content [blocks](#), which can be
  - ▷ [static content](#), i.e. supplied by a module,
  - ▷ [user-supplied content](#), or
  - ▷ [views](#), i.e. listings of content fragments from other [blocks](#).

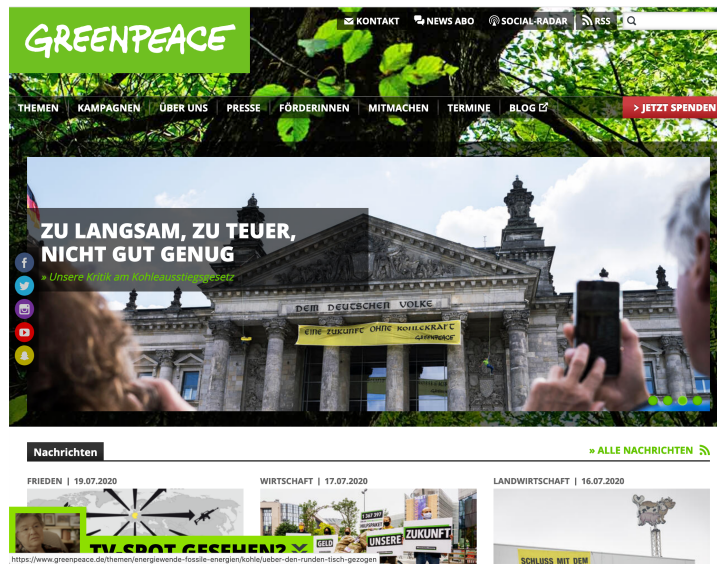
These can be assembled into web pages via a visual interface: the [config bar](#).



To fortify our intuition about the concepts introduced above, let us try to find them in an existing web page.

## Assembling a Web Site via Drupal Blocks (Example)

▷ **Example 15.1.3 (Greenpeace via Drupal)** Can you find the blocks?



We now come to one of the most important features used in WissKI: **Drupal** is modular and extensible; this allows us to build the features for an ontology-based information system as **Drupal modules**.

## Drupal Modules and Themes

- ▷ **Idea:** **Drupal** is designed to be modular and extensible (so it can adapt to the ever-changing web)
- ▷ **Definition 15.1.4 (Modular Design)** **Drupal** functionality is structured into
  - ▷ **Drupal core**– the basic CMS functionality
  - ▷ **modules**– which contribute e.g. new block types (~ 45.000)
  - ▷ **themes**– which contribute new UI layouts (~ 2800)

**Drupal core** is the vanilla system as downloaded, **modules** and **themes** must be installed and configured separately via the **config bar**.

- ▷ The **Drupal core** functionalities include
  - ▷ user/account management
  - ▷ menu management,
  - ▷ RSS feeds,

- ▷ taxonomy,
- ▷ page layout customization (via [blocks](#) and [views](#)),
- ▷ system administration



This brings us to the central data acquisition subsystem in [drupal](#), which we will use to build our system. Much of the actual data in the [drupal](#) system is internally stored in terms of [dictionaries](#): systems of [key/value](#) pairs.

### Bundles and Fields in Drupal (Data Entry)

- ▷ **Definition 15.1.5** [Drupal](#) has a special data type called a [bundle](#), which is essentially a [dictionary](#): it contains [key/value](#) pairs called [fields](#).
  - ▷ [bundles](#) can be nested  $\leadsto$  sub-bundles.
  - ▷ [fields](#) also have data type information, etc. to support editing.
- ▷ [drupal](#) presents [bundles](#) as
  - ▷ HTML lists for reading
  - ▷ HTML forms for data entry/editing
- ▷ [Drupal bundles](#) induce [blocks](#) that can be used for data entry and presentation.

▼ Object

Inventory number: \*

Collection:

Title:

▼ Creation

Artist: Albrecht Dürer

Date:

Place: Nürnberg

Mat./Tech.:

Inscription:

Iconography:

Literature:

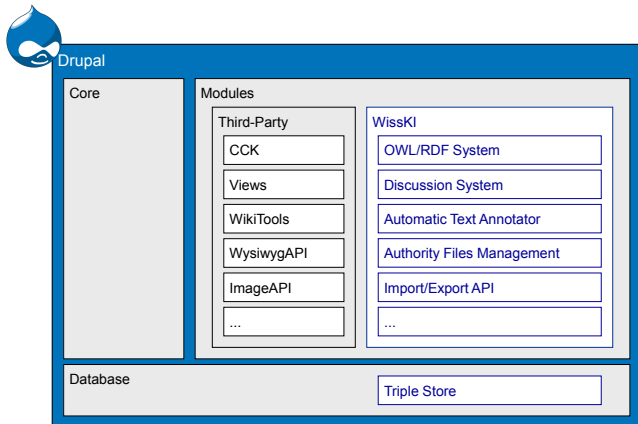
Images:



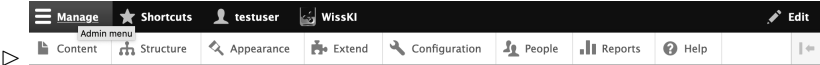
Now we can summarize the WissKI architecture in a simple equation. While this glosses over many of finer points in the system, it is important to keep this in mind for working with the system in practice.

### WissKI System Architecture (Recap)

- ▷  $\text{WissKI} = \text{drupal} + \text{CIDOC CRM} + \text{triple store} + \text{WissKI modules}$



**Note:** Much of WissKI functionality is configurable via the [drupal config bar](#).



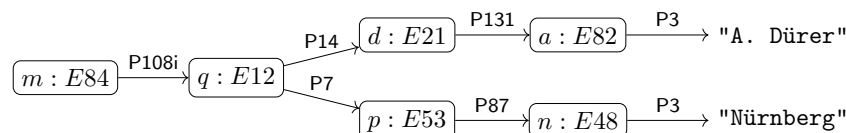
©: Michael Kohlhase 443 FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

## 15.2 Dealing with Ontology Paths: The WissKI Pathbuilder

We now come to what is probably the defining feature of WissKI: the WissKI [path builder](#). It solves the problem that with ontologies, even for simple facts we have to generate entire [ontology path](#).

### The WissKI Path Builder (Idea)

▷ **Recall:** *Albrecht Dürer painted Melencolia 1 in Nürnberg*



▷ **Idea:** Hide the complexity induced by the ontology from the user

▷ Form-based interaction with categories and fields (as in a RDBMS UI)

▷ **Definition 15.2.1** The WissKI [path builder](#) maps [ontology groups](#) and [ontology paths](#) to [drupal bundles](#) and [fields](#).

▷ [ontology groups](#) become data entry forms ([bundles](#)) for the root entities,

▷ their [fields](#) are mapped to [ontology paths](#).

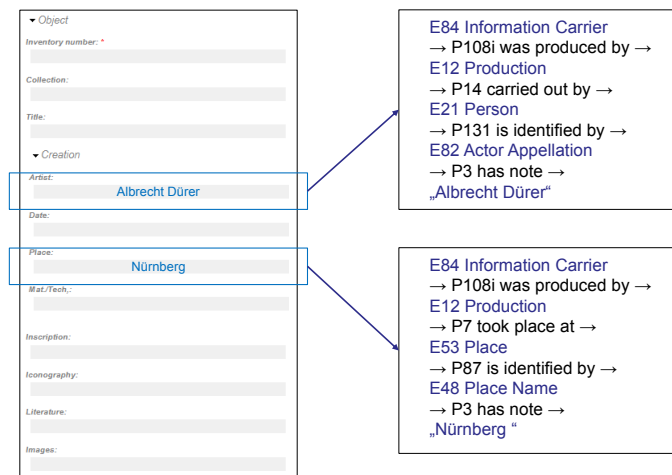
▷ subtrees in the ontology become sub-[bundles](#). (shared objects)



Even though we have introduced all the necessary concepts above, the best way of understanding this is to look at our running example again: the **path builder** induces a data entry form that allows us to enter a whole set of **ontology paths**, introducing and sharing intermediary objects along the way.

## The WissKI Path Builder (Example)

### ▷ Example 15.2.2 (A WissKI Group)



If we look at the data entry form on the left of Example 15.2.2, then we see that we only enter strings, not the objects we mean. So there is the problem of disambiguating which objects that are then linked to some object via **CIDOC CRM** relations we actually mean with the string.

## Sharing and Disambiguation in Path Builders

▷ **Observation 15.2.3** Sometimes we want to refer to existing entities in WissKI.

▷ **Example 15.2.4 (Referring to Nürnberg)** (We love tab completion)



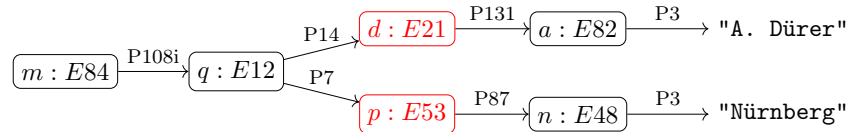
▷ **Example 15.2.5 (To What)** Albrecht Dürer created all his etchings in Nürnberg.

**Problem:** (In paths) we are creating lots of objects, which ones to offer?

► **Idea:** Mark the entities we might want to reuse on paths while specifying them.

► **Definition 15.2.6** A **disambiguation point** in a path marks an entity that can be re-used in data acquisition.

► **Example 15.2.7** **Disambiguation points** are highlighted in red on paths.



©: Michael Kohlhase

446



Now we can have a look at how **drupal** sees (and shows) path builders

## Specifying/Maintaining WissKI Path Builders

► **Recall:** A WissKI **path builder** maps **ontology groups** and **ontology paths** to **drupal bundles** and **fields**.

► **Example 15.2.8 (Specifying a WissKI Path Builder)**

TITLE	PATH	ENABLED	FIELD TYPE	CARDINALITY	OPERATIONS
✚ Werk	Group [ecrm:E22_Man-Made_Object]	<input checked="" type="checkbox"/>		Unlimited	<a href="#">Edit</a>
✚ Titel	ecrm:E22_Man-Made_Object -> ecrm:P102_has_title -> ecrm:E35_Title	<input checked="" type="checkbox"/>	Text (plain)	1	<a href="#">Edit</a>
✚ Verwalter	ecrm:E22_Man-Made_Object -> ecrm:P50_has_current_keeper -> ecrm:E40_Legal_Body -> ecrm:P1_is_identified_by -> ecrm:E82_Actor_Appellation	<input checked="" type="checkbox"/>	Text (plain)	1	<a href="#">Edit</a>
✚ Inventarnummer	ecrm:E22_Man-Made_Object -> ecrm:P1_is_identified_by -> ecrm:E42_Identifier	<input checked="" type="checkbox"/>	Text (plain)	1	<a href="#">Edit</a>
✚ Beziehung	ecrm:E22_Man-Made_Object -> ecrm:P46_forms_part_of -> ecrm:E22_Man-Made_Object -> ecrm:P102_has_title -> ecrm:E35_Title	<input checked="" type="checkbox"/>	Text (plain)	Unlimited	<a href="#">Edit</a>
✚ Herstellung	Group [ecrm:E22_Man-Made_Object -> ecrm:P1081_was_produced_by -> ecrm:E12_Production]	<input checked="" type="checkbox"/>		Unlimited	<a href="#">Edit</a>
✚ Hersteller	ecrm:E22_Man-Made_Object -> ecrm:P1081_was_produced_by -> ecrm:E12_Production -> ecrm:P14_carried_out_by -> ecrm:E21_Person -> ecrm:P131_is_identified_by -> ecrm:E82_Actor_Appellation	<input checked="" type="checkbox"/>	Text (plain)	Unlimited	<a href="#">Edit</a>
✚ Datum	ecrm:E22_Man-Made_Object -> ecrm:P1081_was_produced_by -> ecrm:E12_Production -> ecrm:P4_has_time-span -> ecrm:E52_Time-Span	<input checked="" type="checkbox"/>	Text (plain)	1	<a href="#">Edit</a>
✚ Ort	ecrm:E22_Man-Made_Object -> ecrm:P1081_was_produced_by -> ecrm:E12_Production -> ecrm:P7_took_place_at -> ecrm:E53_Place -> ecrm:P1_is_identified_by -> ecrm:E44_Place_Appellation	<input checked="" type="checkbox"/>	Text (plain)	Unlimited	<a href="#">Edit</a>
✚ Material	ecrm:E22_Man-Made_Object -> ecrm:P1081_was_produced_by -> ecrm:E12_Production -> ecrm:P32_used_general_technique -> ecrm:E57_Material -> ecrm:P1_is_identified_by -> ecrm:E75_Conceptual_Object_Appellation	<input checked="" type="checkbox"/>	Text (plain)	Unlimited	<a href="#">Edit</a>
✚ Technik	ecrm:E22_Man-Made_Object -> ecrm:P1081_was_produced_by -> ecrm:E12_Production -> ecrm:P33_used_specific_technique -> ecrm:E29_Design_or_Procedure -> ecrm:P1_is_identified_by -> ecrm:E75_Conceptual_Object_Appellation	<input checked="" type="checkbox"/>	Text (plain)	Unlimited	<a href="#">Edit</a>
✚ Kommentar	ecrm:E22_Man-Made_Object -> ecrm:P1291s_subject_of -> ecrm:E31_Document	<input checked="" type="checkbox"/>	Text (formatted, long)	1	<a href="#">Edit</a>
✚ Abbildung	ecrm:E22_Man-Made_Object -> ecrm:P1381has_representation -> ecrm:E36_Visual_Item -> ecrm:P1_is_identified_by -> ecrm:E51_Contact_Point	<input checked="" type="checkbox"/>	Image	Unlimited	<a href="#">Edit</a>



©: Michael Kohlhase

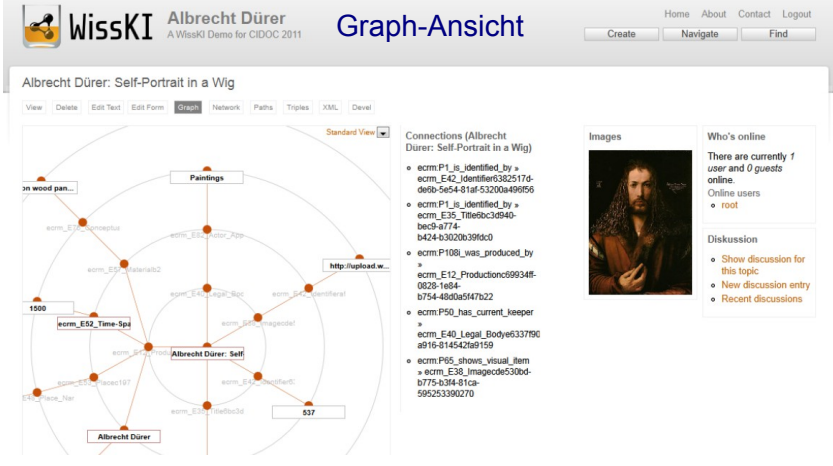
447



Of course all **paths** of an **ontology group** can be visualized as a graph. WissKI supports this as well.

## WissKI Path Builders as Graphs

► **Example 15.2.9 (A WissKI Path Construtor as a Graph)**



Albrecht Dürer: Self-Portrait in a Wig

View Delete Edit Text Edit Form Graph Network Paths Triples XML Devel

Standard View

Connections (Albrecht Dürer: Self-Portrait in a Wig)

- ecm:P1\_is\_identified\_by » ecm:E42\_Identifier(3825176-deb-5a54-01af-53200a49656)
- ecm:P1\_is\_identified\_by » ecm:E35\_Title(6c3d940-bec9-a774-b424-83020c306d0)
- ecm:P1081\_was\_produced\_by » ecm:E12\_Production(69934f-082b-1e84-b754-48da0a547622)
- ecm:P50\_has\_current\_keeper » ecm:E40\_Legal\_Body(633796-a916-814542a6159)
- ecm:P65\_shows\_visual\_item » ecm:E38\_Image(6530bd-b775-b384-81ca-595253390270)

Images

Who's online

There are currently 1 user and 0 guests online.

Online users

- root

Diskussion

- Show discussion for this topic
- New discussion entry
- Recent discussions

► Very nice and helpful, but does not work currently!

SOME RIGHTS RESERVED

©: Michael Kohlhase

448

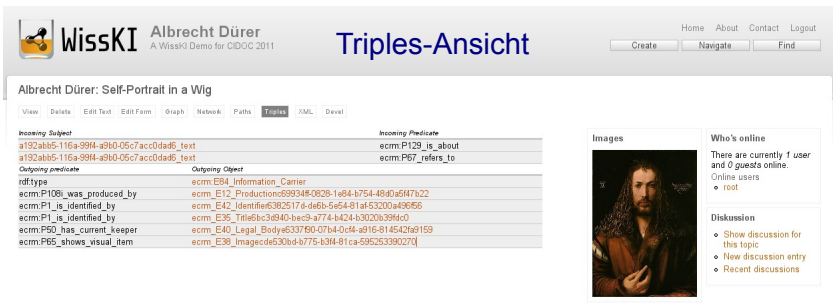
FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

And finally, a **path builder** can be seen as a set of triples – indeed this is the default export format for path builders.

Of course all **paths** of an **ontology group** can be visualized as a graph. WissKI supports this as well.

## WissKI Path Builders as Triples

- Of course we can view **path builders** as sets of triples.
- Example 15.2.10 (A WissKI Path Construtor as Triples)



Albrecht Dürer: Self-Portrait in a Wig

View Delete Edit Text Edit Form Graph Network Paths Triples XML Devel

Incoming Subject	Incoming Predicate	Outgoing Predicate	Outgoing Object
a192ab85-116a-99f4-a9d0-05c7acc0dad6_text	ecm:P129_is_about		
a192ab85-116a-99f4-a9d0-05c7acc0dad6_text	ecm:P67_refers_to		
rd:type	ecm:E84_Information_Carrier		
ecm:P1081_was_produced_by	ecm:E12_Production(69934f-082b-1e84-b754-48da0a547622)		
ecm:P1_is_identified_by	ecm:E42_Identifier(3825176-deb-5a54-01af-53200a49656)		
ecm:P1_is_identified_by	ecm:E35_Title(6c3d940-bec9-a774-b424-83020c306d0)		
ecm:P50_has_current_keeper	ecm:E40_Legal_Body(633796-a916-814542a6159)		
ecm:P65_shows_visual_item	ecm:E38_Image(6530bd-b775-b384-81ca-595253390270)		

Images

Who's online

There are currently 1 user and 0 guests online.

Online users

- root

Diskussion

- Show discussion for this topic
- New discussion entry
- Recent discussions

► Such an export also allows standardized communication.

SOME RIGHTS RESERVED

©: Michael Kohlhase

449

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

But of course, **path builders** can not only be used as data acquisition devices. They also define **drupal blocks** which can be used for data visualization (akin to fact boxes in Wikipedia).

## Data Presentation using Path Builders in WissKI

- **Path builders** can be used as **drupal blocks** for data presentation.

- ▷ For every object  $o$ , aggregate the values of the paths starting in  $o$ .

### ▷ Example 15.2.11 (Compressed View)

**WissKI** Albrecht Dürer  
A WissKI Demo for CIDOC 2011

**Komprimierte Ansicht**

Home About Contact Logout  
Create Navigate Find

Albrecht Dürer: Self-Portrait in a Wig

View Delete Edit Text Edit Form Graph Network Paths Triples XML Devel

Self-Portrait (earlier known as Self-Portrait at Twenty-Eight Years Old Wearing a Coat with Fur Collar or Self-Portrait in a Wig) is a painting on wood panel by the German Renaissance artist Albrecht Dürer. Painted early in 1500, just before his 29th birthday, it is the last of his three painted self-portraits. It is considered the most personal, iconic and complex of his self-portraits, and the one that has become fixed in the popular imagination.

The self-portrait is most remarkable because of its arrogant suggestion of divinity in its resemblance to many earlier representations of Christ. Art historians note the similarities with the conventions of religious painting, including its symmetry, dark tones and the manner in which the artist directly confronts the viewer and raises his hands to the middle of his chest as if in the act of blessing. It is likely that Dürer portrayed himself in this way through a combination of arrogance and a desire by a young and ambitious artist to acknowledge that his talent as God given.

- Object
  - Inventory number: 537
  - Collection: Paintings
  - Title: Self-Portrait in a Wig
- Creation
  - Artist: Albrecht Dürer
  - Date: 1500
  - Place: Abte Pinakothek, Munich
- Images
  - [http://upload.wikimedia.org/wikipedia/commons/thumb/c/c2/D%C3%BCrer\\_self\\_portrait\\_28.jpg/300px-D%C3%BCrer\\_self\\_portrait\\_28.jpg](http://upload.wikimedia.org/wikipedia/commons/thumb/c/c2/D%C3%BCrer_self_portrait_28.jpg/300px-D%C3%BCrer_self_portrait_28.jpg)

Images

Who's online

There are currently 1 user and 0 guests online.

Online users

- root

Diskussion

- Show discussion for this topic
- New discussion entry
- Recent discussions

SOME RIGHTS RESERVED

©: Michael Kohlhase

450

FAU  
FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG

## 15.3 The WissKI Link Block

### The WissKI Link Block (Idea)

- ▷ **Observation 15.3.1** For an entity in a RDF graph, both the outgoing and the incoming relations are important for understanding.
- ▷ **Example 15.3.2** This view only shows the outgoing edges!

**WissKI** Albrecht Dürer  
A WissKI Demo for CIDOC 2011

**Komprimierte Ansicht**

Home About Contact Logout  
Create Navigate Find

Albrecht Dürer: Self-Portrait in a Wig

View Delete Edit Text Edit Form Graph Network Paths Triples XML Devel

Self-Portrait (earlier known as Self-Portrait at Twenty-Eight Years Old Wearing a Coat with Fur Collar or Self-Portrait in a Wig) is a painting on wood panel by the German Renaissance artist Albrecht Dürer. Painted early in 1500, just before his 29th birthday, it is the last of his three painted self-portraits. It is considered the most personal, iconic and complex of his self-portraits, and the one that has become fixed in the popular imagination.

The self-portrait is most remarkable because of its arrogant suggestion of divinity in its resemblance to many earlier representations of Christ. Art historians note the similarities with the conventions of religious painting, including its symmetry, dark tones and the manner in which the artist directly confronts the viewer and raises his hands to the middle of his chest as if in the act of blessing. It is likely that Dürer portrayed himself in this way through a combination of arrogance and a desire by a young and ambitious artist to acknowledge that his talent as God given.

- Object
  - Inventory number: 537
  - Collection: Paintings
  - Title: Self-Portrait in a Wig
- Creation
  - Artist: Albrecht Dürer
  - Date: 1500
  - Place: Abte Pinakothek, Munich
- Images
  - [http://upload.wikimedia.org/wikipedia/commons/thumb/c/c2/D%C3%BCrer\\_self\\_portrait\\_28.jpg/300px-D%C3%BCrer\\_self\\_portrait\\_28.jpg](http://upload.wikimedia.org/wikipedia/commons/thumb/c/c2/D%C3%BCrer_self_portrait_28.jpg/300px-D%C3%BCrer_self_portrait_28.jpg)

Images

Who's online

There are currently 1 user and 0 guests online.

Online users

- root

Diskussion

- Show discussion for this topic
- New discussion entry
- Recent discussions

SOME RIGHTS RESERVED

©: Michael Kohlhase

451

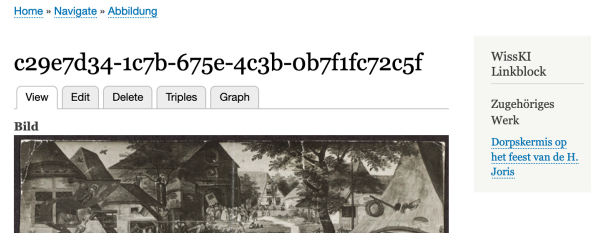
FAU  
FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG

Idea: Add a **block** with "incoming links" to the page, use the **path builder**.

### ▷ Link Blocks (Definition)

▷ **Definition 15.3.3** Let  $p$  be a [drupal](#) page for an [ontology group](#)  $g$ , then a [WissKI link block](#) is a special [drupal block](#) with associated [path builder](#), whose [ontology paths](#) all end in  $g$ .

▷ **Example 15.3.4 (A link block for Images)**



Note the difference between

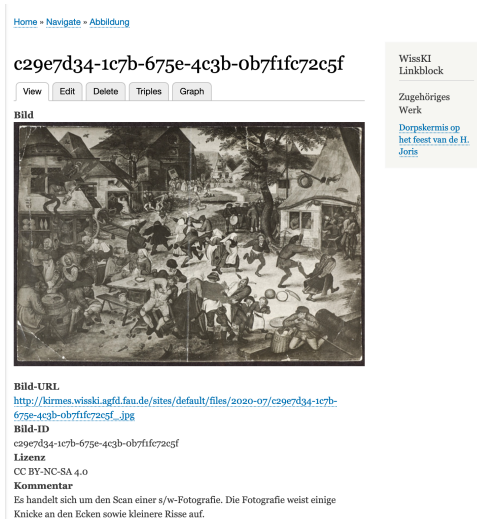
- ▷ a “work” – the original painting Pieter Brueghel created in 1628
- ▷ and an “image of the work” – a b/w photograph of the “work”.

This particular [link block](#) mediates between these two.



## A Link Block in the Wild (the full Picture)

▷ **Example 15.3.5 (A link block for Images)**



- ▷ [outgoing relations](#) below the image,
- ▷ [incoming ones](#) in the [link block](#)

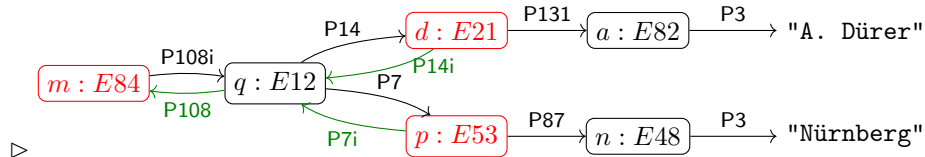


## Making Link Blocks via the Path Builder

▷ How to make a [link block](#) in page  $p$  for group  $g$ ? (Details at [WH])

1. create a [block](#) via the [config bar](#) and place it on  $p$ .
2. associate it with a [link block path builder](#)
3. model paths into  $g$  in the [path builder](#) (various source groups)

**Idea:** You essentially know [link builder](#) paths already: If you have already modeled a path  $g \xrightarrow{r_1} \dots \xrightarrow{r_n} s$  for a group  $s$ , then you have a path  $s \xrightarrow{r_n^{-1}} \dots \xrightarrow{r_1^{-1}} g$ , where  $r_i^{-1}$  are the inverse relations of  $r_i$  (exist in CIDOC CRM)



▷

**Note:** With this setup, you never have to fill out the link block paths!



©: Michael Kohlhase

454



## 15.4 Cultural Heritage Research: Querying WissKI Resources

So far, we have concentrated on the WissKI system, and how that can be used for data acquisition and documentation of [cultural artefacts](#). While we did this we lost view of the most important aspect: what are we doing data acquisition for? Arguably this is [cultural heritage](#) research – and we mean this in an inclusive manner – this could be academic research or researching for a school project or article in a newspaper.

This research and how the WissKI system can support is what we will go into now.

### ▷ Research in WissKI

▷ **So far** we have seen how to acquire complex knowledge about [cultural artefacts](#) using [CIDOC CRM Aboxes](#).

▷ **Question:** But how do we do research using WissKI?

▷ **Answer:** Finding patterns, inherent connections, ... in the data.

▷ **But how?:** That depends on the kind of research you want to do. Here are some WissKI research tools

1. we can use [drupal](#) search on the data.
2. We can formulate our own queries in SPARQL
3. We can pre-configure various queries in [drupal views](#).



©: Michael Kohlhase

455



The simplest form of “research” is just being able to search over the objects that have been created. This is one of the basic facilities WissKI offers out of the box. Already that can be quite useful.

## Drupal Search in WissKI

### ▷ Example 15.4.1

#### Search

Search WissKI Entities Content Users

Search by Entity Title

Entity Title

Finds titles from the cache table

▼ Advanced Search

in Bundles

☒ Künstler

☐ Abbildung

☐ Werk

in Paths

Künstler

Name (erfassungsmasken.name)  contains


Albrecht

Werke dieses Künstlers (pb\_wisslinkblock.werke\_dieses\_kuenstlers)  contains

Melencolia

Match

☒ All: ☐ Any:

 Search WissKI Entities

## SPARQL Endpoint in WissKI

### ▷ Example 15.4.2 Find kirmes paintings and their painters and count them

My account Log out

kirmes.wisski.agfd.fau.de

Home Find Navigate Create Query Endpoint


Home

#### Query Endpoint

Query

```
SELECT (COUNT(?kuenstlername) AS ?anzahl) ?kuenstlername ?werkttitel WHERE { GRAPH ?graph {
  ?kuenstler a <https://kirmes.wisski.agfd.fau.de/ontology/kirmes/ki21a_artist> . ?kuenstler
  <http://erlangen-crm.org/170309/P131_is_identified_by> ?name . ?name a <http://erlangen-crm.org/170309/ER2_Actor_Appellation> . ?name <http://erlangen-crm.org/170309/P3_has_note>
  ?kuenstlername . ?werk a <http://erlangen-crm.org/170309/E22_Man-Made_Object> . ?werk
  <http://erlangen-crm.org/170309/P108I_produced_by> ?herstellung . ?herstellung a
  <http://erlangen-crm.org/170309/E12_Production> . ?herstellung <http://erlangen-crm.org/170309/P14_carried_out_by> ?kuenstler . ?werk <http://erlangen-crm.org/170309/P102_has_title> ?ttitel .
  ?ttitel a <http://erlangen-crm.org/170309/E35_Title> . ?ttitel <http://erlangen-crm.org/170309/P3_has_note> ?werkttitel } GROUP BY ?kuenstlername ?werkttitel
ORDER BY DESC(?anzahl)}
```

Execute Query



Home Find Navigate Create Query Endpoint

### Query Endpoint

?anzahl	?kuenstlername	?werktitel
"2"^^xsd:integer	"Pieter Brueghel (II)"	"Dorpskermis op het feest van de H. Joris "
"1"^^xsd:integer	"Pieter Brueghel (II)"	"Dorpskermis op het feest van de H. Joris"

**Query**

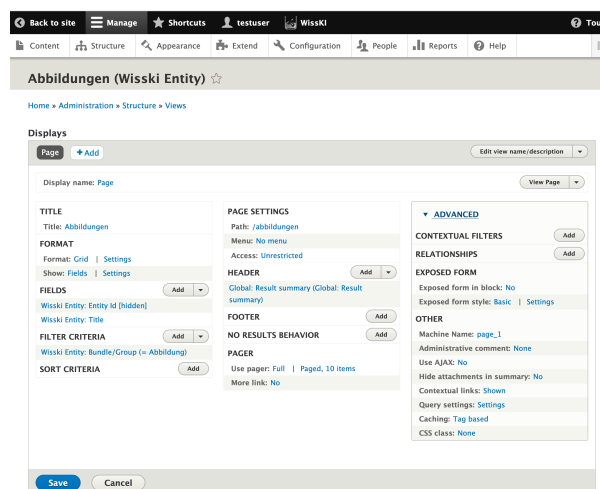
```
SELECT (COUNT (?kuenstlername) AS ?anzahl) ?kuenstlername ?werktitel WHERE { GRAPH ?graph {
  ?kuenstler a <https://kirmes.wisski.agfd.fau.de/ontology/kirmes/kir21a_artist> . ?kuenstler
  <http://erlangen-crm.org/170309/P131_is_identified_by> ?name . ?name a <http://erlangen-crm.org
  /170309/E82_Actor_Appellation> . ?name <http://erlangen-crm.org/170309/P3_has_note>
  ?kuenstlername . ?werk a <http://erlangen-crm.org/170309/E22_Man-Made_Object> . ?werk
```

Execute Query

©: Michael Kohlhase 457

## Data Presentation via Views in WissKI

▷ **Example 15.4.3 (Configuring a View)** This makes a [Drupal block](#).



Back to site Manage Shortcuts testuser WissKI Tour

Content Structure Appearance Extend Configuration People Reports Help

Abbildungen (Wisski Entity) ☆

Home » Administration » Structure » Views

**Displays**

Page + Add Edit view name/description View Page

Display name: Page

**TITLE**  
Title: Abbildungen

**FORMAT**  
Format: Grid Settings

**FIELDS**  
Show: Fields Settings Add  
Wisski Entity: Entity id (hidden)  
Wisski Entity: Title

**FILTER CRITERIA**  
Wisski Entity: Bundle/Group (= Abbildung) Add

**SORT CRITERIA**  
Add

**PAGE SETTINGS**  
Path: /abbildungen  
Menu: No menu  
Access: Unrestricted

**HEADER**  
Global: Result summary (Global: Result summary) Add

**FOOTER**  
Add

**NO RESULTS BEHAVIOR**  
Add

**PAGER**  
Use pager: Full | Paged, 10 items  
More link: No

**ADVANCED**

**CONTEXTUAL FILTERS**  
Add

**RELATIONSHIPS**  
Add

**EXPOSED FORM**  
Exposed form in block: No  
Exposed form style: Basic Settings

**OTHER**  
Machine Name: page\_1  
Administrative comment: None  
Use AJAX: no  
Hide attachments in summary: No  
Contextual links: Shown  
Query settings: Settings  
Caching: Tag based  
CSS class: None

Save Cancel

Drupal generates a SPARQL query, aggregates results into a [block](#).

## This Research is WissKI-instance-local

▷ **Observation 15.4.4** All these research queries only work in the current WissKI instance.

▷ **Observation 15.4.5** *There is probably much more about the entities you are interested in outside your particular WissKI instance.*

▷ **Problem:** How to make use of this?

▷ **Solution:** We need to do two things

1. Make use of other people's ABoxes
2. Provide your ABox to other people.

This practice is called [linked](#)opendata.

(up next)



©: Michael Kohlhase

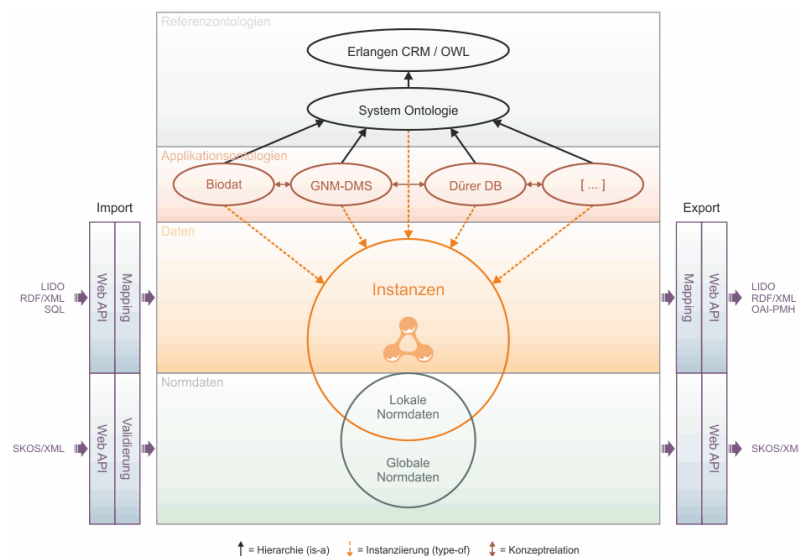
459



## 15.5 Application Ontologies in WissKI

### WissKI Information Architecture (Ontologies)

▷ Ontologies, instances, and export formats



©: Michael Kohlhase

460

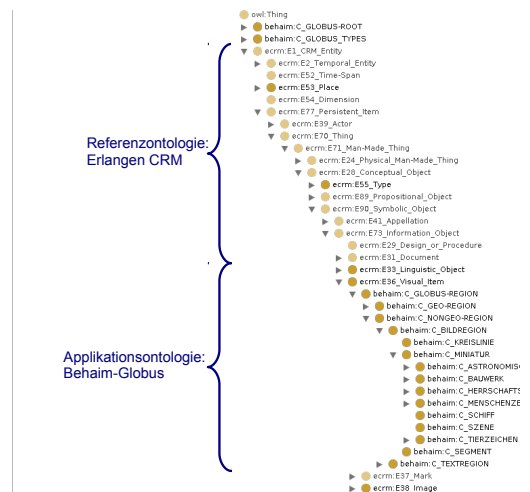


### Application Ontologies extend CIDOC CRM

▷ **Observation 15.5.1** *Sometimes we need more than [CIDOC CRM](#).*

▷ **Definition 15.5.2** A WissKI **application ontology** is one that extends [CIDOC CRM](#), without changing it.

### ▷ Example 15.5.3 (Behaim Application Ontology)



©: Michael Kohlhase

461



## Making an Application Ontology

- ▷ The “current ontology” of a WissKI instance can be configured via the [config bar](#) via the “WissKI ontology” module.
- ▷ The [application ontology](#) should import [CIDOC CRM](#).
- ▷ [Idea](#): Use Protégé for that.



©: Michael Kohlhase

462



## 15.6 The Linked Open Data Cloud

### Linked Open Data

- ▷ **Definition 15.6.1** **Linked data** is structured data in which classified objects are interlinked via [relations](#) with other objects so that the data becomes more useful through [semantic](#) queries and access methods.
- ▷ **Definition 15.6.2** **Linked open data (LOD)** is [linked data](#) which is released under an [open license](#), which does not impede its reuse by the community.
- ▷ **Definition 15.6.3** Given the Semantic Web technology stack, we can create interoperable ontologies and interlinked data sets, we call their totality the **linked open data cloud**.
- ▷ [Recall the LOD Incentives](#):

- ▷ incentivize other authors to **extend/improve the LOD**
  - ↪ more/better data can be generated at a lower cost.
- ▷ generate **attention** to the LOD and **recognition** for authors
  - ↪ this gives alternative revenue models for authors.



©: Michael Kohlhase

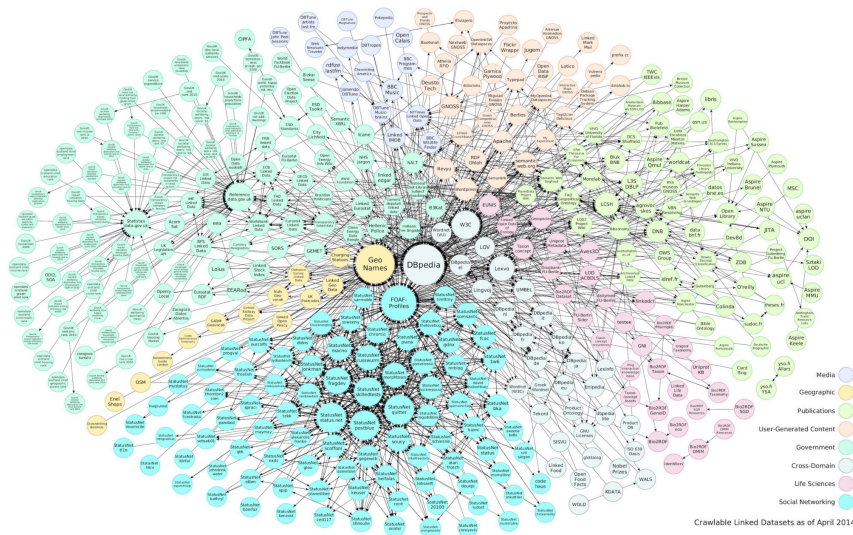
463



By Definition 15.6.3 the **linked open data cloud** is the totality of **linked open data** that has been published. **[lod-cloud:on]** tracks (the larger parts of) it. This gives us a sense of the extend of this giant network of knowledge expressed as triples.

## The Linked Open Data Cloud

- ▷ The **linked open data cloud** in 2014 (today much bigger, but unreadable)



©: Michael Kohlhase

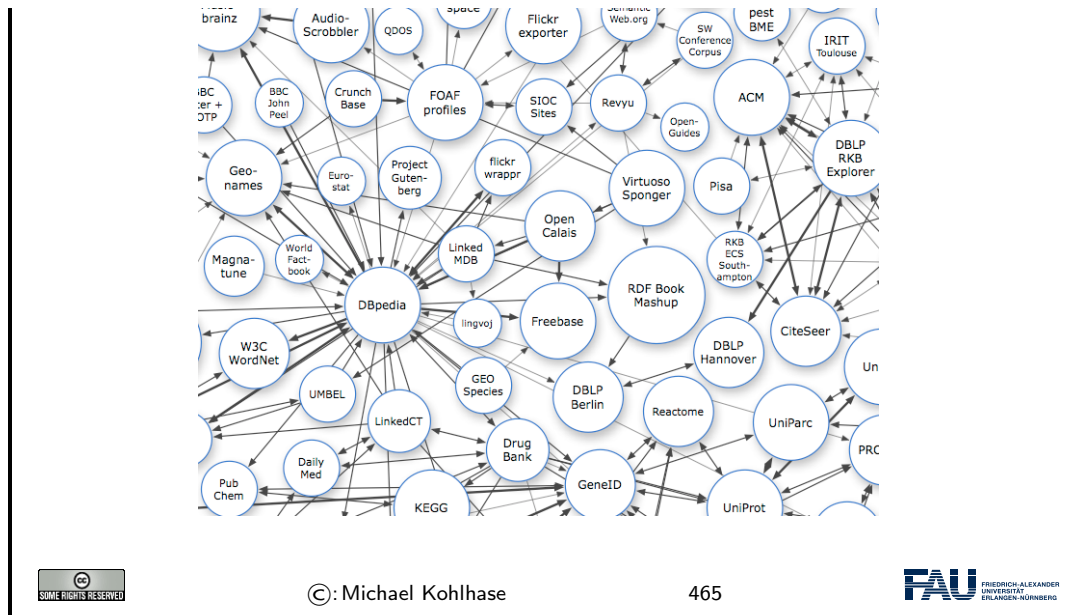
464



We now “zoom in” on this picture to get a better sense”. Each of the circles in the picture is a data set of at least 1000 triples. The DBPedia in the center of this fragment has 3 billion triples alone (in 2014).

## The Linked Open Data Cloud

- ▷ zooming in (data sets and their – interlinked – ontologies)



The ideas of the [linked open data cloud](#) directly apply knowledge about [cultural artefacts](#) as we formalize them in the WissKI system: we can directly reference objects from the cloud in WissKI.

### Using the LOD-Cloud in WissKI

- ▷ **Idea:** Do not re-model entities that already exist (in the LOD Cloud)
- ▷ **Problem:** Most of the LOD Cloud is about things we do not want.
- ▷ But there are some sources that are useful
  - ▷ the **GND** (**G**emeinsame **N**ormdatei [GND]), an authority file for personal/corporate names and keywords from literary catalogs,
  - ▷ **geonames** [GN], a geographical database with more than 25M names and locations
  - ▷ Wikipedia
- ▷ **Observation 15.6.4** All of them provide URIs for real-world entities, which is just what we need for objects in RDF [triples](#).
- ▷ **Definition 15.6.5** WissKI provides special [modules](#) called [adapters](#) for [GND](#) and [geonames](#).

Using [linked open data](#) in WissKI actually makes for higher-quality digitizations, as they are more interoperable. Unfortunately, WissKI only supports the two adapters we mention above. There are many many more that would be useful.

Let us now see how to concretely use an adapter, here for the geonames service.

## Using Geonames in WissKI (Example)

1. **Example 15.6.6** We want to use the “Meilwald” (Erlangen) in WissKI.
2. make a sub-ontology groups “norm data” in the WissKI [path builder](#)
3. The induced sub-bundle looks like this:

Normdatei:

Normdaten ID:

Normdatum URI:

This must be an external URL such as <http://example.com>.

☐ ☐

4. We enter <https://geodata.org> for “Normdatei” and go there to find out the URI for “Meilwald” which goes into “Normdatum URI”.



The GeoNames geographical database covers all countries and contains over eleven million placenames that are available for download free of charge.

Meilwald  all countries  [\[advanced search\]](#)

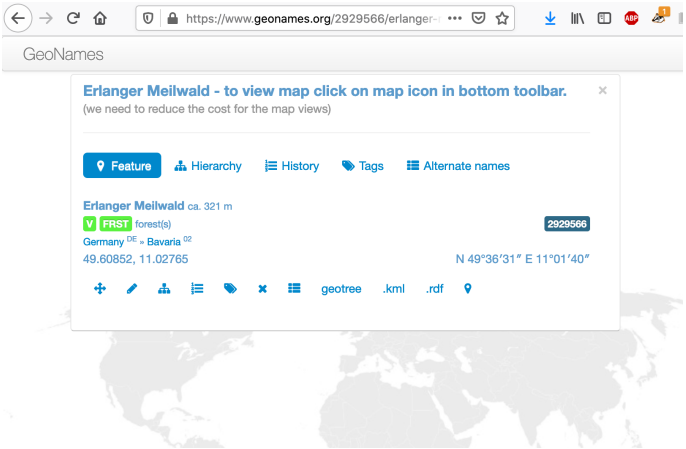
enter a location name, ex: "Paris", "Mount Everest", "New York"

5. there may be multiple results (here only one)

Name	Country	Feature class	Latitude	Longitude
<a href="#">Erlanger Meilwald</a>	<a href="#">Germany</a> , <a href="#">Bavaria</a>	forest(s)	N 49° 36' 30"	E 11° 1' 39"

1 records found for "Meilwald"

6. Select/click the intended one, check the details



7. Enter the URL from the URL bar into “Normdatum URI”.

Normdatei:

Normdaten ID:

Normdatum URI:

This must be an external URL such as <http://example.com>.

©: Michael Kohlhase 467

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

If we – as we did here – tell the story of using authority files in WissKI from a [linked open data](#) perspective, a curious asymmetry becomes apparent: WissKI is using [LOD](#) resources, but is – by and large – not contributing [LOD](#) resources back to the “public domain” of [linked open cultural heritage](#) data.

## Towards a WissKI Commons in the LOD Cloud

- ▷ **Recap:** We can directly refer to (URLs of) external objects in WissKI.
- ▷ **Observation 15.6.7** *The most interesting source for references to [cultural artefacts](#) are other WissKI instances.*
- ▷ **Problem:** A WissKI is an island, unless it exports its data! (few do)
- ▷ **Idea:** We need a [LOD](#) cloud of [cultural heritage research data](#) under to foster object-centric research in the humanities.
- ▷ **Definition 15.6.8** We call the part of this resource that can be created by aggregating WissKI exports the **WissKI commons**.
- ▷ **Observation 15.6.9** *WissKI exports meet the [FAIR](#) principles quite nicely already.*
- ▷ We will be working on a FAU [WissKI commons](#) in the next years. (help wanted)



This asymmetry is a very serious problem, since [cultural heritage](#) research is not profiting as much from digitizations as it could. Keeping data in WissKI silos – this is what we do when we are not exporting WissKI data and referencing objects from other WissKI instances – leads to fragmentation of the research community and to duplication of work.

# Chapter 16

## What did we learn in IWGS?

### Outline of IWGS 1:

- ▷ programming in python (main tool in IWGS)
  - ▷ systematics and culture of programming
  - ▷ program and control structures
  - ▷ basic data structures like numbers and strings, character encodings, unicode, and regular expressions
- ▷ digital documents and document processing
  - ▷ text files
  - ▷ markup systems, HTML, and CSS
  - ▷ XML: Documents are trees.
- ▷ Web technologies for interactive documents and web applications
  - ▷ Internet infrastructure: web browsers and servers
  - ▷ serverside computing: bottle routing and
  - ▷ clientside interaction: dynamic HTML, Javascript, HTML forms
- ▷ Web Application Project (fill in the blanks to obtain a working web app)



©: Michael Kohlhase

469



### Outline of IWGS-II:

- ▷ Project Management and Collaboration on Data, Documents, and Software
  - ▷ Revision Control Systems
  - ▷ Issue Trackers and Project Wikis
- ▷ Data bases
  - ▷ CRUD operations, DB querying, and python embedding

- ▷ XML and JSON for file-based data storage
- ▷ Image Processing
  - ▷ Basics
  - ▷ Image transformations, Image Understanding
- ▷ Legal Foundations of Information Systems
  - ▷ Copyright & Licensing
  - ▷ Data Protection (GDPR)
- ▷ Ontologies, Semantic Web, and WissKI
  - ▷ Ontologies (inference  $\leadsto$  get out more than you put in)
  - ▷ Semantic Web Technologies (standardize ontology formats and inference)
  - ▷ Using SWTech for cultural heritage  $\leadsto$  the WissKI System

# Bibliography

- [All18] Jay Allen. *New User Tutorial: Basic Shell Commands*. 2018. URL: <https://www.liquidweb.com/kb/new-user-tutorial-basic-shell-commands/> (visited on 10/22/2018).
- [BHK16] Jens Bove, Lutz Heusinger, and Angela Kailus. *Marburger Informations-, Dokumentations- und Administrations-System (MIDAS): Handbuch und CD*. 4th ed. K.G.Saur, 2016. DOI: 10.11588/artdok.00003770.
- [BLFM05] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. Internet Engineering Task Force (IETF), 2005. URL: <http://www.ietf.org/rfc/rfc3986.txt>.
- [CC] *CIDOC CRM - The CIDOC Conceptual Reference Model*. URL: <http://www.cidoc-crm.org/> (visited on 07/13/2020).
- [CQ69] Allan M. Collins and M. Ross Quillian. “Retrieval time from semantic memory”. In: *Journal of verbal learning and verbal behavior* 8.2 (1969), pp. 240–247. DOI: 10.1016/S0022-5371(69)80069-1.
- [CS14] Scott Chacon and Ben Straub. *Pro Git*. 2nd Edition. APress, 2014. ISBN: 978-1484200773. URL: <https://git-scm.com/book/en/v2>.
- [CSFP04] Ben Collins-Sussman, Brian W. Fitzpatrick, and Michael Pilato. *Version Control With Subversion*. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 2004. ISBN: 0596004486. URL: <http://svnbook.red-bean.com>.
- [CSSa] *All CSS Specifications*. URL: <https://www.w3.org/Style/CSS/specs.en.html> (visited on 01/12/2020).
- [CSSb] *CSS Specificity*. URL: [https://en.wikipedia.org/wiki/Cascading\\_Style\\_Sheets#Specificity](https://en.wikipedia.org/wiki/Cascading_Style_Sheets#Specificity) (visited on 12/03/2018).
- [CSSc] *CSS Tutorial*. URL: <https://www.w3schools.com/css/default.asp> (visited on 12/02/2018).
- [DCM12] DCMI Usage Board. *DCMI Metadata Terms*. DCMI Recommendation. Dublin Core Metadata Initiative, June 14, 2012. URL: <http://dublincore.org/documents/2012/06/14/dcmi-terms/>.
- [Dri10] Vincent Driessen. *A successful Git branching model*. online at <http://nvie.com/posts/a-successful-git-branching-model/>. 2010. URL: <http://nvie.com/posts/a-successful-git-branching-model/> (visited on 03/19/2015).
- [Dru] *Drupal.org – Community plumbing*. URL: <http://drupal.org> (visited on 02/14/2015).
- [Ecm] *ECMAScript Language Specification*. ECMA Standard. 5<sup>th</sup> Edition. Dec. 2009.
- [ECRMa] *erlangen-crm*. URL: <https://github.com/erlangen-crm> (visited on 07/13/2020).
- [ECRMb] *Erlangen CRM/OWL - An OWL DL 1.0 implementation of the CIDOC Conceptual Reference Model (CIDOC CRM)*. URL: <http://erlangen-crm.org/> (visited on 07/13/2020).

- [FAIR18] European Commission Expert Group on FAIR Data. *Turning FAIR into reality*. 2018. DOI: 10.2777/1524.
- [Fie+99] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. Internet Engineering Task Force (IETF), 1999. URL: <http://www.ietf.org/rfc/rfc2616.txt>.
- [FOAF14] *FOAF Vocabulary Specification 0.99*. Namespace Document. The FOAF Project, Jan. 14, 2014. URL: <http://xmlns.com/foaf/spec/>.
- [Gla17] Matt Glaman. *Drupal 8 Development Cookbook – Harness the power of Drupal 8 with this recipe-based practical guide*. 2nd ed. Packt Publishing, 2-17. ISBN: 9781788290401.
- [GN] *Geonames*. URL: <https://www.geonames.org/> (visited on 07/29/2020).
- [GND] *DNB – The Integrated Authority File (GND)*. URL: [https://www.dnb.de/EN/Professionell/Standardisierung/GND/gnd\\_node.html](https://www.dnb.de/EN/Professionell/Standardisierung/GND/gnd_node.html) (visited on 07/29/2020).
- [Her+13a] Ivan Herman et al. *RDF 1.1 Primer (Second Edition). Rich Structured Data Markup for Web Documents*. W3C Working Group Note. World Wide Web Consortium (W3C), 2013. URL: <http://www.w3.org/TR/rdfa-primer>.
- [Her+13b] Ivan Herman et al. *RDFa 1.1 Primer – Second Edition. Rich Structured Data Markup for Web Documents*. W3C Working Group Note. World Wide Web Consortium (W3C), Apr. 19, 2013. URL: <http://www.w3.org/TR/xhtml-rdfa-primer/>.
- [Hic+14] Ian Hickson et al. *HTML5. A Vocabulary and Associated APIs for HTML and XHTML*. W3C Recommendation. World Wide Web Consortium (W3C), Oct. 28, 2014. URL: <http://www.w3.org/TR/html5/>.
- [HiDa] *HiDa*. URL: <https://www.startext.de/produkte/hida> (visited on 07/12/2020).
- [Hit+12] Pascal Hitzler et al. *OWL 2 Web Ontology Language Primer (Second Edition)*. W3C Recommendation. World Wide Web Consortium (W3C), 2012. URL: <http://www.w3.org/TR/owl-primer>.
- [HL11] Martin Hilbert and Priscila López. “The World’s Technological Capacity to Store, Communicate, and Compute Information”. In: *Science* 331 (2011). DOI: 10.1126/science.1200970. URL: <http://www.sciencemag.org/content/331/6018/692.full.pdf>.
- [HWC] *The Hello World Collection*. URL: <http://helloworldcollection.de/> (visited on 11/23/2018).
- [Kar] Folgert Karsdorp. *Python Programming for the Humanities*. URL: <http://www.karsdorp.io/python-course/> (visited on 10/14/2018).
- [KC04] Graham Klyne and Jeremy J. Carroll. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation. World Wide Web Consortium (W3C), Feb. 10, 2004. URL: <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>.
- [Koh06] Michael Kohlhase. *OMDoc – An open markup format for mathematical documents [Version 1.2]*. LNAI 4180. Springer Verlag, Aug. 2006. URL: <http://omdoc.org/pubs/omdoc1.2.pdf>.
- [Koh08] Michael Kohlhase. “Using L<sup>A</sup>T<sub>E</sub>X as a Semantic Markup Format”. In: *Mathematics in Computer Science 2.2* (2008), pp. 279–304. URL: <https://kwarc.info/kohlhase/papers/mcs08-stex.pdf>.
- [Koh20] Michael Kohlhase. *sT<sub>E</sub>X: Semantic Markup in T<sub>E</sub>X/L<sup>A</sup>T<sub>E</sub>X*. Tech. rep. Comprehensive T<sub>E</sub>X Archive Network (CTAN), 2020. URL: <http://www.ctan.org/get/macros/latex/contrib/stex/sty/stex.pdf>.
- [LP] *Learn Python – Free Interactive Python Tutorial*. URL: <https://www.learnpython.org/> (visited on 10/24/2018).
- [LXMLa] *lxml – XML and HTML with Python*. URL: <https://lxml.de> (visited on 12/09/2019).

- [LXMLb] *lxml API*. URL: <https://lxml.de/api/> (visited on 12/09/2019).
- [LXMLc] *The lxml.etree Tutorial*. URL: <https://lxml.de/tutorial.html> (visited on 12/09/2019).
- [Nor+18a] Emily Nordmann et al. *Lecture capture: Practical recommendations for students and lecturers*. 2018. URL: <https://osf.io/huydx/download>.
- [Nor+18b] Emily Nordmann et al. *Vorlesungsaufzeichnungen nutzen: Eine Anleitung für Studierende*. 2018. URL: <https://osf.io/e6r7a/download>.
- [OWL09] OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation. World Wide Web Consortium (W3C), Oct. 27, 2009. URL: <http://www.w3.org/TR/2009/REC-owl2-overview-20091027/>.
- [P3D] *Python 3 Documentation*. URL: <https://docs.python.org/3/> (visited on 09/02/2014).
- [PMDA] *Python – MySQL Database Access*. URL: [https://www.tutorialspoint.com/python/python\\_database\\_access.htm](https://www.tutorialspoint.com/python/python_database_access.htm) (visited on 11/18/2018).
- [Pro] *Protégé*. Project Home page at <http://protege.stanford.edu>. URL: <http://protege.stanford.edu>.
- [PRR97] G. Probst, St. Raub, and Kai Romhardt. *Wissen managen*. 4 (2003). Gabler Verlag, 1997.
- [PS08] Eric Prud'hommeaux and Andy Seaborne. *SPARQL Query Language for RDF*. W3C Recommendation. World Wide Web Consortium (W3C), Jan. 15, 2008. URL: <http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>.
- [PyRegex] Rodolfo Carvalho. *PyRegex - Your Python Regular Expression's Best Buddy*. URL: <http://www.pyregex.com/> (visited on 12/03/2018).
- [Pyt] *re – Regular expression operations*. online manual at <https://docs.python.org/2/library/re.html>. URL: <https://docs.python.org/2/library/re.html>.
- [RHJ98] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. *HTML 4.0 Specification*. W3C Recommendation REC-html40. World Wide Web Consortium (W3C), Apr. 1998. URL: <http://www.w3.org/TR/PR-xml.html>.
- [Smi76] Adam Smith. *An Inquiry into the Nature and Causes of the Wealth of Nations*. W. Strahan and T. Cadell, 1776.
- [SR14] Guus Schreiber and Yves Raimond. *RDF 1.1 Primer*. W3C Working Group Note. World Wide Web Consortium (W3C), 2014. URL: <http://www.w3.org/TR/rdf-primer>.
- [SSU04] Susan Schreibman, Ray Siemens, and John Unsworth, eds. *A Companion to Digital Humanities*. Wiley-Blackwell, 2004. ISBN: 978-1-405-10321-3. URL: <http://www.digitalhumanities.org/companion>.
- [Sth] *A Beginner's Python Tutorial*. <http://www.sthurlow.com/python/>. seen 2014-09-02. URL: <http://www.sthurlow.com/python/>.
- [STPL] *Simple Template Engine*. URL: <https://bottlepy.org/docs/dev/stpl.html> (visited on 12/08/2018).
- [SUMO] *Suggested Upper Merged Ontology*. URL: <http://www.adampease.org/OP/> (visited on 01/25/2019).
- [Swe13] Al Sweigart. *Invent with Python: Learn to program by making computer games*. 2nd ed. online at <http://inventwithpython.com>. 2013. ISBN: 978-0-9821060-1-3. URL: <http://inventwithpython.com>.
- [Tom17] Todd Tomlinson. *Enterprise Drupal 8 Development – For Advanced Projects and Large Development Teams*. Apress, 2017. ISBN: 9781484202548.
- [Tur95] Sherry Turkle. *Life on the Screen: Identity in the Age of the Internet*. Simon & Schuster, 1995.

- [WH] *WissKI Handbuch*. URL: [http://wiss-ki.eu/documentation/wisski\\_handbuch](http://wiss-ki.eu/documentation/wisski_handbuch) (visited on 07/23/2020).
- [Wil+16] Mark D. Wilkinson et al. “The FAIR Guiding Principles for scientific data management and stewardship”. In: *Scientific Data* 3 (2016). DOI: 10.1038/sdata.2016.18.
- [Xam] *apache friends - Xampp*. <http://www.apachefriends.org/en/xampp.html>. URL: <http://www.apachefriends.org/en/xampp.html>.

# Index

- Cryptography, 159
- Accessible, 271
- Authentication, 159
- Drupal, 306
- F-strings, 53
- Findable, 271
- Internal, 158
- Interoperable, 271
- Markdown, 162
- Metadata, 271
- Mutable, 84
- Ownership, 256
- Private, 158
- Public, 158
- Raising, 181
- Reusable, 271
- sparql
  - endpoint, 297
- ABox, 286
- assertions, 286
- absolute
  - URI, 93
- academic
  - culture, 5
- accept, 164
- access
  - control, 198
  - management, 198
- Anti-Counterfeiting Trade Agreement, 258
- adapters, 321
- added, 147
- anonymized, 267
- anonymous
  - function, 54
- application
  - ontology, 318
- architectural
  - work, 258
- arity, 40
- American Standard Code for Information Inter-
  - change, 47
- assertions, 283
- ABox, 283
- assignee, 165
- assigns, 26, 164
- attachment, 161
- attribution, 265
- attribute, 82
  - node, 81
- attributes, 175
- audiovisual
  - work, 258
- aural
  - markup, 64
- authentication
  - factor, 159
- authority, 92
- authorization, 158
- availability
  - control, 266
- axioms, 286
- back
  - end, 99
- balanced
  - bracketing
    - structure, 81
- base, 45
- basic
  - multilingual
    - plane, 49
- begin
  - tag, 69
- Berne
  - convention, 257
- binary, 16, 46
  - file, 64
  - unit
    - prefix, 66
- bit, 66
- blocks, 306
- body, 28
- Booleans, 27
- border, 115
- bottle
  - WSGI, 108
- box, 115
- bracketing
  - balanced (structure), 81

- branch, 150
- branches, 28
- browsing, 92
- bugtracker, 161
- issue
  - tracker, 160
  - tracking
    - system, 160
- bundle, 308
- byte, 66
- Creative Commons
  - license, 264
- cells, 24
- centralized, 151
- certificate
  - authority, 201
- character
  - encoding, 50
- characters, 49
- checkout, 146
- child
  - table, 176
- choreographic
  - work, 258
- CIDOC
  - CRM, 288
- civil
  - law
    - tradition, 257
- clone, 152
- close, 165
- closed, 37
- closing
  - tag, 81
- code
  - block, 108
  - cell, 24
  - line, 108
  - point, 49
- column
  - name, 169
  - specification, 172
- attributes, 169
- field)s, 169
- columns, 169
- commercial
  - use, 265
- commit, 146
- common
  - law
    - tradition, 257
- complexess, 27
- component, 92
- compose, 33
- composition
  - principle, 33
- classes, 286
- concepts, 281, 286
- concrete
  - data, 271
- condition, 28
- conditional
  - execution, 28
  - statement, 28
- conflict, 147
  - marker, 147
- content, 115
- control
  - flow, 28
  - structure, 28
  - word, 63
- conversation, 161
- cookie, 110
- copyleft, 263
- copyright, 259
  - holder, 260
  - infringement, 260
- copyrightable
  - work, 258
- copyrighted, 259
- Cascading Style Sheets, 112
- cultural
  - artefact, 269
  - heritage, 269
- cursor, 186
- CWA, 301
- closed
  - world
    - assumption, 301
- cypher
  - text, 159
- data
  - controller, 266
  - subject, 266
  - transfer
    - control, 266
- database, 167
  - browser, 171
  - schema, 172
  - transaction, 179
- DBMS, 169
- database
  - management
    - system, 169
- DDL, 172
- data

- definition
  - language, 172
- decimal, 46
- declaration, 113
  - block, 113
- decode, 159
  - key, 159
- default
  - namespace
    - declaration, 87
  - value, 55
- DELETE, 96
- delete, 174
- deleted, 147
- dependencies, 165
- derivative
  - works, 265
- derive, 282
- derived, 286
- description, 161
  - logic, 286
- developers, 158
- development, 156
  - history, 146
- diff, 147
- digital
  - text, 63
- digits, 45
- direct
  - identifier, 267
- disambiguation
  - point, 311
- distributed, 151
- document
  - root, 82
  - type, 64
- DOM, 79
- document
  - object
    - model, 79
- dot
  - notation, 40
- double
  - star
    - operator, 57
- downstream, 152
- dramatic
  - work, 258
- drupal
  - core, 307
- DUPLICATE, 165
- dynamic
  - route, 105
- element
  - node, 81
- elements, 34
- empty
  - element
    - tag, 81
  - tag, 69
- encode
  - key, 159
- encoding, 159
- end
  - tag, 69
- entities, 175
- ERD, 175
- entity
  - relationship
    - diagram, 175
- Erlangen
  - CRM/OWL, 288
- escape
  - character, 52
  - sequence, 52
- end-user
  - license
    - agreement, 262
- event-handler
  - attribute, 126
- events, 126, 288
- exbi, 66
- exception, 181
- exploitation
  - rights, 260
- expressions, 108
- formatted
  - string
    - literal, 53
- f-strings, 53
- facts, 286
- FAIR, 270
- fair
  - use
    - doctrine, 261
- feature
  - branch, 150, 156
  - request, 161
- fetches, 152
- fields, 308
- FIXED, 165
- floats, 27
- FLOSS, 263
- Free/Libre/Open-Source
  - Software, 263
- open

- source, 263
- for
  - loop, 35
- foreign
  - key, 176
- fork, 150, 156
- forks, 152
- form
  - action, 74
  - data, 102
- formatted
  - text, 63
- fragment, 92
- front
  - end, 99
- function/argument
  - notation, 284
- function
  - object, 54
- functional
  - syntax, 297
- GDPR, 266
- geonames, 321
- GET, 96
- GFM, 163
- GitHub
  - flavored
    - markdown, 163
- gibi, 66
- GIT
  - flow, 156
- GND, 321
- Gemeinsame
  - Normdatei, 321
- General
  - Public
    - License, 263
- grayscale, 207
- namespaces, 157
- groups, 157
- guests, 158
- handled, 181
- head, 146
- head
  - revision, 146
- headless, 151
- bare, 151
- height, 115
- hexadecimal, 46
- higher-order
  - function, 55
- host, 96
- hostnames, 96
- HyperText Markup Language, 69
- HTTP, 95
  - basic
    - authentication, 198
    - request, 96
- Hypertext
  - Transfer
    - Protocol, 95
- hunks, 147
- hyperlink, 92
- hypertext, 92
- ICH, 269
- intangible
  - cultural
    - heritage, 269
- idempotent, 96
- immaterial
  - thing, 288
- immutable, 84
- implicit, 282
- inference, 282
- information
  - access
    - control, 266
    - privacy, 265
- inheritable, 117
- inheritance
  - factor, 159
- inherits, 117
- Initialize, 153
- input
  - control, 266
  - element, 74
- inserting, 173
- instructions, 172
- integers, 27
- integrity, 180
- intellectual
  - property, 255
- Internet, 91
- INVALID, 165
- inventory
  - book, 273
- internationalized
  - resource
    - identifier, 94
- IRIs, 94
- ISO-Latin, 48
- isolation, 180
- issue, 161
  - metadata, 161
  - number, 165

- bug
  - report, 161
- issues, 161
- iterates, 35
- iteration, 35
- join, 183
- table
  - join, 183
- keyword
  - argument, 55, 56
- kibi, 66
- knowledge
  - factor, 159
- labels, 165
- Let's Encrypt, 201
- library, 41
- license, 261
- licensee, 261
- licensor, 261
- line
  - feed
    - character, 64
  - number, 64
- text
  - line, 64
- lines, 64
- link
  - block, 314
- linked
  - data, 319
- list, 34
  - constructor, 34
- literary
  - work, 258
- local
  - name, 87
  - repository, 152
- localhost, 96
- locator, 147
- LOD, 319
- linked
  - open
    - data, 319
- loop, 28
- maintainers, 158
- margin, 115
- markdown
  - cell, 24
- markup, 63
  - code, 63
  - format, 64
- document
  - markup, 63
- master
  - branch, 150, 156
- trunk, 150
- material
  - thing, 288
- mebi, 66
- media
  - query, 119
- merge, 147
- two-way
  - merge, 147
- three-way
  - merge, 147
- methods, 96
- milestones, 165
- modules, 307
- musical
  - work, 258
- named
  - wildcard, 105
- namespace
  - prefix, 87
- namespace
  - declaration, 81, 87
- narrative
  - data, 271
- navigating, 92
- node
  - text, 81
- null
  - value, 169
- object, 294
- objects, 40, 286
- obligation
  - of
    - separation, 266
- octal, 46
- Open
  - Data
    - Commons, 264
- ODF, 65
- Open Office Format, 65
- ontology, 286
  - group, 292, 309
  - path, 292
  - web
    - language, 296
- OOXML, 65
- Office

- Open
  - XML, 65
- open, 264
  - content, 264
  - data, 264
  - license, 264
- opened, 37
- opening
  - tag, 81
- OWA, 301
- open
  - world
    - assumption, 301
- owners, 158, 256
- ownership
  - factor, 159
- padding, 115
- page
  - inspector, 119
- parameter
  - substitution, 187
- parent, 147
  - table, 176
- reference
  - table, 176
- participants, 165
- patches, 147
- path, 92
  - builder, 309
- pebi, 66
- PERL
  - CGI, 103
- permission
  - level, 158
- persistence
  - layer, 100
- personal
  - group, 157
  - rights, 259
- PGS
  - work, 258
- pictorial, graphic and sculptural
  - work, 258
- physical
  - access
    - control, 266
- PII, 266
- personally
  - identifiable
    - information, 266
- pixel, 206
- plain
  - text, 63, 159
- positional
  - number
    - system, 45
- POST, 96
- predicate, 294
- prefixed
  - name, 87
- primary
  - key, 176
- primitive, 33
- projects, 157
- property, 113, 256, 294
  - right, 256
  - value, 294
- properties, 286
- relationships, 286
- pseudonymized, 267
- public
  - domain, 259
  - key
    - cryptography, 159
- pull, 152
  - request, 156
- punch
  - card, 48
- pushed, 152
- PUT, 96
- python
  - console, 21
- quasi-identifiers, 267
- query, 92, 182
- radix, 45
- range, 35
- raster, 206
- raw
  - cell, 24
  - string
    - literal, 52
- RDBMS, 169
- RDF
  - graph, 294
- Resource Description Framework, 293
- read, 37
- regexp, 57
- regular
  - expression, 57
- relations, 175, 282, 286
- relative
  - URI, 93
- release
  - branch, 150, 156
- remote

- repository, 152
- renewal
  - provision, 261
- reporters, 158
- reports, 164
- repositories, 146
- repository
  - hosting
    - service, 157
  - management
    - system, 157
  - server, 157
- represented, 286
- research
  - data, 270
- resolution, 165
- resolved, 147
- resource, 294
- revision, 146
  - control
    - action, 146
    - system, 146
- route, 104
  - filter, 105
  - function, 104
- routes, 103
- routing, 103, 104
- serverside
  - routing, 104
- row, 169
- database
  - record, 169
- RTFM, 17, 42
- rules, 113
- RWD, 119
- responsive
  - web
    - design, 119
- safe, 96
- scheme, 92
- selectors, 113
- self-join, 183
- semantic, 302
  - network, 281
  - web, 276
- semantics, 33
- sequence, 35
- server-side
  - scripting
    - framework, 103
- share
  - alike, 265
- shell, 22
- sound
  - recording, 258
- source, 16, 147
- space-time, 288
- SQL, 172
  - injection
    - attack, 180
- structured
  - query
    - language, 172
- SQL-sanitizes, 187
- SQLite
  - shell, 170
- staging, 154
  - branch, 150
- star
  - operator, 56
- state, 165
- static
  - content, 306
- stpl
  - python, 108
- streams, 37
- string
  - literal, 52
- strings, 27, 51
- subject, 294
- symbolic
  - data, 271
- syntax, 33
- system
  - access
    - control, 266
- table
  - name, 169
- tables, 169
- tags, 69
- target, 147
- TBox, 286
- terminology, 286
- tebi, 66
- template
  - engine, 107
  - file, 107, 108
  - functions, 109
  - processing, 107
- template
  - processor, 107
- term
  - provision, 261
- terminal, 21
- terminology, 283
- Isa-Hierarchy, 283

- TBox, 283
- territory
  - provision, 261
- text
  - editor, 65
  - file, 64
  - node, 81, 84
- textual
  - content, 64
- themes, 307
- third party
  - cookies, 110
- title, 161
- transaction, 179
- trriages, 164
- triple, 294
- statement, 294
- triplestore, 299
- RDF
  - store, 299
- UCS, 49
- universal
  - character
    - set, 49
- UNA, 301
- unique
  - name
    - assumption, 301
- unary, 46
  - natural
    - numbers, 44
- unicode
  - Standard, 49
- unique, 176
- update, 146, 174
- upstream, 152
- URI, 92
  - decoding, 95
  - encoding, 94
- uniform
  - resource
    - identifier, 92
- URL, 94
- uniform
  - resource
    - locator, 94
- URN, 94
- uniform
  - resource
    - name, 94
- user
  - agent, 95
- user-supplied
  - content, 306
- value, 113, 169
- variable, 25
  - assignment, 26
  - name, 25
- view, 184
- database
  - view, 184
- views, 306
- visual
  - markup, 64
- vocabulary, 286
- web
  - application, 99
  - browser, 70
  - page, 92
  - resource, 92
  - server, 96
  - site, 92
- WFH, 260
- work made for hire, 260
- width, 115
- WissKI
  - commons, 323
- WONTFIX, 165
- word
  - processor, 65
- working
  - copy, 146
- WORKSFORME, 165
- write, 37
- WWW, 91
- World Wide Web, 91
- WWWeb, 91
- XML
  - document
    - tree, 81
  - literal, 85
  - path
    - language, 87
- XML
  - namespace, 87
- yobi, 66
- zebi, 66