

Informatische Werkzeuge in den Geistes- und Sozialwissenschaften 1

Prof. Dr. Michael Kohlhase

Professur für Wissensrepräsentation und -verarbeitung
Informatik, FAU Erlangen-Nürnberg

Michael.Kohlhase@FAU.de

July 26, 2019

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 “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) 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 – 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. 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 CSS; see Chapter 3.
2. Data bases; in particular Entity Relationship diagrams, CRUD operations, and DB querying; see Chapter 8.
3. Web technologies for interactive documents and applications; in particular Internet infrastructure, web browsers and servers, PHP, dynamic HTML, Javascript, and HTML; see `?sec.webxml?`.

The last topic will be integrated into a simple student project.

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

1. Copyright and Data Privacy as legal foundations of data/program oriented work
2. large-scale collaborative development tools: revision control system and issue trackers, in particular Git and GitLab
3. Image processing tools, e.g. tensorflow, pyCV
4. Semantic Web and WissKI
5. Information systems for academic peer review (EasyChair in form of a student project)

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 made available for the students of this course only. It is still very much a draft and will develop over the course of the current course and 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 \LaTeX package [Koh08; Koh18], 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 mostly based on 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 Heinrich Stamerjohanns, Florian Rabe, and Peter Baumann.

All course materials have been restructured and semantically annotated in the \LaTeX 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 2018/19 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 [?sec.iwgs-contents?](#).

[Recorded Syllabus Winter Semester 2018/19:](#)

#	date	until	slide	page
1	Oct 18.	admin, overview	11	6
2	Oct 25.	python intro	34	25
	Nov. 1.	All Hallows Day (public holiday)		
3	Nov. 8.	python fundamentals	50	33
4	Nov. 15.	review fundamentals, functions	56	37
5	Nov. 22.	number/character representation, unicode	68	46
6	Nov. 29.	regular expressions	77	51
7	Dec. 6.	plain/formatted text, HTML	87	57
8	Dec. 13.	HTML & CSS	103	67
9	Dec. 20.	Review: HTML & CSS	103	67
10	Jan. 10.	New Year recap; CSS	111	73
11	Jan. 17.	Architecture of the WWW	130	85
12	Jan. 24.	web applications, bottle	145	92
13	Jan. 31.	client-side computation, JavaScript, JQuery	154	98

[Recorded Syllabus Summer Semester 2018:](#)

#	date	until	slide	page
1.	April 25.	admin, overview, Revision Control	175	118
2.	May 2.	distributed revision control, workflows	187	126
3.	May 9.	GitLab, issues		
4.	May 16.	Databases, DDL, sqlite3	216	144
5.	May 23.	SQL Queries, Views	226	150
	May 30	Public Holiday: Christi Himmelfahrt		
6.	June 6.	Image Processing	268	181
7.	June 13.	Image Maps via SVG/CSS	296	200
	June 20.	Public Holiday: Fronleichnam		
8.	June 27.	Legal Foundations of IT	310	212
9.	July 4.	Information Privacy, Semantic Web	327	220
10.	July 11.	RDF, Linkd Open Data	356	236
11.	July 18.	WissKI, What have we learned	373	247
	July 25.	Exam		

Contents

Preface	i
Course Concept	i
Course Contents	i
This Document	ii
Acknowledgments	ii
Recorded Syllabus	iii
1 Preliminaries	1
1.1 Administrativa	1
1.2 Goals, Culture, & Outline of IWGS	3
1.3 About My Lecturing	5
I IWGS-1: Programming, Documents, Web Applications	9
2 Introduction to Programming	11
2.1 Programming in IWGS	11
2.1.1 Introduction to Programming	11
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	24
2.2.3 Python Control Structures	27
2.2.4 Sequences and Iteration	30
2.2.5 Input and Output	32
2.2.6 Functions and Libraries in Python	34
2.2.7 A Final word on Programming in IWGS	38
3 Documents as Digital Objects	39
3.1 Preliminaries: Data Structures, Documents, and Sizes	39
3.1.1 Representing and Manipulating Numbers	39
3.1.2 Characters and their Encodings	43
3.1.3 Computing with Strings	47
3.1.4 Representing & Manipulating Documents on a Computer	52
3.1.5 Measuring Sizes of Documents/Units of Information	53
3.2 Multimedia Documents on the World Wide Web	56
3.2.1 Hypertext Markup Language	56
3.2.2 Cascading Stylesheets	61
3.3 An Overview over XML Technologies	71

4	Web Applications	77
4.1	Basic Concepts of the World Wide Web	77
4.1.1	Preliminaries	77
4.1.2	Addressing on the World Wide Web	79
4.1.3	Running the World Wide Web	81
4.1.4	HTML Forms and the Web	84
4.2	Generating HTML on the Server	85
4.2.1	Templating in Python via STPL	86
4.2.2	Routing, and Argument Passing in Bottle	90
4.3	Dynamic HTML: Client-side Manipulation of HTML Documents	93
4.3.1	JavaScript in HTML	94
4.3.2	JQuery: Write Less, Do More	98
5	What did we learn in IWGS-1?	101
II	IWGS-II: DH Project Tools	103
6	Semester Change-Over	105
6.1	Administrativa	105
7	Collaboration and Project Management	111
7.1	Revision Control Systems	111
7.1.1	Dealing with Large/Distributed Projects and Document Collections	111
7.1.2	Centralized Version Control	116
7.1.3	Distributed Revision Control	120
7.1.4	Working with GIT in small Projects	121
7.1.5	Working with GIT in large Projects	124
7.2	Working with GIT and GitLab/GitHub	125
7.2.1	Excursion: Authentication with SSH	127
7.3	Bug/Issue Tracking Systems	128
8	Databases	133
8.1	Introduction	133
8.2	Relational Databases	135
8.3	SQL – A Standardized Interface to RDBMS	137
8.4	ER-Diagrams and Complex Database Schemata	140
8.5	RDBMS in Python	143
8.6	Excursion: Programming with Exceptions in Python	145
8.7	Querying and Views in SQL	147
8.8	Querying via Python	149
8.9	Project: A Web GUI for a Books Database	151
9	Image Processing	157
10	Legal Foundations of Information Technology	203
10.1	Intellectual Property	203
10.2	Copyright	206
10.3	Licensing	209
10.4	Information Privacy	212

11 Ontologies, Semantic Web, & WissKI	215
11.1 Documenting our Cultural Heritage	215
11.2 Semantic Web Technologies	217
11.2.1 The Semantic Web	217
11.2.2 Semantic Networks	222
11.2.3 Ontologies	226
11.2.4 The Semantic Web Technology Stack	229
11.2.5 The Linked Open Data Cloud	236
11.3 The WissKI System: A Virtual Research Environment for Cultural Heritage	237
12 What did we learn in IWGS?	247

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
 - ▷ we will teach you all you need to know
 - ▷ You can do this course if you want!



©: Michael Kohlhase

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 ~ 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.

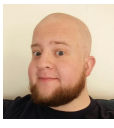

IWGS Tutorials

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



Teaching Assistants: (Doctoral Students in CS)

- ▷ Jonas Betzendahl: jonas.betzendahl@fau.de
- ▷ Philipp Kurth: 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 them as we go along (first time I teach IWGS)
 - ▷ 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
- ▷ 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 

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 Kohlhasse

6



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

7



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
- ▷ Web technologies for interactive documents and applications
 - ▷ Internet infrastructure: web browsers and servers
 - ▷ PHP, dynamic HTML, Javascript, HTML forms
- ▷ Web Application Project (design your own!)



©: Michael Kohlhase

8



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

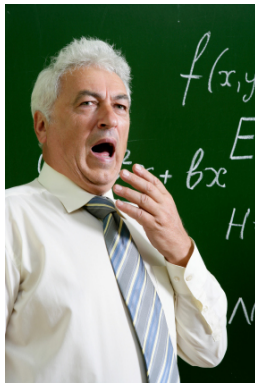
9



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 \leadsto take the following with “cum kilo salis”¹, 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

10



So there is a tension between

- 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?

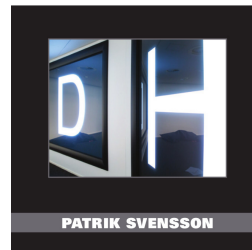
- ▷ We know how to keep large audiences engaged and motivated (*even televised*)
- ▷ But the topic is different (*IWGS is arguably more complex than Sports/Media*)

¹with much more than the proverbial grain of salt.



BIG DIGITAL HUMANITIES

IMAGINING A MEETING PLACE FOR THE HUMANITIES AND THE DIGITAL



- ▷ 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 Kohlhasse

11



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 Kohlhasse

12



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 Kohlhasse

13



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 Kohlhasse

14



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 arbitrary¹ 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

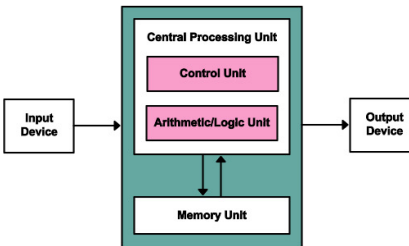
- i)* storing and manipulating data about the world,
- ii)* encoding, generating, and interpreting images, audio, and video,
- iii)* transporting information for communication,
- iv)* representing knowledge and reasoning,
- v)* transforming, optimizing, and verifying other programs,
- vi)* learning patterns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

[Computer Hardware/Software & Programming](#)

¹as long as they are “computable”, not all are.

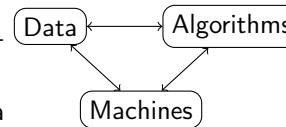
▷ **Definition 2.1.1 computer hardware** consists of devices that execute commands/instructions:

- ▷ **central processing unit (CPU)**
- ▷ **memory**: e.g. RAM, ROM, ...
- ▷ **storage**: e.g. Disks, SSD, tape, ...
- ▷ **input**: e.g. keyboard, touch-screen, ...
- ▷ **output**: e.g. screen, earphone, printer, ...



▷ **software** = **data** and **programs**

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



▷ hardware stores data and runs programs.

- ▷ Programming = writing programs (Telling the computer what to do)
- ▷ The computer does exactly as told
 - ▷ 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)



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** 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 computers (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”². It is instructional to think about what the program and the data might be in this analogy.

²Marvin Minsky; one of the founders of AI

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)

Programming Languages

- ▷ **Definition 2.1.2** 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



©: Michael Kohlhasse

16



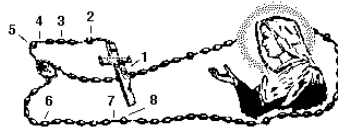
In computer science, we distinguish two levels on which we can talk about programs. The more general is the level of **algorithms**, which is independent of the concrete programming language. Algorithms express the general ideas and flow of computation and can be realized in various languages, but are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

▷ **Remark 2.1.3** Compilers are efficient, but more cumbersome for development.



©: Michael Kohlhasse

17



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.

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 2.1.4 (Deep Fundamental Result)** There are things no **computer** can compute.

▷ **Example 2.1.5** whether another program will terminate in finite time.

▷ **Remark 2.1.6 (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 2.1.7 (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)



©: Michael Kohlhasse

18



▷ Artificial Intelligence

- ▷ **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”.



©: Michael Kohlhasse

19



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 2.1.8** *Modern programming languages compose various primitives and give them a pleasing, concise, and uniform syntax.*
- ▷ **Question:** What does all of this even mean?
- ▷ **Definition 2.1.9** 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 2.1.10 (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 2.1.11** *The semantics of a programming language, is determined by the meaning of its primitives and composition principles.*
- ▷ **Definition 2.1.12** Programming language **syntax** concerns 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 2.1.13** *To understand a programming language, we (only) have to understand its primitives, **composition principle**, and their syntax.*
- ▷ **Definition 2.1.14** The “art of **programming**” consists of composing the primitives of a programming language.
- ▷ **Observation 2.1.15** *We only need very few – about half a dozen – primitives to obtain a Turing complete programming language.*
- ▷ **Observation 2.1.16** *The space of program behaviors we can achieve by programming is infinitely large nonetheless.*
- ▷ **Remark 2.1.17** More primitives make programming more convenient.
- ▷ **Remark 2.1.18** Primitives in one language can be composed in others.



A note on Programming: Little vs. Large Languages

- ▷ **Observation 2.1.19** *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.



2.1.2 Programming in IWGS

After the general introduction to “programming” in the last Subsection, 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 Kohlhase

23



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



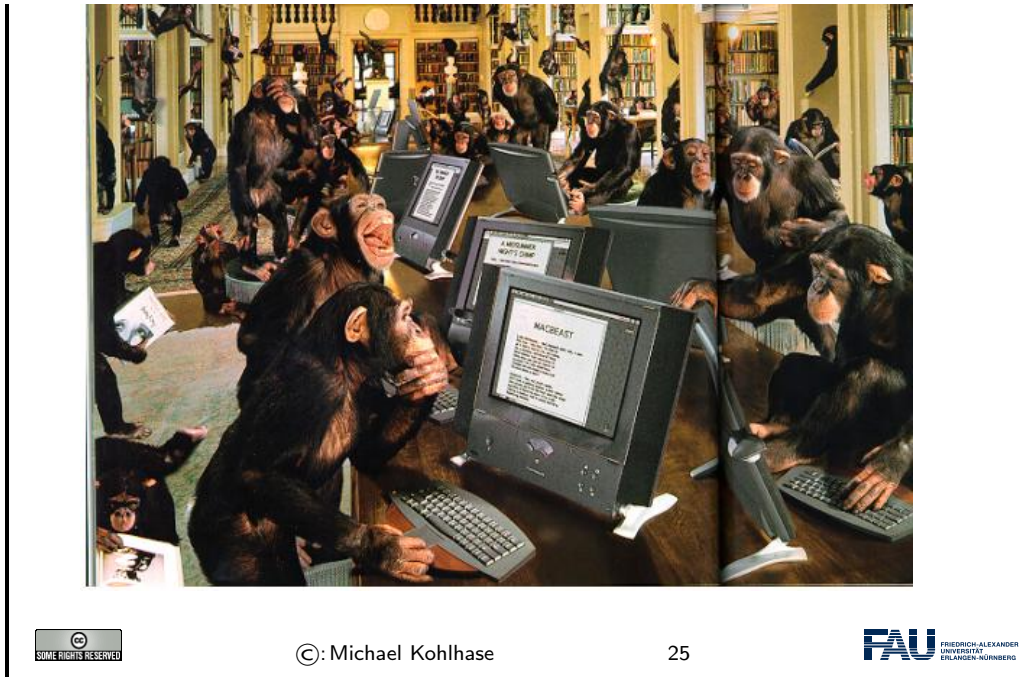
©: Michael Kohlhase

24



However, the result would probably be the following:

⚠ 2am in the Kollegienhaus CIP Pool ⚠



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 Crapletstm" (– ca. 1997)



©: Michael Kohlhasse

26



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

▷ **Homework:** Establish a python interpreter (version 3.7) (not 2.7.?, 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 Kohlhasse

27



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 shell¹, 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)

¹EDNOTE: 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

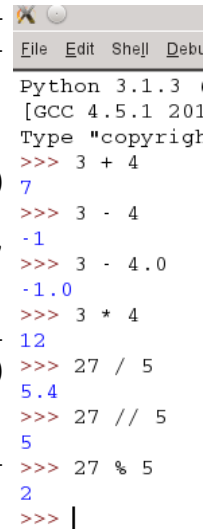
▷ **Floats** (numbers with a decimal point)

- ▷ **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 mostly use the pythonAnywhere 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 – a free “beginner’s account” will do fine for IWGS.

pythonAnywhere 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 pythonAnywhere for IWGS.
- ▷ **Homework**: Set up pythonAnywhere
 - ▷ make a “beginner’s account” at <http://pythonanywhere.com> (sufficient for IWGS)
 - ▷ give the IWGS account **iwgsTeacher** access under Account/Teacher.

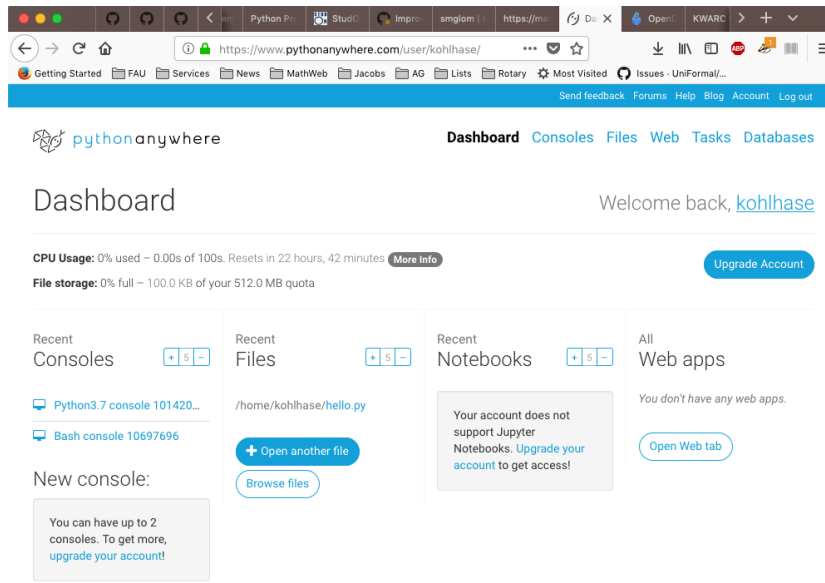


The advantage of a cloud IDE like pythonAnywhere for a course like IWGS is that you do not need any installation, cannot lose your files, and your teachers (the course instructor and the teaching assistants) can see (and even directly interact with) the your run time environment. This gives us a much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

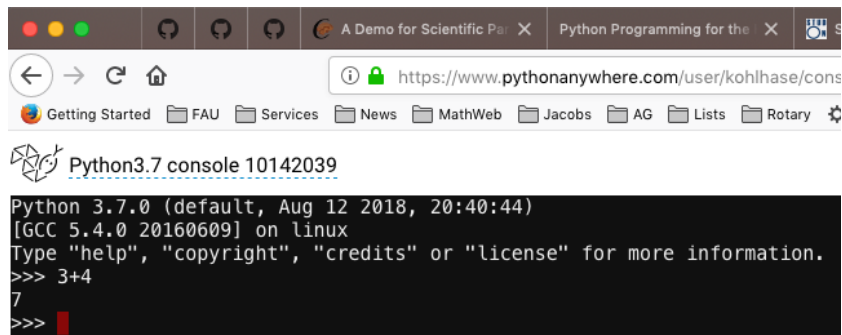
Both IDLE3 as well as pythonAnywhere come with an integrated editor for writing python programs. These editors gives you python syntax highlighting, and interpreter and debugger integration. In short, IDLE3 and pythonAnywhere are integrated development environments for python. Let us now go through the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

pythonAnywhere Components

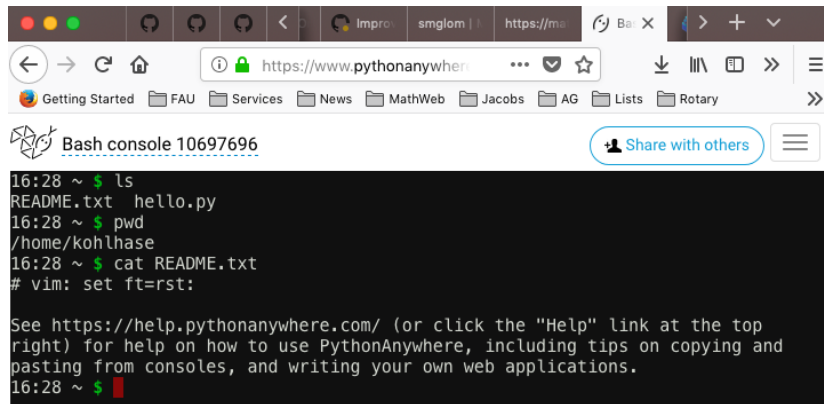
- ▷ The pythonAnywhere **dashboard** gives you access to all components



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



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



```

16:28 ~ $ ls
README.txt hello.py
16:28 ~ $ pwd
/home/kohlhase
16:28 ~ $ cat README.txt
# vim: set ft=rst:

See https://help.pythonanywhere.com/ (or click the "Help" link at the top
right) for help on how to use PythonAnywhere, including tips on copying and
pasting from consoles, and writing your own web applications.
16:28 ~ $

```

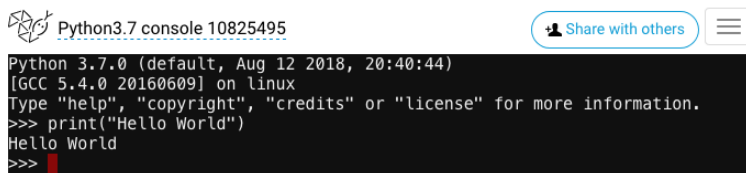
- ▷ **Useful shell commands:** See e.g. [All18] for a basic tutorial
 - ▷ `ls`: “list” the files in this directory
 - ▷ `mkdir`: “make” folder (called “directory”)
 - ▷ `pwd`: “print working directory” (where am I)
 - ▷ `cd <dirname>`: “change directory”
 - ▷ `<dirname>` = `..`: one up in the directory tree
 - ▷ empty `<dirname>`: go to your home directory.
 - ▷ `rm <filename>`, `cp/mv <filename> <newname>/<dirname>`: 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)



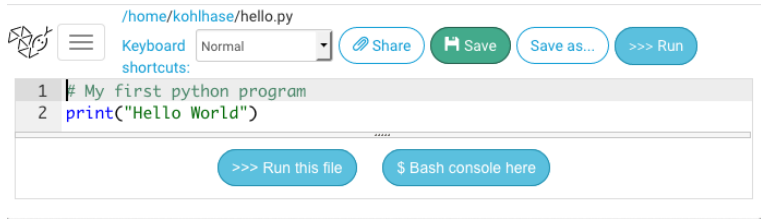
```

Python3.7 console 10825495
Python 3.7.0 (default, Aug 12 2018, 20:40:44)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
>>>

```

- ▷ **Alternatively:**
 1. got to the pythonAnywhere dashboard select “Files”, type file name `hello.py`, and press “New file”

2. Type your program,



3. Press the “Run” button (you may have to destroy a console first)

3' Alternatively go to your bash console and type

```
python hello.py
```

SOME RIGHTS RESERVED ©: Michael Kohlhase 31 **FAU** FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

We have seen that we can just call a program from the bash console, if we stored it in a file. In fact, we can do better: we can make our program behave like a native bash 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/python
```

which the bash console interprets as “call the program `python` on me”.

3. Finally, we make the file `hello` executable, i.e. tell the bash console 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 bash console 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 bash console and automate repetitive tasks.

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 2.2.1** A **variable** is a memory location which contains a value and an associated identifier – the **variable name**.
- ▷ note: In python a variable names
 - ▷ must start with letter or `_`,
 - ▷ cannot be python keywords
 - ▷ 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 2.2.2 (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 2.2.3 (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 2.2.4 (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 2.2.4; 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 into types
 - ▷ 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 2.2.5** python has the following five basic **types**

Data type	Name	Examples
Integers	int	1, -5, 0, ...
Floats	float	1.2, .125, -1.0, ...
Strings	str	"Hello", 'Hello', "123", 'a', ...
Booleans	bool	True, False
Complex numbers	complex	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 2.2.6** 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 2.2.7**

```
>>> 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 2.2.2 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.

Branching and Looping

- ▷ **Problem:** Up to now programs seem to execute all the instructions in sequence, from the first to the last (a **linear program**)
- ▷ **Idea:** Change the **control flow**, i.e. the sequence of execution of the program instructions via **control structures**.
- ▷ **Definition 2.2.8 Branching** (or **conditional execution**) allows to execute (or not to execute) certain parts of a program – the **branches**– depending on **conditions**. We call a code block that enables branching a **conditional statement**.
- ▷ **Definition 2.2.9 Looping** allows to execute certain parts of a program – the **body**– multiple times depending on **conditions**. We call the code block that specifies looping a **loop**.
- ▷ **Definition 2.2.10** A **condition** (or **Boolean expression**) is an expression that can be evaluated to True or False.
- ▷ **Example 2.2.11** 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 $x < 3$



©: Michael Kohlhasse

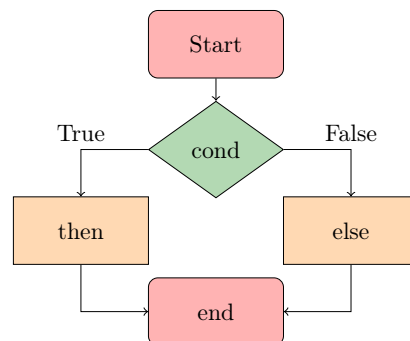
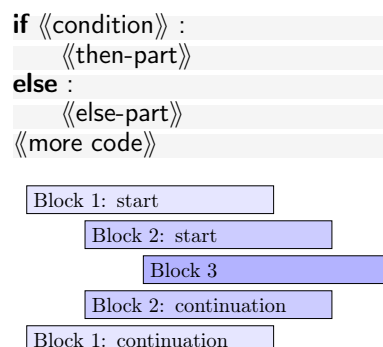
38



After this general introduction – branching and looping) are supported by all programming language in some form – we will see how this is realized in **python**

Branching in python

- ▷ **Definition 2.2.12** Branching via if/else statements



- ▷ «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 branching in action.

Branching Example

- ▷ **Example 2.2.13 (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 branching 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 Branching

- ▷ **Variant:** multiple branching 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 branching could achieved by nested if/else structures.
- ▷ **Example 2.2.14 (Better Empathy in python)** In Example 2.2.13 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 looping in python.

Looping in python

▷ Definition 2.2.15 looping via **while**-blocks

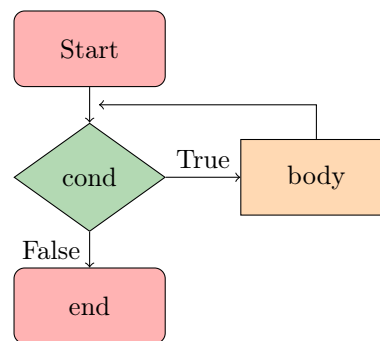
▷ syntax of the **while** loop

```
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.

Looping Examples

▷ Example 2.2.16 (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 2.2.17 (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▷ **Example 2.2.18 (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)
```



©: Michael Kohlhasse

43



Example 2.2.16 and Example 2.2.16 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

```
count = -10
prod = 1
while count < 1000000:
    prod *= count
    count += 1
```

More Efficient

```
count = -10
prod = 1
while count <= 1000000:
    prod *= count
    if count == 0 :
        break
    count += 1
```

2.2.4 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 2.2.19** A **list** is a finite sequence of objects.
- ▷ In programming languages, lists are used for locally storing and passing around collections of objects.
- ▷ In python lists can be written as a list of comma-separated expressions between square brackets.
- ▷ **Example 2.2.20 (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 2.2.21** List elements can be accessed by specifying ranges

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

▷ **Example 2.2.22** Lists can be constructed by python functions

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

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



Range objects are useful, because they are easily and flexibly constructed for [iteration](#) (up next).

Iterating over Lists/Sequences in python

▷ **Definition 2.2.23** 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 2.2.24**

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

▷ Lists and strings can also act as ranges

Example 2.2.25

```
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

Other Sequences in python. e.g Dictionaries

▷ **Definition 2.2.26** A *dictionary* is an unordered, indexed collection of or-

dered 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 2.2.27** Dictionaries can be used for various purposes,

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

- ▷ 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 2.2.28 (Printing Keys and Values)**

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

- ▷ more dictionary commands

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



2.2.5 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)



©: Michael Kohlhasse

48



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

- ▷ **Definition 2.2.29** A **file** is a resource for recording data in a storage device.
- ▷ **Definition 2.2.30** 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 2.2.31** 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.



©: Michael Kohlhasse

49



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 ≡ 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 2.2.32 (Duplicating the contents of a file)**

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

▷ **Example 2.2.33 (Reading a file linewise)**

<pre>>>> f.readline() 'This is the first line of the file.\n' >>> f.readline() 'Second line of the file\n' >>> f.readline() ''</pre>	<pre>>>> 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)



2.2.6 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 2.2.34**

```
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)

- ▷ **Example 2.2.35 (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 2.2.36** 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 pi: document arguments}
    """
    <<body>> # it can contain p1, ..., pn, and 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 .



Anonymous and Higher-Order Functions (lambda)

- ▷ **Observation 2.2.37** A python function definition combines making a function object with giving it a name.

- ▷ **Definition 2.2.38** python also allows to make **anonymous functions** via the lambda constructor for **function objects**:

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

- ▷ **Example 2.2.39** 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)



Higher-Order Functions in python

- ▷ **Definition 2.2.40** We call a function a **higher-order function**, iff it takes a function as argument.
- ▷ **Definition 2.2.41** `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 2.2.42** Mapping over and filtering a list

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



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 2.2.5 are [classes](#), ...
- ▷ **Definition 2.2.43** In python all values belong to a [class](#), which provide special functions we call [methods](#). Values are also called [objects](#), to em-

phasise class aspects. **Method** application is written with **dot notation**:
`⟨obj⟩.⟨meth⟩(⟨args⟩)` corresponds to `⟨meth⟩(⟨obj⟩,⟨args⟩)`.

▷ **Example 2.2.44** 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
```

▷ **Example 2.2.45 (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



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 2.2.46** 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 ≥ 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](#))



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.

2.2.7 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

57



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

Documents as Digital Objects

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.

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

3.1 Preliminaries: Data Structures, Documents, and Sizes

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

58



3.1.1 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.

Natural Numbers

- ▷ Numbers are symbolic representations of numeric quantities.
- ▷ There are many ways to represent numbers (more on this later)
- ▷ let's take the simplest one (about 8,000 to 10,000 years old)



- ▷ we count by making marks on some surface.
- ▷ For instance `////` stands for the number four (be it in 4 apples, or 4 worms)

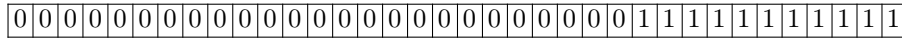


In addition to manipulating normal objects directly linked to their daily survival, humans also invented the manipulation of place-holders or symbols. A *symbol* represents an object or a set of objects in an abstract way. The earliest examples for symbols are the cave paintings showing iconic silhouettes of animals like the famous ones of Cro-Magnon. The invention of symbols is not only an artistic, pleasurable “waste of time” for humans, but it had tremendous consequences. There is archaeological evidence that in ancient times, namely at least some 8000 to 10000 years ago, humans started to use tally bones for counting. This means that the symbol “bone with marks” was used to represent numbers. The important aspect is that this bone is a symbol that is completely detached from its original down to earth meaning, most likely of being a tool or a waste product from a meal. Instead it stands for a universal concept that can be applied to arbitrary objects.

So far so good, let us see how this would be represented on a computer:

Unary Natural Numbers on the Computer

- ▷ **Definition 3.1.1** We call the representation of natural numbers by slashes on a surface the **unary natural numbers**
- ▷ **Question:** How do we represent them on a computer? (not bones or walls)
- ▷ **Idea:** If we have a memory bank of n binary digits, initialize all by 0, represent each slash by a 1 from the right.
- ▷ **Example 3.1.2** Memory bank with 32 binary digits, representing 11.



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 /)



©: Michael Kohlhase

60



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 φ_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 $i \bmod b$, $i := i \div b$) until $i = 0$.

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

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



©: Michael Kohlhase

61



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 φ_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 φ_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	/	////// ₁
binary	\mathbb{N}_2	2	0,1	0101000111 ₂
octal	\mathbb{N}_8	8	0,1,...,7	63027 ₈
decimal	\mathbb{N}_{10}	10	0,1,...,9	162098 ₁₀ or 162098
hexadecimal	\mathbb{N}_{16}	16	0,1,...,9,A,...,F	FF3A12 ₁₆

- ▷ **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**

<p>Addition base 4</p> $ \begin{array}{r} 1 2 3 \\ + 1_1 2_1 \\ \hline 3 1 2 \end{array} $	<p>binary multiplication</p> $ \begin{array}{r} 0 0 \\ * 1 0 \\ \hline 0 0 0 \\ 0 1 0 \\ \hline 0 1 0 \\ 1 1 0 \\ \hline 1 1 0 \\ 1 1 0 \\ \hline 1 1 0 \end{array} $
---	---



©: Michael Kohlhasse

63



3.1.2 Characters and their Encodings

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 Subsection 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 3.1.10** 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

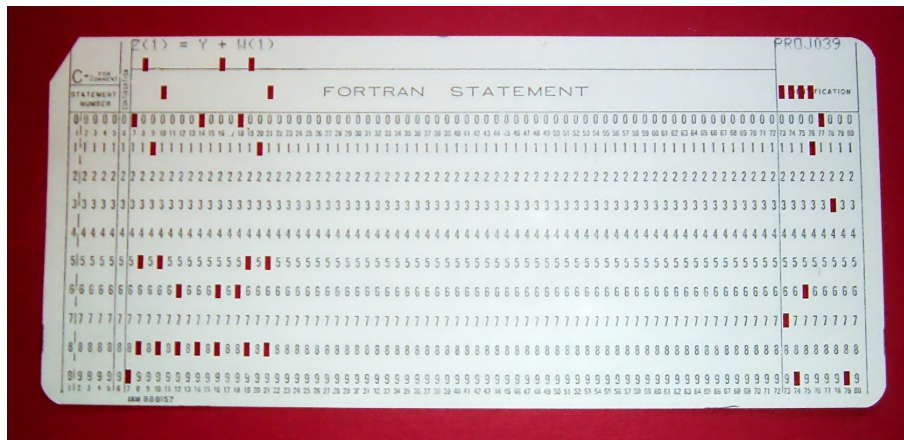
- ▷ **Motivated by punchcards:** The character 0 (binary 0000000) carries no information NUL, (used as dividers)
 Character 127 (binary 1111111) can be used for deleting (overwriting) last value (cannot delete holes)
- ▷ The ASCII code was standardized in 1963 and is still prevalent in computers today (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 3.1.11** 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.

Playing with Strings and Characters in python

- ▷ **△**: in python, characters are just strings of length 1.
- ▷ **ord** gives the ASCII number of the character, **chr** ASCII character for a number.
- ▷ **Example 3.1.12 (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 $[i:j]$

$([i] \hat{=} [i:i])$

▷ **Example 3.1.13** taking strings apart and re-assembling them.

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



©: Michael Kohlhase

66



⚠ **Note:** Example 3.1.12 and Example 3.1.13 (or any other examples in this lecture) is not production code, but didactially motivated – to show you what you can do with the objects we are presenting in `python`.

In particular, if we “lowercase” a character that is already lowercase – e.g. by `lc('c')`, then we get out of the range of the ASCII code: the answer is `\x83`, which is the character with the hexadecimal code 83 (decimal 130).

In production code (e.g. the `python lower` method), we would have some range checks, etc.

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 3.1.14 (German ASCII)** remap e.g. `[↦ Ä,] ↦ Ü` in German ASCII
 (“`Apple]`” comes out as “`Apple Ü`”)
- ▷ **Definition 3.1.15 (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

67



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 3.1.16 (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 3.1.17** 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 3.1.18** Most (non-Chinese) characters have code points in [1, 65536] (the **basic multilingual plane**).
- ▷ **Notation 3.1.19** 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;



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 3.1.20** 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 3.1.21 (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 3.1.22** The UTF-8 encoding follows the following encoding scheme

Unicode	Byte1	Byte2	Byte3	Byte4
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 3.1.23** \$ = 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)



©: Michael Kohlhasse

69



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..

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

Programming with UniCode strings is particularly simple, strings in `python` are UTF-8-encoded UniCode strings and all operations on them are UniCode-based¹ This makes the introduction to UniCode in `python` very short, we only have to know how to produce non-ASCII characters – which are on regular keyboards.

Unicode in python

▷ the python `str` 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'
```



©: Michael Kohlhasse

70



3.1.3 Computing with Strings

In this Subsection we introduce methods to automatically deal with documents – actually large strings for the moment. We introduce “regular expressions”, a domain-specific language for locating substrings of a particular form in a document. Regular expressions are useful in many document-related tasks, e.g. advanced searching and replacing, therefore most programming languages – python is no exception – integrate them as a sublanguage.

¹Older programming languages have ASCII strings only, and UniCode strings are supplied by external modules.

Before we go into [regular expressions](#), we will extend our repertoire on handling and formatting strings: we will introduce [string literals](#), which allow writing complex strings.

String Literals in python

- ▷ **Problem:** How to write strings including special characters?
- ▷ **Definition 3.1.24** python uses [string literals](#), i.e. character sequences surrounded by one, two, or three sets of matched single or double quotes for string input. The content can contain [escape sequences](#), i.e. the [escape character](#) backslash followed by a code character for problematic characters:

Seq	Meaning	Seq	Meaning
\\	Backslash (\)	\'	Single quote (')
\"	Double quote (")	\a	Bell (BEL)
\b	Backspace (BS)	\f	Form-feed (FF)
\n	Linefeed (LF)	\r	Carriage Return (CR)
\t	Horizontal Tab (TAB)	\v	Vertical Tab (VT)

In triple-quoted string literals, unescaped newlines and quotes are honored, except that three unescaped quotes in a row terminate the literal.

Prefixing a string literal with a `r` or `R` turns it into a [raw string literal](#), in which backslashes have no special meaning.

- ▷ **Note:** using the backslash as an escape character forces us to escape it as well.
- ▷ **Example 3.1.25** The string `"a\nb\nc"` has length five and three lines, but the string `r"a\nb\nc"` only has length seven and only one line.



Formatted String Literals (aka. f-strings)

- ▷ **Definition 3.1.26** [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 `{/}` are represented by `{{/}}`.

- ▷ **Example 3.1.27 (An f-String for IWGS)**

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

- ▷ **Example 3.1.28 (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.'
```

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
```



©: Michael Kohlhase

72



Now we can come to the main topic of this Subsection: [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 3.1.29** A [regular expression](#) (also called [regex](#)) is a formal expression that specifies a set of strings.

▷ **Definition 3.1.30 (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^{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 `\`: `\?`.



©: Michael Kohlhase

73



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

Regular Expression Examples

▷ **Example 3.1.31 (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 3.1.32 (A more complex example)** The following regexp times in a variety of formats, such as 10:22am, 21:10, 08h55, and 7.15 pm.

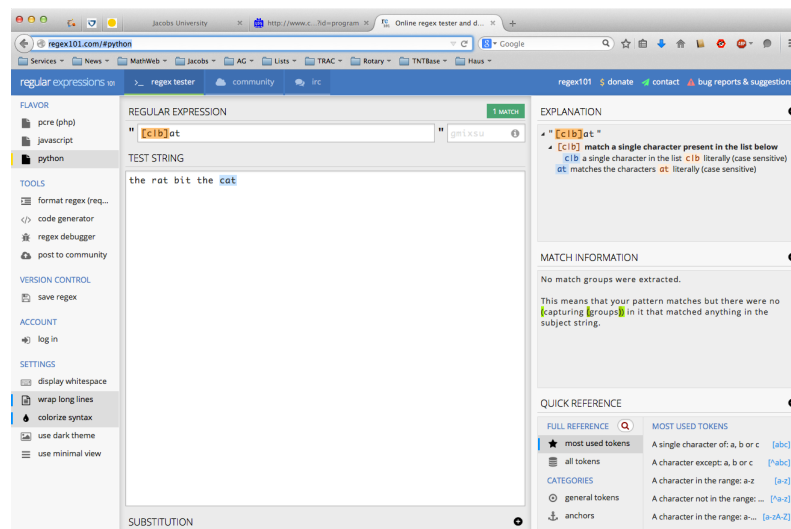
```
^(?:([0]?[0-9]|1[012])|(?:(?:1[3-9]|2[0-3]))):([0-5]?[0-9])\d(?:\s?(?:1(am|AM|pm|PM)))?$$
```



As we have seen regular expressions can become quite cryptic and long (cf. e.g. Example 3.1.32), 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, go e.g. to <http://regex101.com>



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 *pmetavarpat*.


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



We will now see what we can do with regular expressions in a practical example.

Example: Correcting and Anonymizing Documents

- ▷ **Example 3.1.33** We write a `corranon` 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 famILLiar with Paul Erdős or Henri Poincaré?* (Unicode)

Here is the function

- ▷ we first add blanks after commata


```
def corranon (s)
    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),r" ".upper()+m.group(2),
            s)
```
- ▷ 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}+)(l)",
          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

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?



3.1.4 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.

File Types

▷ **Definition 3.1.34** A **text file** is a computer file that is structured as a sequence of encoded characters. Computer files that are not text files are called **binary files**.

▷ **Remark 3.1.35** Text files are usually encoded with ASCII, ISO-Latin, or – increasingly – UniCode encodings like UTF-8.



Remark 3.1.36 **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.)

Digital Text

▷ **Definition 3.1.37** **Digital text** is a digital encoding of textual material that can be read without much processing.

▷ **Definition 3.1.38** Digital text is subdivided into **plain text**, where all characters carry the textual information and **formatted text**, which also contains **markup codes**.

▷ Even though formatted text can read directly, it is usually consumed by humans through a **document renderer**, i.e. a device that interprets the **control words** and visualizes the textual content accordingly.

▷ **Remark 3.1.39** **Document markup** turns plain text into formatted text.



Text Editors

- ▷ **Definition 3.1.40** A **text editor** is a program used for editing **text files**.
- ▷ **Example 3.1.41** 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 3.1.42** 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**.



Word Processors and Formatted Text

- ▷ **Definition 3.1.43** A **word processor** is a software application, that performs the task of composition, editing, formatting, printing of documents represented as formatted text. The particular representation format is called the **document format**.
- ▷ **Example 3.1.44** Popular word processors include
 - ▷ MS Word is an elaborated word processor for Windows, whose native format is **Office Open XML** (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 3.1.45** **Text editors** 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.

3.1.5 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 Subsection 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 3.1.46** 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 3.1.47** The **byte** is a derived **unit** for information capacity: 1 B = 8 b.



©: Michael Kohlhasse

82



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 3.1.48 (Binary Prefixes)** The following **binary unit prefixes** 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}	1.074×10^9	giga	G
tebi	Ti	2^{40}	1.1×10^{12}	tera	T
pebi	Pi	2^{50}	1.125×10^{15}	peta	P
exbi	Ei	2^{60}	1.153×10^{18}	exa	E
zebi	Zi	2^{70}	1.181×10^{21}	zetta	Z
yobi	Yi	2^{80}	1.209×10^{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 3.1.49** You can buy hard-disks that say that their capacity is “one tera-byte”, but they actually have a capacity of one tebibyte.



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

How much Information?

Bit (b)	<i>binary digit 0/1</i>
Byte (B)	<i>8 bit</i>
2 Bytes	A Unicode character in UTF.
10 Bytes	your name.
Kilobyte (k B)	<i>1,000 bytes OR 10^3 bytes</i>
2 Kilobytes	A Typewritten page.
100 Kilobytes	A low-resolution photograph.
Megabyte (M B)	<i>1,000,000 bytes OR 10^6 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.
Gigabyte (G B)	<i>1,000,000,000 bytes or 10^9 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.



How much Information?

Terabyte (T B)	<i>1,000,000,000,000 bytes or 10^{12} 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.
Petabyte (P B)	<i>1,000,000,000,000,000 bytes or 10^{15} 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).
Exabyte (E B)	<i>1,000,000,000,000,000,000 bytes or 10^{18} 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.
Zettabyte (Z B)	<i>1,000,000,000,000,000,000,000 bytes or 10^{21} bytes</i>
2 Zettabytes	Total volume digital data transmitted in 2011
100 Zettabytes	Data equivalent to the human Genome in one body.



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.

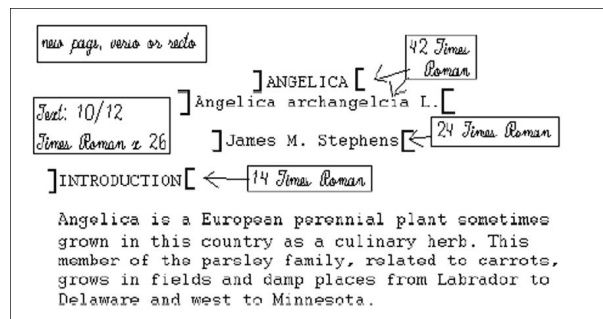
3.2 Multimedia Documents on the World Wide Web

We have seen the client-server infrastructure of the [WWW](#), which essentially specifies how hypertext documents are retrieved. Now we look into the documents themselves.

In [?character-encodings?](#) have already discussed how texts can be encoded in files. But for the rich documents we see on the [WWW](#), we have to realize that documents are more than just sequences of characters. This is traditionally captured in the notion of document markup.

Document Markup

- ▷ **Definition 3.2.1** **Document markup** is the process of adding **control words** (special character sequences also called **markup codes**) to a document to control the structure, formatting, or the relationship among its parts.
- ▷ **Definition 3.2.2** The vocabulary and composition rules for a particular kind of document 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**.
- ▷ **Example 3.2.3** A text with markup codes (for printing)



There are many systems for document markup, ranging from informal ones as in [?document-markup.ex?](#) 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.

3.2.1 Hypertext Markup Language

[WWW](#) documents have a specialized **markup language** 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.

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

▷ **Definition 3.2.5 (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 3.2.6** 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 introduce 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.

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 Kohlhasse<br/>Jacobs University Bremen</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>3. Conclusion</h2>
<p> As we have seen in the <a href="#intro">introduction</a> this
was very easy.</p>
</body>
</html>

```



©: Michael Kohlhase

88



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)

Anatomy of a HTML Web Page

Michael Kohlhase
Jacobs University Bremen

1. Introduction

This is really easy, just start writing

3. Main Part: show off features

We can can markup *text styles* inline.

And we can make itemizations:

- with a list item
- and another one

3. Conclusion

As we have seen in the [introduction](#) this was very easy.



©: Michael Kohlhase

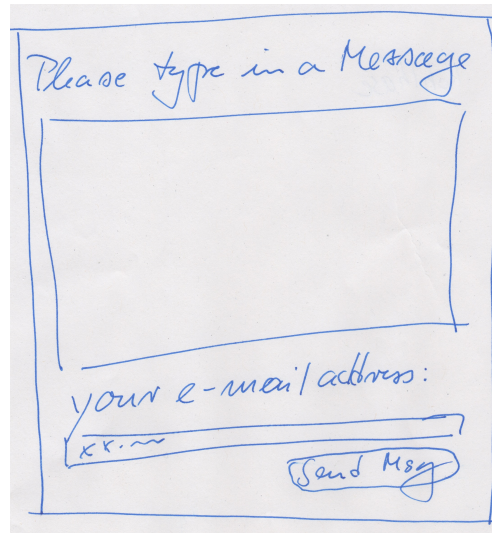
89



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 a 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

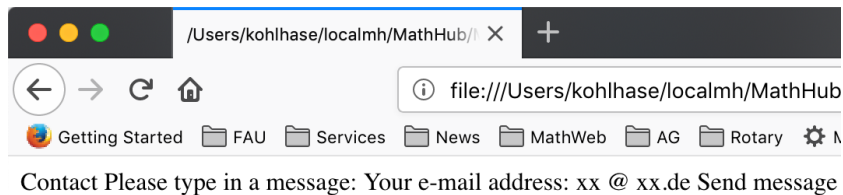


- ▷ put the intended text into a file: contact.html

```

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

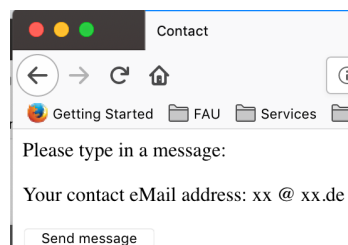
- ▷ load into your browser to check the state



- ▷ 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>
  
```

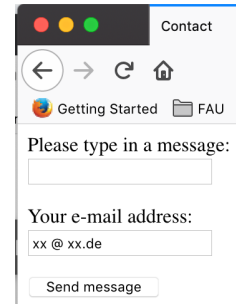


- ▷ add input fields and breaks:

```

<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):

```

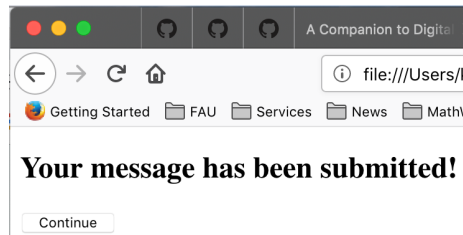
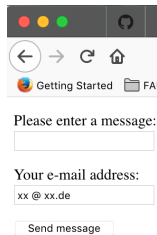
<title>Contact</title>
<form action="contact-after.html">
  <h2>Please enter a message:</h2>
  <input name="msg" type="text"/>
  <h3>Your e-mail address:</h3>
  <input name="addr" type="text"
        value="xx @ xx.de"/>
  <br/>
  <input type="submit"
        value="Send message"/>
</form>

```

```

<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)



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 3.2.7** The HTML form element groups the layout and input elements:
 - ▷ `<form action="⟨URI⟩">` specifies the **form action**.
 - ▷ `<input type="submit".../>` triggers the form action: it sends a query $?n_1=v_1 \& \dots \& n_k=v_k$ to the page at $\langle \text{URI} \rangle$, 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.
- ▷ **Example 3.2.8 (In the Contact Form)** We send the request

contact-after.html?msg=Hi&addr=foo@bar.de

The query part after the ? is currently ignored

- ▷ We will come to the full story of processing actions later.



©: Michael Kohlhasse

91



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
```

☐ Male
☐ Female
☐ 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
```

My major is ☐ Computer Science ☐ Digital Humanities ☐ Other

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

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

Upload your resume No file selected.

- ▷ 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>
```

Which animal d
☒ Bird
☐ Hamster
☐ Cat
☐ Dog



©: Michael Kohlhasse

92



3.2.2 Cascading Stylesheets

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 3.2.9** The **Cascading Style Sheets** (CSS), is a style sheet language that allows authors and users to attach style (e.g., fonts and spacing) to structured documents.
- ▷ **Example 3.2.10** Our text file from Example 3.2.6 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>
```



Now that we have seen the example, let us fix the basic terminology of CSS.

CSS: Rules, Selectors, and Declarations

- ▷ **Definition 3.2.11** A CSS style sheet consists of a sequence of **rules** that in turn consist of a set of **selectors** that determine which elements the rule applies to and a **declaration block** that specifies styling information.
- ▷ **Definition 3.2.12** 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 3.2.13** In Example 3.2.10 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.

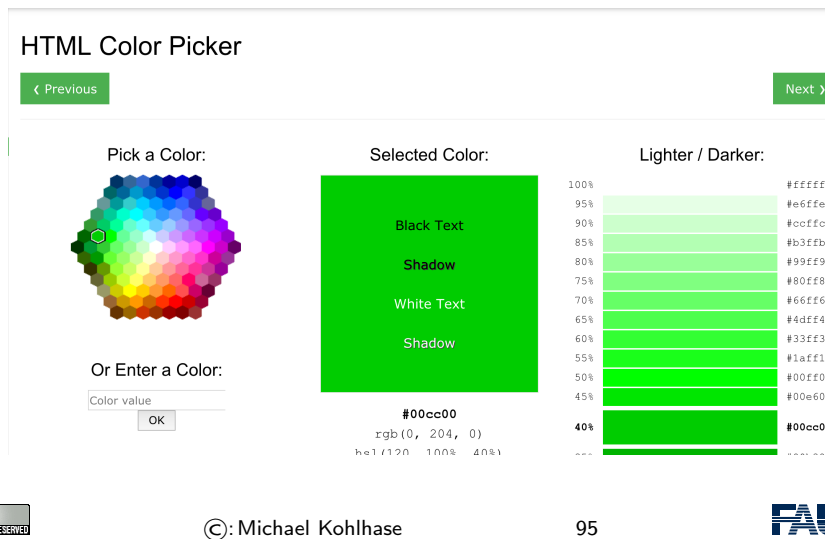


In the example on the last slide, we specified the background color of the page as `#d0e4fe;`, which

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



Again, 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 3.2.14 (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 Kohlhase</span>
    <span class="affil">FAU Erlangen–Nuernberg</span>
  </div>
</div>
...
```

And the CSS file referenced in line three:

```
.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 Kohlhasse

96



And here is the result in the browser:

A Styled HTML Title Box (Result)



©: Michael Kohlhasse

97



We will now go over a useful fragment of CSS in more detail and introduce it by example. For a more complete introduction, see e.g. [CSSb].

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 $\hat{=}$ $\langle\langle\text{elname}\rangle\rangle$, attributes $\hat{=}$ $[\langle\langle\text{attname}\rangle\rangle=\langle\langle\text{attval}\rangle\rangle]$
 - ▷ **Example:** `p[xml:lang='de']` applies to `<p lang="de">...</p>`
- ▷ Any elements with a given class attribute
 - ▷ **Selector:** `.<elname>`
 - ▷ **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



©: Michael Kohlhasse

98



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

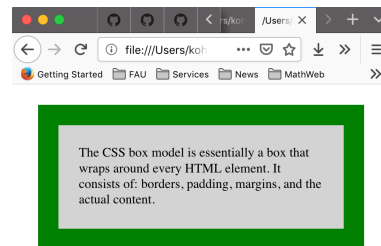
▷ **Definition 3.2.15** For layout, CSS considers all HTML elements as boxes, i.e. document areas with a given **width** and **height**. A CSS **box** has four parts:

- ▷ **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;
}
```



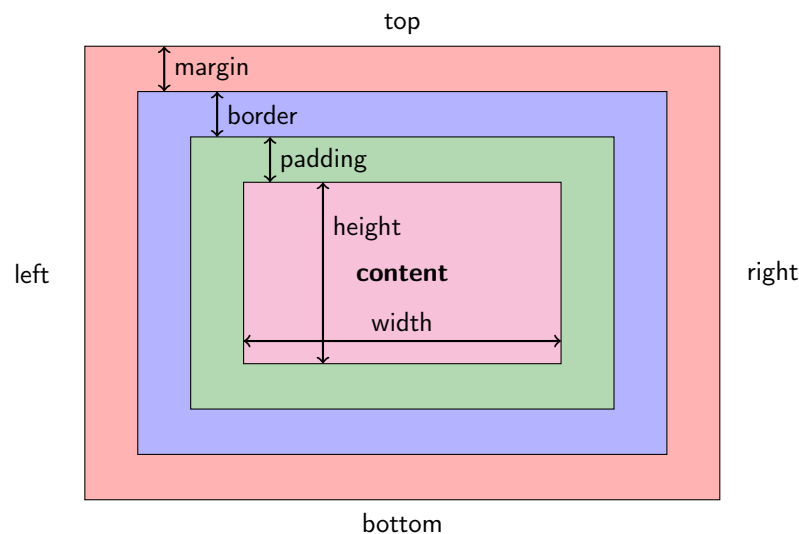
Note that the overall width of the CSS box is 375 pixels.



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. [CSSa])
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 106 for details).

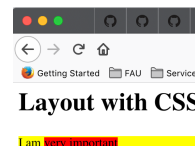
We now look at an example to fortify our intuition.

Cascading of selectors in CSS: Prioritization Example

▷ **Example 3.2.16** 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}
```



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 3.2.17** 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

103



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.

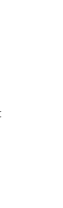
CSS-Flow: How Boxes Flow to their Place

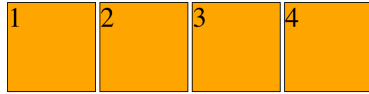
▷ CSS-Flow describes how different elements are distributed in the visible area
(*how they flow; hence the name*)

▷ **Example 3.2.18** Block-level Boxes (here `divs`) flow to the left

<code><div class="square">1</div></code>	+	<code>.square {font-size:200%;</code>	=	
<code><div class="square">2</div></code>		<code>height:100px;</code>		
<code><div class="square">3</div></code>		<code>width:100px;</code>		
<code><div class="square">4</div></code>		<code>border:1px solid black;</code>		
		<code>margin:2px;</code>		
		<code>background-color:orange;}</code>		

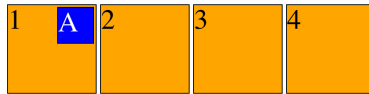
▷ **Example 3.2.19** `float:left` floats boxes as far as they will go
(*without overlap*)

<code><body></code>	+	<code>.square {font-size:200%;</code>	=	
<code><div class="square">1</div></code>		<code>height:100px;</code>		
<code><div class="square">2</div></code>		<code>width:100px;</code>		
<code><div class="square">3</div></code>		<code>border:1px solid black;</code>		
		<code>margin:2px;</code>		
		<code>background-color:orange;</code>		
		<code>float:left}</code>		



▷ **Example 3.2.20** float:right in a div will float inside the corresponding box

<pre><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></pre>	+	<pre>.smallsq {color:white; height: 40px;width: 40px; border: 1px solid black; margin: 2px; background-color: blue; float: right}</pre>	=
--	---	---	---



▷ **Example 3.2.21** `float:left` will let contents flow around an obstacle

<pre><div class="square" style="font-size:small"> <div class="smallsq">A</div> flow, flow, flow, flow, flow, flow, flow, flow, flow, flow. </div></pre>	+	<pre>.smallsq {color:white; height: 40px;width: 40px; border: 1px solid black; margin: 2px; background-color: blue; float: right}</pre>	=
---	---	---	---



The large space (>2px) is caused because there is no linebreaking

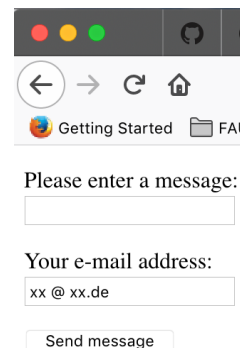


To fortify our intuition on CSS, we take up the “contact form” example from above and improve the layout in a step-by-step process concentrating on one aspect at a time.

CSS in Practice: Worked Example

▷ Recap: The unstyled contact form:

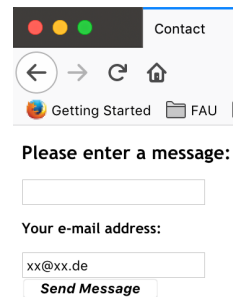
```
<title>Contact</title>
<form action="contact-after.html">
  <h2>Please enter a message:</h2>
  <input name="msg" type="text"/>
  <h3>Your e-mail address:</h3>
  <input name="addr" type="text"
    value="xx @ xx.de"/>
  <br/>
  <input type="submit"
    value="Send message"/>
</form>
```



- ▷ Add a CSS file with font information

```
<link rel="stylesheet" type="text/css"
href="csscontact1.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;}
```

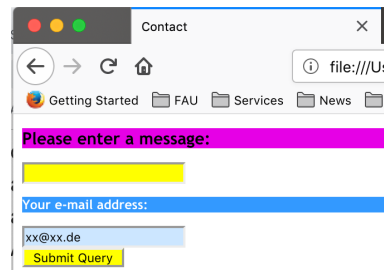


▷ Add lots of color

(oops, what about the size)

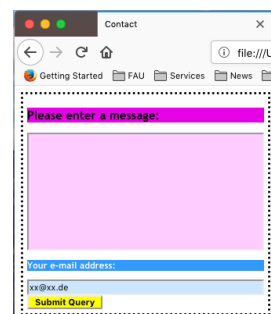
```
<h2>Please enter a message:</h2>
<h3>Your e-mail address:</h3>
<input class="important" name="addr"
style="background-color:#cce6ff"
type="text" value="xx@xx.de"/>
```

```
h2 {background-color: #e60e6;}
h3 {background-color: #3399ff;
color: white;}
input{background-color:yellow}
```



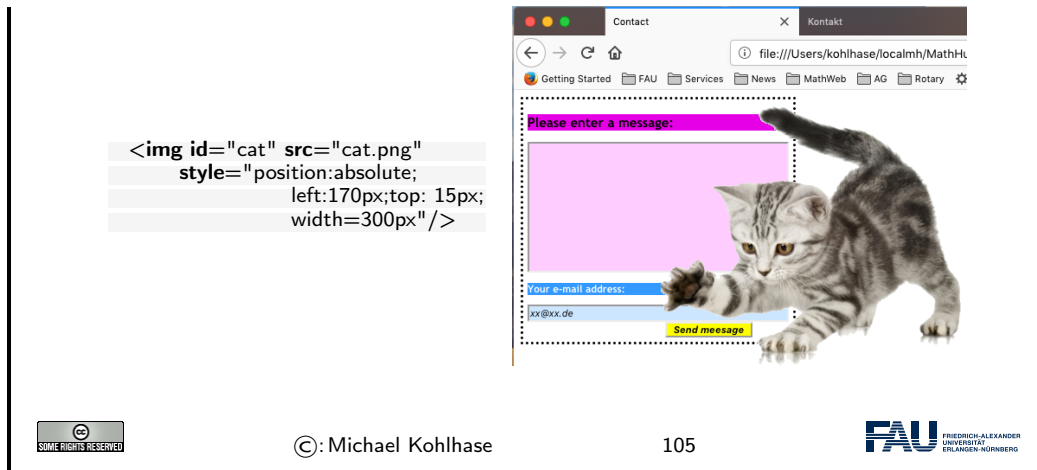
▷ Add size information and a dotted frame

```
<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.

But how to find out what the browser really sees?

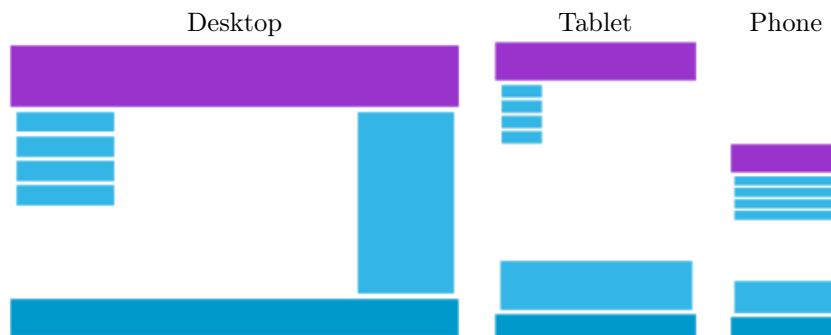
- ▷ CSS has many interesting inheritance rules
- ▷ **Definition 3.2.22** The **page inspector** tool gives you an overview over the internal state of the browser.
- ▷ **Example 3.2.23**



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 3.2.24** **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 3.2.25** A **web page** with content blocks



Implementation: CSS-based layout with relative sizes and **media queries** – CSS conditionals based on client screen size/resolution/...



3.3 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.

▷ Excursion: XML (EXtensible Markup Language)

- ▷ XML is language family for the Web
 - ▷ 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 & simplified SGML

- ▷ logic annotation (*markup*) instead of presentation!
- ▷ many tools available: parsers, compression, data bases, ...
- ▷ **conceptually:** transfer of directed graphs instead of strings.

▷ details at <http://www.w3c.org>



©: Michael Kohlhasse

108



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. document metadata)

▷ **Example 3.3.1** Open a PDF file in Acrobat Reader, then click on *File* \ Document Properties \ Document 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>
```



©: Michael Kohlhasse

109



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.

XML is Everywhere (E.g. Web Pages)

▷ **Example 3.3.2** Open web page file in FireFox, then click on *View* \ PageSource, you get the following text: (showing only a small part and reformatting)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Michael Kohlhasse</title>
    <meta name="generator"
      content="Page generated from XML sources with the WSMML 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 3.3.3** XHTML is the XML version of HTML (just make it valid XML)



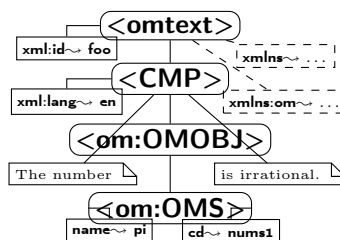
XML Documents as Trees

- ▷ **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 3.3.4** The XML document tree is made up of element nodes, attribute nodes, text nodes (and namespace declarations, comments,...)
- ▷ **Definition 3.3.5** For communication this tree is serialized into a balanced bracketing structure, where
- ▷ an element is represented by the brackets <el> (called the opening tag) and </el> (called the closing tag).
 - ▷ The leaves of the tree are represented by empty elements (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.



▷ Trees in Computer Science

- ▷ **Observation 3.3.6** 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

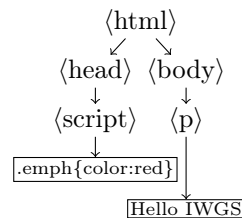
- ▷ **Observation 3.3.7** In well-bracketed structures, brackets contain two kinds of objects

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

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

- ▷ **Example 3.3.8** 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>
```



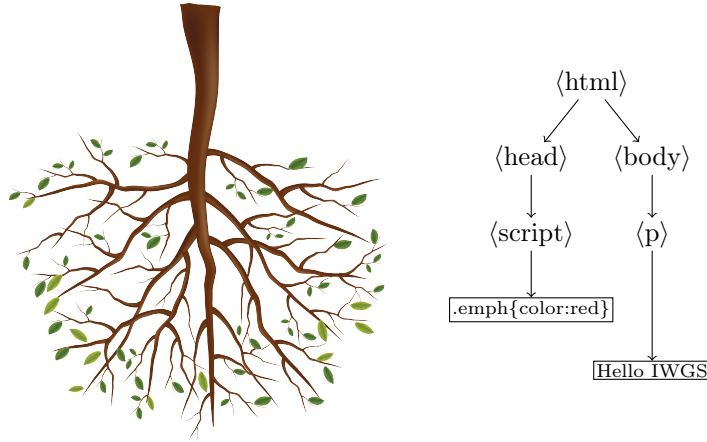
Well-Bracketed Structures: Tree Nomenclature

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

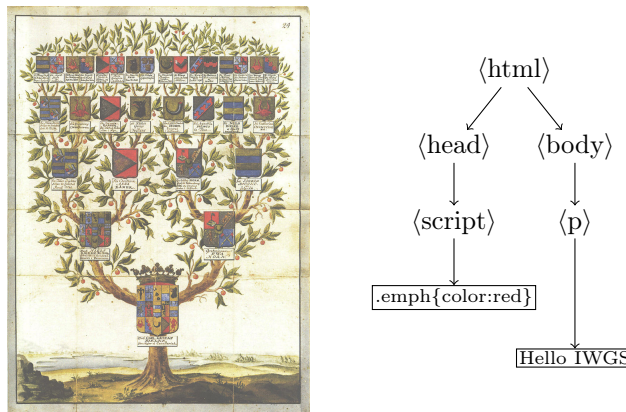
branches, leaves, and height).

(but written upside-down)

▷ **Example 3.3.9** In a tree, there is only one path from the root to the leaves



▷ **Definition 3.3.10** We speak of **parent**, **child**, **descendent**, and **ancestor** 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.

The Document Object Model

▷ **Definition 3.3.11** The **document object model (DOM)** is a data structure for storing documents as marked-up documents as document trees together with a standardized set of access methods for manipulating them.

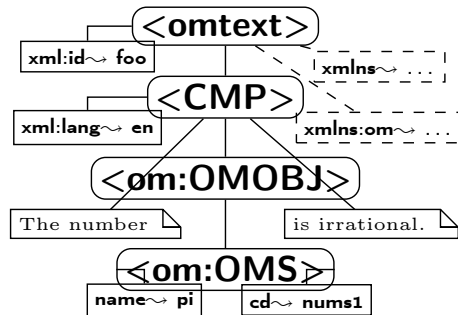


One of the great advantages of viewing marked-up documents as trees is that we can describe subsets of its nodes.

XPath, A Language for talking about XML Tree Fragments

▷ **Definition 3.3.12** The **XML path language** (XPath) is a language framework for specifying fragments of XML trees.

▷ **Example 3.3.13**



XPath exp.	fragment
/	root
omtext/CMP/*	all <CMP> children
//@name	the name attribute on the <OMS> element
//CMP/*[1]	the first child of all <OMS> 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.

Chapter 4

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 4.0.1** 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 4.0.2** 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 Kohlhasse

117



For that, we will first need to understand the basics of how the World-Wide Web works (see Section 4.1), how we can generate HTML documents programmatically (in our case in **python**; see Section 4.2), and finally how we can make HTML pages dynamic by client-side manipulation (see **?sec.clientside?**).

4.1 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.

4.1.1 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 4.1.1** The **Internet** is a worldwide computer network that connects hundreds of thousands of smaller networks. (The mother of all networks)
- ▷ **Definition 4.1.2** 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 4.1.3** A **web page** is a document (usually marked up in HTML) on the WWWeb that can include multimedia data and hyperlinks.
- ▷ **Definition 4.1.4** 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 4.1.5** 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 4.1.6** 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 **WWWeb** is a multimedia hypertext.



4.1.2 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 4.1.7** 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]. 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 4.1.8** 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).



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

▷ **Example 4.1.10** Relative URIs are more convenient to write

relative URI	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 4.1.11** To distinguish them from relative URIs, we call URIs **absolute URIs**.



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/kohlhasse/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 *names*”.

Uniform Resource Names and Locators

- ▷ **Definition 4.1.12** 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 4.1.13** The following URI is a URL (try it in your browser)
`http://kwarc.info/kohlhasse/index.html`
- ▷ **Example 4.1.14** `urn:isbn:978-3-540-37897-6` only identifies [Koh06] (it is in the library)
- ▷ **Example 4.1.15** 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 WWW**. (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 4.1.16** URIs are ASCII strings.
- ▷ **Problem:** This is awkward e.g. for France Télécom, worse in Asia.
- ▷ **Solution?:** Use [unicode](#) (no, too young/unsafe)
- ▷ **Definition 4.1.17** [Internationalized resource identifiers \(IRIs\)](#) extend the ASCII-based URIs to the [universal character set](#).
- ▷ **Definition 4.1.18** [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.

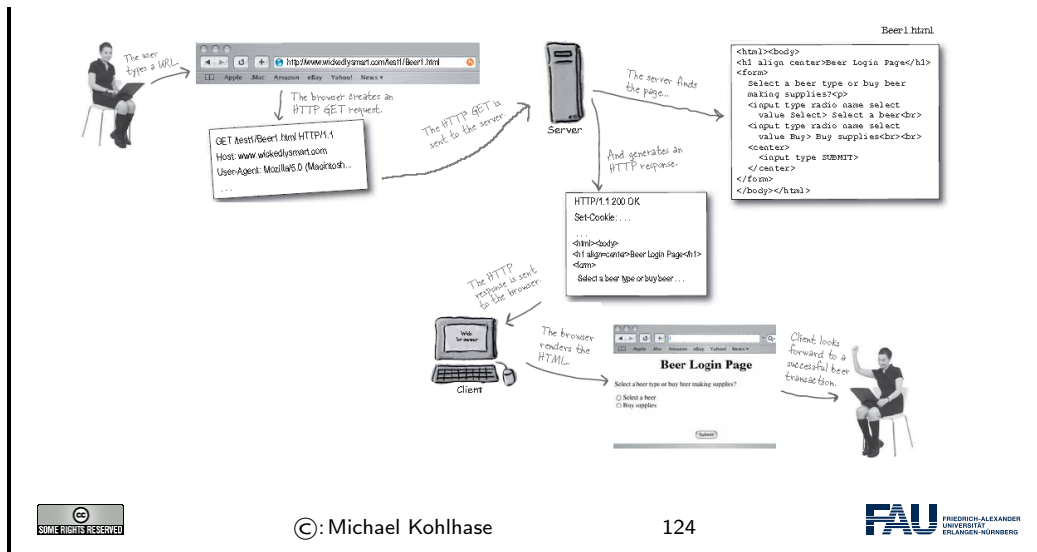
[URI decoding](#) is the dual operation.
- ▷ **Example 4.1.19** The letter “ı” (U+ 142) would be represented as %C5%82.
- ▷ **Example 4.1.20** <http://www.Übergrößen.de> becomes <http://www.%C3%9Cbergr%C3%B6%C3%9Fen.de>
- ▷ **Remark 4.1.21** Your browser can still show the URI-decoded version (so you can read it)



4.1.3 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



We will now go through and introduce the infrastructure components of the WWW in the order we encounter them. We start with the user agent; in our example the web browser used by the user to request the web page by entering its URL into the URL bar.

Web Browsers

- ▷ **Definition 4.1.22** 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 4.1.23 (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.

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 4.1.24** 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 4.1.25 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 4.1.26** Most important HTTP requests(5 more less prominent)

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

▷ **Definition 4.1.27** 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 4.1.28** 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.**)



©: Michael Kohlhasse

126



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 4.1.29** 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 4.1.30 (Common Web Servers)**

▷ **apache** is an open source web server that serves about 50% of the WWWeb.

▷ **nginx** is a lightweight open source web server (**ca. 35%**)

.

▷ **IIS** is a proprietary server provided by Microsoft.

▷ **Definition 4.1.31** A web server can **host** – i.e serve resources for – multiple domains that can be addressed in the authority components of URLs. This usually includes the special **hostname localhost** 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].



©: Michael Kohlhasse

127



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

```
wget https://kwarc.info
--2019-03-06 14:04:19-- https://kwarc.info/
Loaded CA certificate '/etc/ssl/certs/ca-certificates.crt'
Resolving kwarc.info (kwarc.info)... 131.188.48.212, 2001:638:a000:4148:131:188:48:212
Connecting to kwarc.info (kwarc.info)|131.188.48.212|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 8711 (8.5K) [text/html]
Saving to: 'index.html'

index.html 100%[=====] 8.51K --.-KB/s in 0s
2019-03-06 14:04:19 (30.0 MB/s) - 'index.html' saved [8711/8711]
```

- ▷ Looking at the response from the server

```
cat index.html
<!DOCTYPE html> <html lang="en-us"> ... <body> ... </body> </html>
```



©: Michael Kohlhasse

128



4.1.4 HTML Forms and the Web

The first requirement for web applications is already met by *html* in terms of HTML forms (see slide 91 ff.). Let us recap.

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 4.1.32** Forms contain input fields and explanations.

```
<form name="input" action="/user/" method="get">
  Username: <input type="text" name="user" />
  <input type="submit" value="Submit" />
</form>
```

yields

Username:

Pressing the submit button activates a HTTP GET request to the URL `/user/?user=⟨name⟩` (other methods (e.g. POST) also possible)

- ▷ 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 4.1.33 (Using GET vs. POST in HTML Forms)**

	GET	POST
Caching	possible	never
Browser History	Yes	never
Bookmarking	Yes	No
Change Server Data	No	Yes
Size Restrictions	$\leq 2KB$	No
Encryption	No	HTTPS

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.



4.2 Generating HTML on the Server

As the WWW 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 we can also program them! (easy to do)
- ▷ **Definition 4.2.1** A **server-side scripting framework** is a web server extension that generates web pages upon HTTP GET requests.
- ▷ **Example 4.2.2** perl is a scripting language with good string manipulation facilities. perl CGI is an early **server-side scripting framework** based on this.
- ▷ **Observation 4.2.3** 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 4.2.4** 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 **URIs** – the **routes**, we call this **routing**.



4.2.1 Templating in Python via STPL

In , we use python for programming, so let us see how we would generate HTML pages in python.

What would we do in python

▷ Example 4.2.5 (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 4.2.6 (Python Solution) use python functions:

```
def htmlpage (t,b):
    f"<html><head><title>{t}</title></head><body>{t}</body></html>"
    htmlpage("Hello","Hello IWGS")
```

Problem 2: If HTML markup dominate, 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)

▷ If HTML markup dominates, want to use a HTML editor (mode)

▷ ▷ e.g. for HTML syntax highlighting/indentation/completion/checking



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 4.2.7** 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 4.2.8** 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 4.2.9** PHP (originally “Programmable Home Page Tools”) is a very successful server-side scripting framework following this model.



Before we start with a templating engine in python, we will present extended function/argument patterns².

EdN:2

Argument Passing in python: Default Arguments

▷ **Definition 4.2.10** The last $k \leq n$ of n parameters of a **function** can be **default 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 4.2.11** The head of the open function is

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



Argument Passing in python: Flexible Arity

▷ **Definition 4.2.12** python **functions** can take a variable number of parameters: $\text{def } f(p_1, \dots, p_k, *r)$ allows $f(a_1, \dots, a_k, a_{k+1}, \dots, a_n)$ and binds the parameter r to the list $[a_{k+1}, \dots, a_n]$.

▷ **Example 4.2.13**

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

▷ We can also use the "star syntax" in the function call

▷ **Example 4.2.14 (Passing a list)**

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



Argument Passing in python: Keyword Arguments

▷ **Definition 4.2.15** python **functions** can take **keyword arguments**: if k is a sequence of key/value pairs then $\text{def } f(p_1, \dots, p_n, **k)$, binds the keys to values in the body of f .

▷ **Example 4.2.16**

```
def kw_args(farg, **kwargs):
```

²EdNOTE: remove this in the next year; it is already in the python intro

```

    print "formal arg:", farg
    for key in kwargs:
        print "another keyword arg: %s: %s" % (key, kwargs[key])
>>> kw_args(farg=1, myarg2="two", myarg3=3)

```

▷ Again, we can use the “double star syntax” in the function call

▷ **Example 4.2.17 (Passing a dictionary)**

```

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

```



©: Michael Kohlhase

136



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 4.2.18** **Bottle WSGI** is a server-side scripting framework based on python. It supplies the template engine stpl (Simple Template Engine). ([documentation at \[STPL\]](#))

▷ Bottle WSGI comes pre-installed on pythonAnywhere, so we only have to import stpl to use it.

```

from bottle import template

```

stpl uses the template function for template processing and `{{...}}` to embed into a template string.

```

>>> template('Hello {{name}}!', name='World')
u'Hello World!'

```

template accepts the same argument pattern as `format`³ and returns **unicode strings**. E.g.

```

>>> 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

137



³EdNOTE: need to introduce that above

This 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 4.2.19** 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 4.2.20** 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 4.2.21** 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 4.2.22 (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 4.2.23 (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

```




The basic pattern is that we separate concerns and build a separate template file for any “design feature” of the respective web site. Sometimes these have parameters, which we can test via the defined predicate and whose values we can obtain via `get`.

Using Templates

- ▷ We can use the base name of the template file in the template function to process it.

▷ Example 4.2.24 (Extended Example)

- ▷ A python file `books.py` that provides the data and calls the template function.

```
from bottle import template
books = [{ 'author': 'Tolkien', 'year': 1937, 'title': 'The Hobbit' },
          { 'author': 'Twain', 'year': 1876, 'title': 'Tom Sawyer' },
          { 'author': 'Hemmingway', 'year': 1940, 'title': 'For Whom the Bell Tolls' } ]
template('books', booklist=books)
```

- ▷ A `stpl` template `books.tpl` for a books table

```
<table>
  <tr><th>author</th><th>year</th><th>title</th></tr>
  % for book in books include('book.tpl', **booklist)
</table>
```

- ▷ An auxiliary `stpl` template `book.tpl` for row

```
<tr><td>{{author}}</td><td>{{year}}</td><td>{{title}}</td></tr>
```



4.2.2 Routing, and Argument Passing in Bottle

Routing in Bottle WSGI

- ▷ **Definition 4.2.25** **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**.
- ▷ **Bottle WSGI** 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 4.2.26 (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)

```



Dynamic and Method-specific Routes in Bottle

- ▷ But we can do more with routes
- ▷ **Definition 4.2.27** A **dynamic route** is a route annotation that contains a **named wildcard**, which can be picked up in the route function.
- ▷ **Example 4.2.28** 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 template('Hello {{name}}, how are you?', name=name)

```

With the wildcard `<name>` we can bind the route function `greet` to all paths and via its argument `greet` customize the greeting.

- ▷ **Definition 4.2.29** Dynamic routes can be restricted by a **route filter** to make them more selective.
- ▷ **Example 4.2.30 (Concrete Filters)** `:int` for integers or `:re:<regex>` for **regular expressions**

```

@route('/object/<id:int>')
@route('/show/<name:re:[a-z]+>')

```



Dealing with HTTP GET and POST Data

- ▷ **Recall:** from a HTML form we get a GET and POST request with query $?n_1=v_1 \& \dots \& n_k=v_k$
- ▷ Bottle WSGI provides the request object for dealing with HTTP request data.
- ▷ **Example 4.2.31 (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: `smtplib`, which makes this relatively easy. (SMTP $\hat{=}$ "Simple Mail Transfer Protocol")
- ▷ **Example 4.2.32 (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")



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 4.2.33 (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 4.2.34** 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 4.2.35** **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 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.

4.3 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:

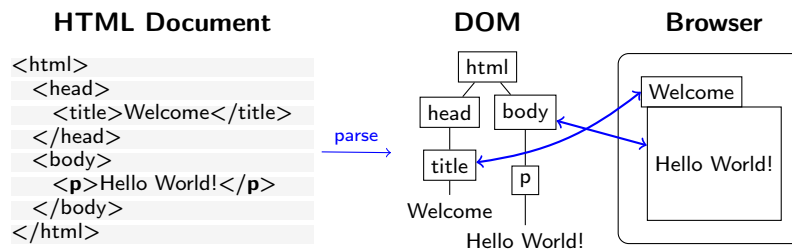
- i) a machine-accessible representation of the document structure, and
- ii) a program interpreter in the web browser, and
- iii) a way to send programs to the browser together with the documents.

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 4.3.1** 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)



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 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.

4.3.1 JavaScript in HTML

Dynamic HTML

- ▷ **Idea:** generate parts of the web page dynamically by manipulating the DOM.
- ▷ **Definition 4.3.2** 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 4.3.3** 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>
```



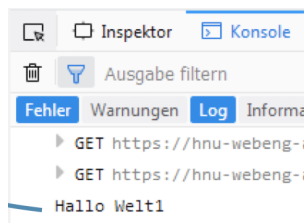
The example above already shows a JavaScript command: `document.write`, which replaces the content of the `<body>` element with its argument – this is only useful for testing and debugging purposes.

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 4.3.4 (Logging to the browser console)

```
console.log("hello IWGS")
```



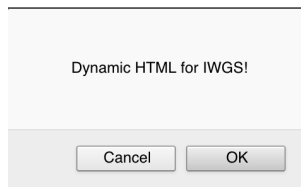
▷ Example 4.3.5 (Raising a Popup)

```
alert("Dynamic HTML for IWGS!")
```



▷ Example 4.3.6 (Asking for Confirmation)

```
var returnValue = confirm("Dynamic HTML for IWGS!")
```



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

149



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

150



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 handle an event: some elements have default event handlers, e.g. an HTML anchor `` will handle a click event by issuing a

HTTP GET request for `⟨URI⟩`. 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 `` 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 4.3.7 (Stupid but Fun)

```
<body>
<h2>A Pyramid</h2>
<div id="pyramid"/>

<script type="test/javascript">
  var char = "#";
  var triangle = "";
  var str = "";
  for(var i=0;i<=10;i++){
    str = str + char;
    triangle = triangle + str + "<br/>"
  }
  var elem = document.getElementById("pyramid");
  elem.innerHTML=triangle;
</script>
</body>
</html>
```

Eine Pyramide

```
#
##
###
####
#####
#####
#####
#####
#####
#####
```



The HTML document in Example 4.3.7 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 `
` 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 4.3.7. It is the same as if the `#`-strings and `
` sequence had been written in the HTML – which – at least for pyramids of greater depth – would have been quite tedious for the author.

4.3.2 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 4.3.8** 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-3.1.1.min.js"/>
```

or from the Internet directly (only works if you are online)

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js" />
```



©: Michael Kohlhasse

152



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



©: Michael Kohlhasse

153



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 4.3.9** 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 5

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
- ▷ Web technologies for interactive documents and applications
 - ▷ Internet infrastructure: web browsers and servers
 - ▷ PHP, dynamic HTML, Javascript, HTML forms
- ▷ Web Application Project (design your own!)



©: Michael Kohlhase

156



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



Part II

IWGS-II: DH Project Tools

Chapter 6

Semester Change-Over

6.1 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 Kohlhasse

158



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 ~ check understanding/preparation)



©: Michael Kohlhase

159



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

160



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.

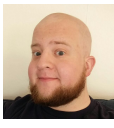

IWGS Tutorials

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



Teaching Assistants: (Doctoral Students in CS)

- ▷ Jonas Betzendahl: jonas.betzendahl@fau.de
- ▷ Philipp Kurth: 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 Kohlhasse 161 

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 them as we go along (first time I teach IWGS)
 - ▷ 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
- ▷ 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 Kohlhasse 162 

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



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

164



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 7.

Chapter 7

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 7.1)
- issue tracking and management via GitHub/GitLab (Section 7.3).

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.

7.1 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.

7.1.1 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 7.1.1** *Artefact collections can become large and long-lived.*
- ▷ **Problem:** How to manage them effectively?

- ▷ **Example 7.1.2** 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 (≥ 5 M articles, ≥ 280 languages, ca. 40M files, ≥ 130 k active users, since 2001).
 - ▷ “2048”: a simple browser/app game with lots and lots of variants (forks) in three years.



©: Michael Kohlhasse

165



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 7.1.3** 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 7.1.4** For technical books, multiple editions follow to adapt them to changing domain or correct errors.



©: Michael Kohlhasse

166



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.

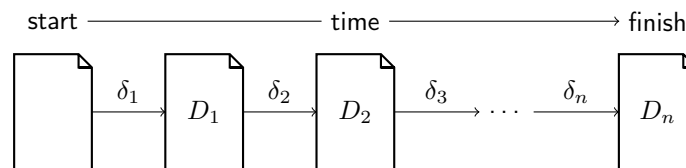
Document Lifecycle Mgmt. & Collaboration Approaches

▷ **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 multiframe support, no snapshots

▷ **Summary:** only supports serial collaboration, no multiframe support



larger teams \leadsto more time wasted



©: Michael Kohlhasse

167



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

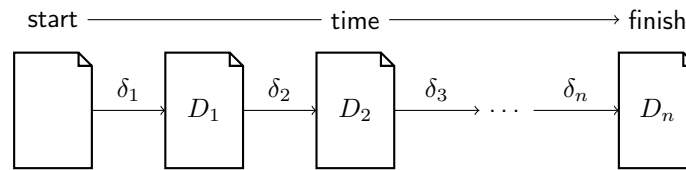
▷ **Practice:** Put your documents on Dropbox or MS Sharepoint, or use a Wiki.

▷ **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



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.

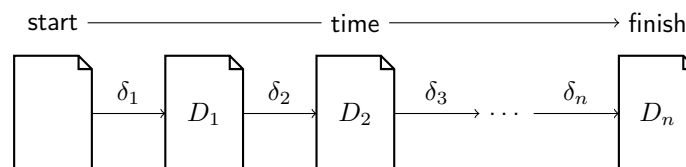
Document Lifecycle Mgmt. & Collaboration Approaches

▷ **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



larger teams \leadsto more time wasted



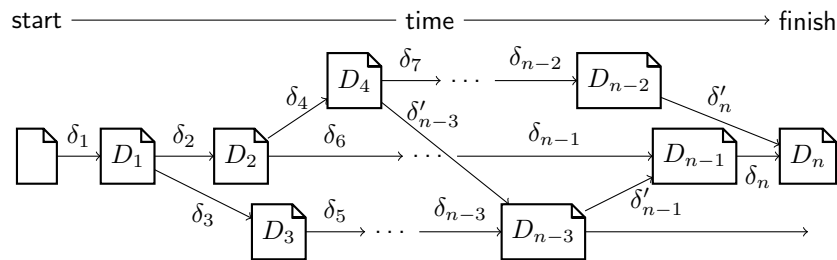
While document consistency is directly guaranteed by the system, intra document, semantic consistency is very hard to achieve, as there is usually no possibility to block out other authors in order to do a larger rewrite. Though the systems give access to the version history, it's character-by-character 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



larger teams \leadsto large-scale parallelization/experimentation



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 δ 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.

7.1.2 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 7.1.5** 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 7.1.6** 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 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 7.1.7** *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 7.1.5 and Definition 7.1.6 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 7.1.8** **diff** is a file comparison utility that computes differences between two **text files** f_1 and f_2 . Differences are output linewise in a **diff file** (also called a **patch**), which can be applied to f_1 to obtain f_2 via the **patch** utility.
- ▷ **Example 7.1.9**

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 7.1.10** A diff file consists of a sequence of **hunks** that in turn consist of a locator which contrasts the source and target locations (in terms of line numbers) followed by the added/deleted lines.



Merging Differences with merge3

- ▷ There are basically two ways of **merge** the differences of files into one.
- ▷ **Definition 7.1.11** In **two-way merge**, an automated procedure tries to combine two different files by copying over differences by guessing or asking the user.
- ▷ **Definition 7.1.12** In **three-way merge** the files are assumed to be created by changing a joint original (the **parent**) by editing. The **merge3** tool examines the differences and patterns appearing in the changes between both files as well as the parent, building a relationship model to generate a new revision. Usually, non-conflicting differences (affecting only one of the files) can directly be copied over.



With this, we can now understand the revision control workflows in our concrete system.

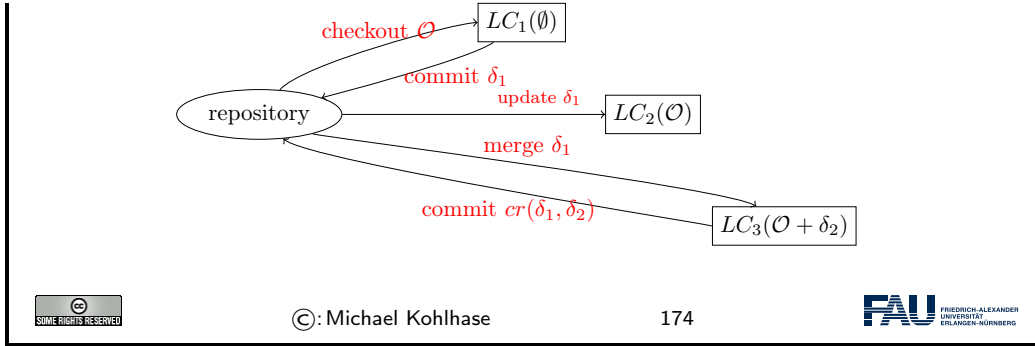
In its simplest form, a revision control system, can be understood using the Subversion system that is heavily used in open source projects that have a relatively hierarchical development model.

Centralized Version Control (with Subversion)

- ▷ **Definition 7.1.13** 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 **diff** differences and patching the repository and working copies. Conflicts are resolved by (three-way) **merge**.

- ▷ **Example 7.1.14 (A Workflow with three Working Copies)**



In the workflow of Example 7.1.14 is a typical one:

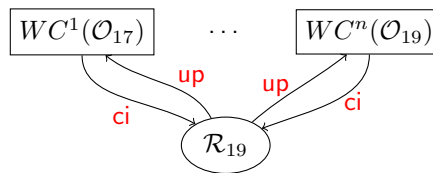
1. A first user checks out a new working copy LC_1 , from the empty repository, adds a couple of files – we denote the new document collection at this point with \mathcal{O} , and commits the difference δ_1 between the working copy and \mathcal{O} to the repository which δ_1 logs it as “revision 1”.
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 δ_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 δ_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 δ_1 and δ_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 with Subversion

- ▷ **Idea:** We can use the same technique for collaboration between multiple working copies.
- ▷ **Diff-Based Collaboration:**



The Subversion system takes care of the synchronization:

- ▷ you can only commit, if your revision is HEAD (otherwise update)
- ▷ update merges the changes into your working copy

- ▷ If there are changes on the same line, you have a conflict.

```

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

```



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 changes¹. 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 7.1.15** *A central repository entails – ultimately – a single line of development. ↔ changes have to be merged into the repository eventually.*
- ▷ **But:** we want to develop – and commit – to variants in parallel.
- ▷ **Definition 7.1.16** 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 7.1.17** 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 7.1.18** 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.

¹Don't drink and write!

7.1.3 Distributed Revision Control

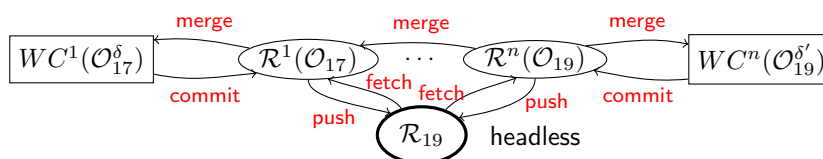
In this Subsection we will introduce distributed revision control systems using the GIT system as an example. As this is the currently dominant system, we will also go into more detail about concrete usage of the system.

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 7.1.19** 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 7.1.20** We call a repository **headless** (or **bare**), if used without working copy, usually in a web server.



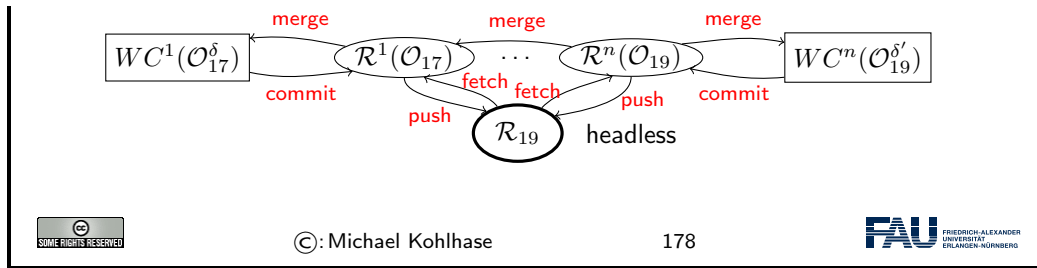
The concept of distributed revision control systems is motivated by the two shortcomings at the top of the slide, which can be remedies by a single – if relatively radical idea: allowing lots of repositories that can communicate with each other by exchanging **patches**. Local repositories allow commits 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 7.1.21** 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 **fetch**ed from and **push**ed to a remote repository (the **upstream**/**downstream** repositories).
- ▷ 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 repositories**)
- ▷ GitLab, an open source **repository management system** and **repository hosting service** at <http://GitLab.com> (**free public/private repositories**)



©: Michael Kohlhasse

179



7.1.4 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`
- ▷ Make 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': yp70uzyj
Password for 'https://yp70uzyj@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)

- ▷ A repository can be connected to one or several [remote repositories](#)

- ▷ `git remote -v` shows the [remote repositories](#) e.g.

```
MiKo:collaboration kohlhasse$ 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.

```
kohlhasse$ git remote add upstream git@gl.kwarc.info:test/collab.git
kohlhasse$ 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`



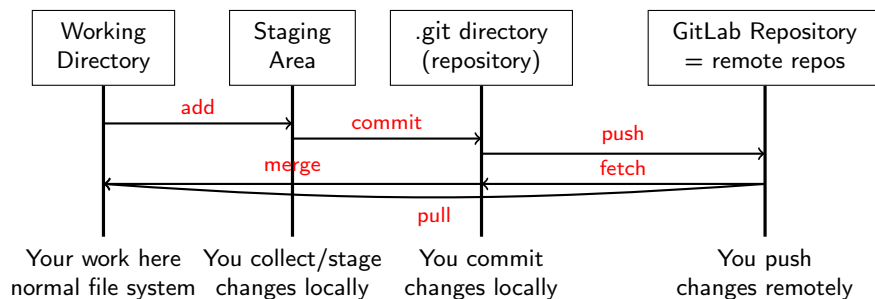
We will now come to a GIT peculiarity that is important to understand for working with GIT: Often we only want to **commit** only a subset of the changed files – e.g. because the changes already constitute a achievement of their own or we want to split the development into multiple commits. There are 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**



commits act only on staged files \leadsto `git add foo.tex`

- ▷ basic GIT commands (there are many variants and options \leadsto 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.

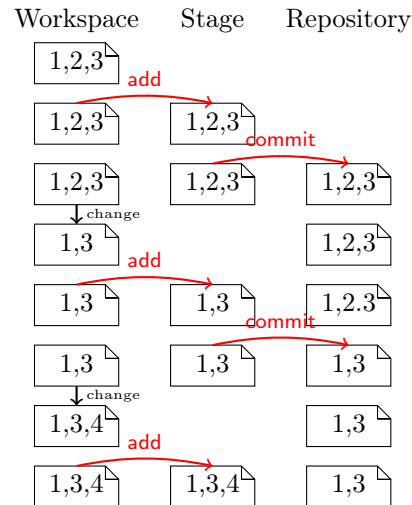
An Example Git Workflow

▷ **Example 7.1.22** 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.

7.1.5 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: branch 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, Remote Repositories

▷ GIT special commands for making, switching, and merging branches.

<code>git branch <branch></code>	makes a branch with name <name>
<code>git checkout <branch></code>	switches a working copy to branch <branch>
<code>git branch -v</code>	shows all branches
<code>git branch -d <branch></code>	deletes branch <branch>

Intuition: In GIT branches are very similar to repositories, but more lightweight. Repositories can have different permissions.

▷ **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.

▷ Git commands for working with remote repositories:

git remote add $\langle\langle\text{name}\rangle\rangle$ $\langle\langle\text{URI}\rangle\rangle$	gives the repos at $\langle\langle\text{URI}\rangle\rangle$ the name $\langle\langle\text{name}\rangle\rangle$
git remote show	shows all remote repositories

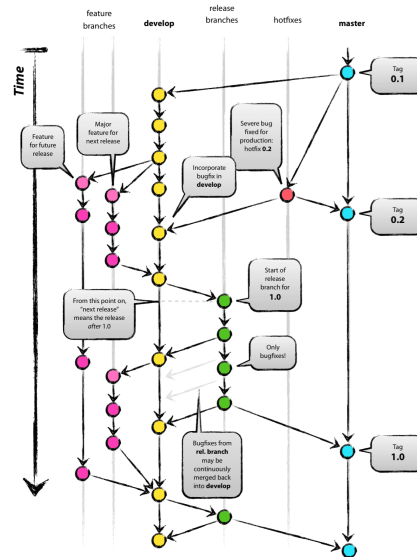


What we have seen above, let us briefly discuss an elaborate workflow suitable for large development teams, which has become known under the name “GitFlow”.

GitFlow: An Elaborate Development Model based on GIT

▷ **Definition 7.1.23** [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** 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.



7.2 Working with GIT and GitLab/GitHub

Working with GitLab/GitHub


▷ GIT it sufficient to set up a **remote repository** (but tedious and error-prone)

- ▷ **Idea:** Use a GIT **repository manager** like GitLab/GitHub (we use GitLab)
- ▷ **Definition 7.2.1** A **repository management system** is an information system 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 7.2.2** GitLab is an open source repository management system and **repository hosting service** at <http://GitLab.com> (free public/private repositories)
- ▷ **Definition 7.2.3** A **repository hosting service** is a web-based repository management system that also offers storage space for repositories.
- ▷ **Example 7.2.4** 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.
- ▷ **Definition 7.2.5** Often, repository management systems organize repositories 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**
 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 projects go 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 7.2.6** 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)



Using GitLab for the IWGS Project

- ▷ Make a in a member



©: Michael Kohlhasse

188



7.2.1 Excursion: Authentication with SSH

Authentication

- ▷ **Definition 7.2.7 Authentication** is the process of ascertaining that somebody really is who they claim to be.
- ▷ **Definition 7.2.8** 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.



©: Michael Kohlhasse

189



Authentication by Cryptographic Public Keys

- ▷ **Definition 7.2.9 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 .
- ▷ Split key into **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.



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.

7.3 Bug/Issue Tracking Systems

In this Section we will discuss **issue tracking systems**, which support the collaborative management of reports on a particular problem, 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. We will mainly look at systems that originate from software engineering applications here.

Bug/Issue Tracking Systems

- ▷ **Definition 7.3.1** An **issue tracker** (also called **issue tracking system** simply **bugtracker**) is a software application that keeps track of reported **issues** – i.e. software bugs and feature requests – in software development projects.
- ▷ **Example 7.3.2** 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>
 - ▷ GitLab: <http://gitlab.com> (open source version of GitHub)
 - ▷ JIRA: <https://www.atlassian.com/software/jira> (proprietary)

Most bugtrackers also integrate a **wiki** and integrate a revision control system via extended **markdown**.



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 7.3.3** 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)
- ▷ **discussion** about the bug.
- ▷ **attachment**: e.g. a screen shot, set of inputs, etc.



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 7.3.4 (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 7.3.5 (A good one)**

I crash each time I go to foo.com (Mozilla build 20000609, Win NT 4.0SP5). This link will crash Mozilla 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)

▷ **Definition 7.3.6** A **feature request** is an issue that only specifies the expected behavior and proposes ways of implementing that.



Markdown a simple Markup Language Generating HTML.

▷ **Idea**: We can translate between markup languages.

▷ **Definition 7.3.7** **Markdown** is a family of markup languages 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.

- ▷ **Example 7.3.8** 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.
- ▷ **Example 7.3.9** We show the most important Markdown commands.

Markdown syntax	Generated HTML
# Heading	<h1>Heading</h1>
## Sub-heading	<h2>Sub-heading</h2>
### Another deeper heading	<h3>Another deeper heading</h3>
Paragraphs are separated by a blank line.	<p>Paragraphs are separated by a blank line.</p>
Two spaces at the end of a line leave a line break.	<p>Two spaces at the end of a line leave a line break.</p>
Text attributes <i>italic</i> , bold , <code>monospace</code> .	<p>Text attributes italic, bold, <code>monospace</code>.</p>
Bullet list:	<p>Bullet list:</p>
* apples	
* oranges	apples
* pears	oranges
	pears
	
Numbered list:	<p>Numbered list:</p>
1. apples	
2. oranges	apples
3. pears	oranges
	pears
	
A [link](http://example.com) .	<p>A link.</p>



Tracker-Specific Markdown Extensions

- ▷ **Remark 7.3.10** Source code hosting systems offer special extensions for referencing their components.
- ▷ **Example 7.3.11** GitLab recognizes
 - ▷ @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 4c0decbb yesterday.*
 - ▷ [file](path/to/file) for file references, e.g. *as we see in [pre.tex](../lib/pre.tex)*
- ▷ **Observation 7.3.12** *very useful for project planning and reporting*



Bugtracker Workflow

- ▷ **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, . . .
- ▷ developer accepts 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)
- ▷ bug closure
- ▷ **Observation 7.3.13** An *issue tracker* can serve as a full-blown project planning system, if used accordingly.
- ▷ **Definition 7.3.14** For timing work plans, most *issue trackers* provide *milestones* that issues can be targetted to.



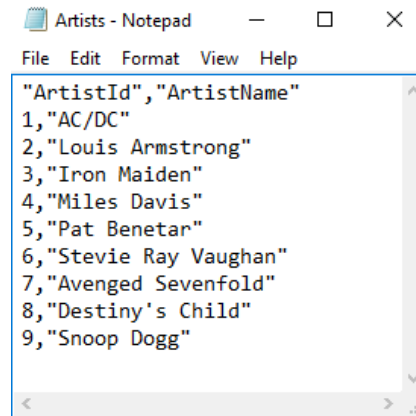
Administrative Metadata for Issues

- ▷ to make the issue-based workflows work we need data
- ▷ **Definition 7.3.15 (Administrative Metadata)** issue metadata can specify
 - ▷ **issue number**: for referencing with e.g. #15
 - ▷ an **assignee**: a developer currently responsible
 - ▷ **comments**: a discussion thread focused on this issue.
 - ▷ **participants**: people who get notified of changes/comments
 - ▷ **labels**: for specializing bug search
 - ▷ a **status**: 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?

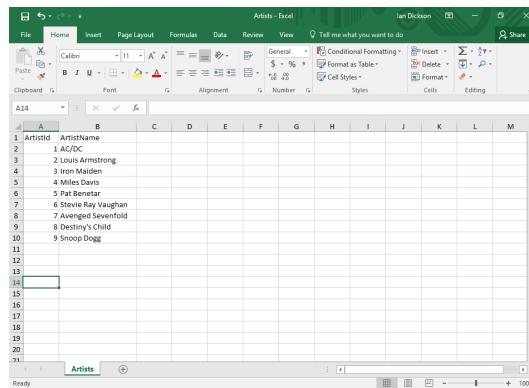


not persistent (-))

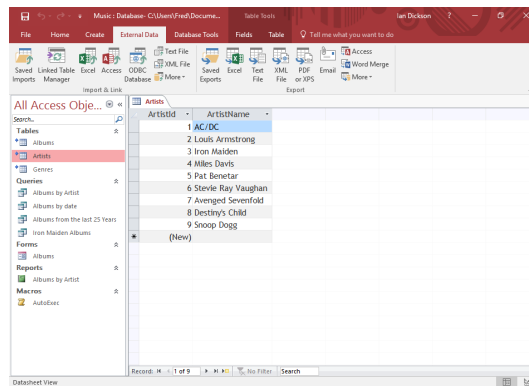
- ▷ In a text file (persistent (+), fast (+), sequential access (-), unstructured (-))



- ▷ In a spreadsheet (persistent (+), 2D-structured (+-), relations (+), slow (-))



- ▷ In a database (persistent (+), scalable (+), relations(+), managed (+), slow (-))



- ▷ Databases constitute the most scalable, persistent solution.

8.2 Relational Databases

We will now study a particular kind of database: [relational databases](#), as these are the most widely used and structured ones.⁴

EdN:4

(Relational) Database Management Systems

- ▷ **Definition 8.2.1** 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 8.2.2** 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 8.2.3 (Bibliographic Data)**

Last	First	YOB	YOD	Title	YOP	Publisher	City
Twain	Mark	1835	1910	Huckleberry	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 8.2.4** Tables are identified by **table name** and individual components of **records** by **column name**.



©: Michael Kohlhasse

201



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.

Open-Source Relational Database Management Systems

- ▷ **Definition 8.2.5** 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 8.2.6** PostgreSQL is an open source RDBMS with an emphasis on extensibility, standards compliance, and scalability.



⁴EdNOTE: MK: In the last years, NoSQL databases and JSON have gained prominence. Intro them at the end and reference them here.

- ▷ **Definition 8.2.7** SQLite is an embeddable RDBMS. Instead of a database server, SQLite uses a single database file, therefore no server configuration is necessary.
- ▷ **Remark 8.2.8** 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 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, test

```
> 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 8.3.8, 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|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
```



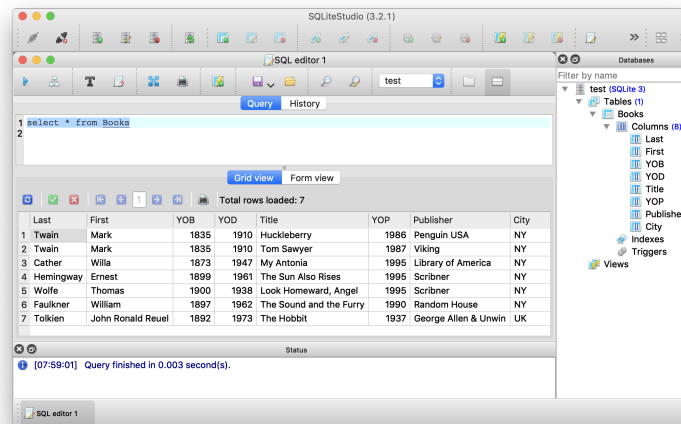
EdN:5

Interacting with SQLite via the database shell⁵ is nice, but can be quite tedious. Fortunately, there are better alternatives.

A Graphical User Interface for SQLite

⁵EdNOTE: MK: maybe introduce that separately somewhere?

- ▷ **Definition 8.2.9** 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)



8.3 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 8.3.1** **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.



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 access.

DDL: Data Definition Language

▷ **Definition 8.3.2** The [data definition language \(DDL\)](#) is a subset of SQL instructions that address the creation and deletion of database objects.

▷ **Definition 8.3.3** 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 8.3.4 (Creating a Table)** The following SQL statement creates the table from Example 8.2.3

```
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 8.3.5** The SQL statement **DROP** ⟨⟨obj⟩⟩ ⟨⟨name⟩⟩ deletes the database object of class ⟨⟨obj⟩⟩ with name ⟨⟨name⟩⟩.



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 8.3.6** SQL specifies data types for [values](#) including

- ▷ **VARCHAR** (⟨⟨length⟩⟩): character strings, including Unicode, of a variable length is up to the maximum length of ⟨⟨length⟩⟩.
- ▷ **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** (⟨⟨length⟩⟩) (character large object) up to (typically) 2 Gi B
- ▷ **BLOB** (⟨⟨length⟩⟩) (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 8.3.7** SQL provides the **INSERT INTO** command for inserting records into a table. This comes in two forms:

1. **INSERT INTO** *⟨table⟩* **VALUES** (*⟨vals⟩*); where *⟨vals⟩* is a comma-separated list of values given in the order the columns were declared in the **CREATE TABLE** instruction.
2. **INSERT INTO** *⟨table⟩* (*⟨cols⟩*) **VALUES** (*⟨vals⟩*) where *⟨vals⟩* is a comma-separated list of values given in the order of *⟨cols⟩* (a subset of columns) all other fields are filled with **NULL**

▷ **Example 8.3.8 (Inserting into the Books Table)** The given the table Books from Example 8.3.4 we can add a record with

```
INSERT INTO Books
VALUES ('Tolkien', 'John Ronald Reuel', 1892, 1973, 'The Hobbit', 1937,
      'George Allen & Unwin', 'UK');
```

▷ **Example 8.3.9 (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', 'Kohlhasse', '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 8.3.10** The SQL **delete** statement allows to change existing records.

```
DELETE FROM ⟨table⟩ WHERE ⟨condition⟩;
```

▷ **Example 8.3.11** 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 8.7)



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 8.3.12** The SQL **update** statement allows to change existing records.

```
UPDATE <<table>>
SET <<column>>1 = <<value>>1, <<column>>2 = <<value>>2, ...
WHERE <<condition>>;
```

- ▷ **Example 8.3.13** Updating the publisher in “Huckleberry Finn”.

```
UPDATE Books
SET Publisher = 'Chatto & Windus', YOP = 1884, Cit = 'London'
WHERE Title = 'Huckleberry Finn'
```

- ⚠ **Again:** If we leave out the **WHERE** clause, all rows are updated.



8.4 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 8.2.3:

Last	First	YOB	YOD	Title	YOP	Publisher	City
Twain	Mark	1835	1910	Huckleberry	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 relation-**

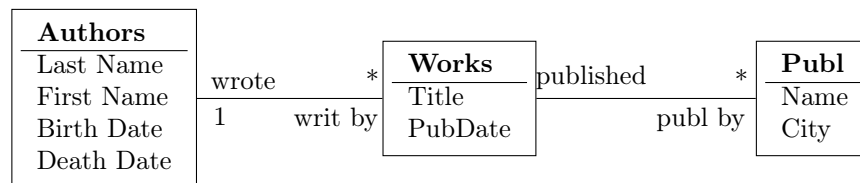
ship diagrams themselves.

Entity Relationship Diagrams

- ▷ **Definition 8.4.1** 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 8.4.2 (An ERD for Books)** Recall the Books table from Example 8.2.3:

Last	First	YOB	YOD	Title	YOP	Publisher	City
Twain	Mark	1835	1910	Huckleberry	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.



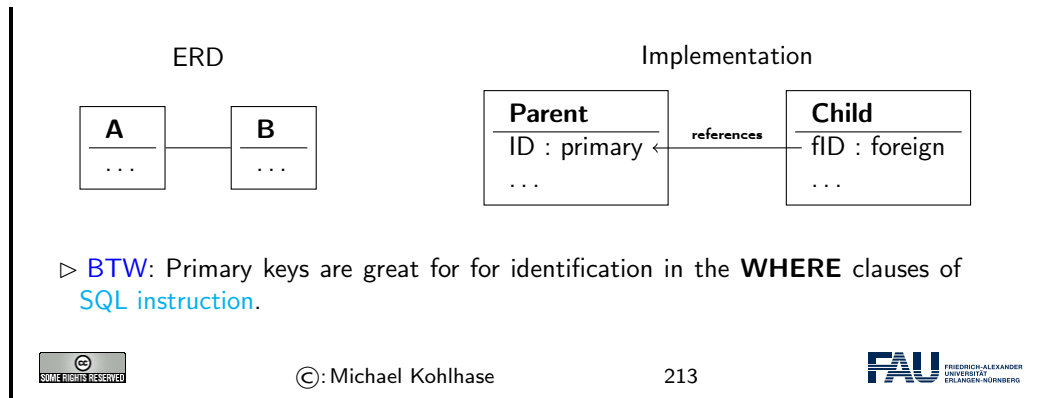
Generally, a good database design is almost always worth the effort, since it makes the code and maintenance of the applications based on this database much simpler and intuitive.

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

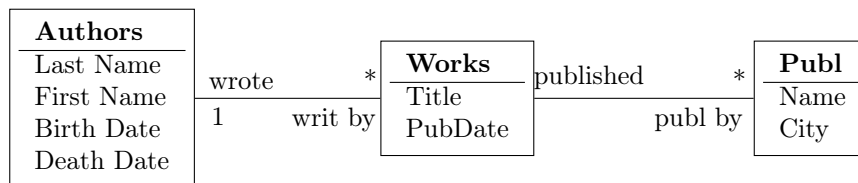
- ▷ **Definition 8.4.3** A column in a table can be designated as a **primary key**. This constrains its values to be non-null and **unique** i.e. all distinct. In DDL, we just add the keyword **PRIMARY KEY** to the column specification.
- ▷ **Definition 8.4.4** A **foreign key** is a column (or collection of **columnss**) in one table (the called **child table**) that refers to the primary key in another table (called the **reference table** or **parent table**).
- ▷ **Intuition:** Together primary keys and **foreign keyss** can be used to link tables or (dually) to spread information over multiple tables.



We now fortify our intuition on primary and foreign keys by taking up Example 8.4.2 again.

Linking Tables via Primary and Foreign Keys (Example)

▷ **Example 8.4.5** Continuing Example 8.4.2, 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)
);
```

▷ **Example 8.4.6 (Inserting into the Works Table)** The given the tables Works Authors, and Publishers from Example 8.4.5 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);
```



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.

8.5 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 = sqlite.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⟩⟩
# 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 `sqlite.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 `sqlite.connect` function is a database object `db`.
2. Then we create a [cursor](#) object `cursor` (cf. slide 224 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. 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 8.3.8 as a basis.

Creating Tables in python

▷ **Example 8.5.1** Creating the table of Example 8.3.4

```
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.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 217 – 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.

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 226 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 8.5.2** 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(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;
```



©: Michael Kohlhasse

217



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 **SQL instruction**, but with the parameter substitution in the f-string gets two. This would have raised an error and saved the school administration.

8.6 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 8.6.1 (Kohlhasse's Law)** *I can be an idiot, and I do make mistakes!*

▷ **Definition 8.6.2** An **exception** is a special python object. **Raising** an exception e terminates computation and passes e to the next higher level.

▷ **Example 8.6.3 (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 8.6.4** Things can go wrong when connecting to a database(e.g. *missing file*)
- ▷ **Idea:** Raise exceptions and **handle** them.
- ▷ **Example 8.6.5** 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.



8.7 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 8.7.1** We call a **SELECT** instruction a **query**.
- ▷ **Example 8.7.2** **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 8.7.3** **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 8.7.4** Ordering can be ascending (**ASC**) or descending (**DESC**)
SELECT First, Last **FROM** Books **ORDER** Last **ASC**, YOP **DESC**;

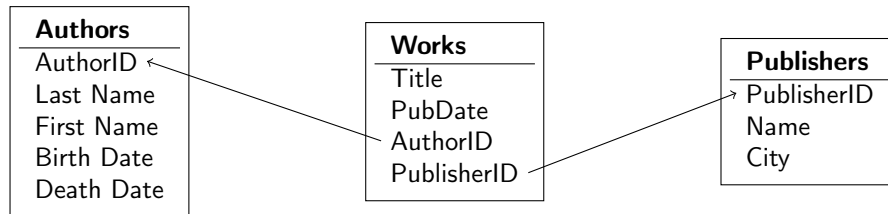


There are some more variants, for instance we can add a **GROUP BY** clause, which allows to group the result table according to various conditions.

We now generalize SQL queries by combining multiple tables into a virtual table from which we aggregate the results.

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 8.7.5** 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 8.7.6** Joining all three tables from Example 8.4.2.

```

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
  
```

SQLiteStudio (3.2.1) - SQL editor 1

Query History

```

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

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 8.7.6, the Works table queries like the original Books table.
- ▷ **Wouldn't it be nice** If we could also insert/update into that?
- ▷ **Definition 8.7.7** 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 8.7.8** we can 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)



©: Michael Kohlhase

222



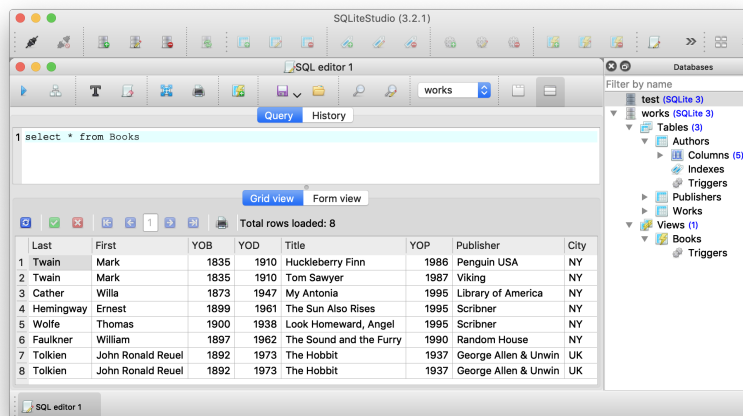
Remark: With views we can “have our cake and eat it too”: We can make our [database schema](#) space-efficient by removing redundancies using “small tables” and still have our “big tables” that make our life convenient e.g. when inserting records. Consider our Books example.

Database Views: Persisting Queries (Books Example)

- ▷ **Example 8.7.9** Use the query from Example 8.7.6 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.



©: Michael Kohlhase

223



8.8 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 8.8.1** 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

224



EdN:6

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.⁶

▷ Extended Example: Listing Authors from the Books Table

▷ Example 8.8.2

```
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

225



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.

SQLite3 Parameter Substitution

- ▷ **Observation 8.8.3** *We often need variables as parameters in `cursor.execute`.*
- ▷ **Example 8.8.4** In Example 8.8.2 we can ask the user for a year.

⁶EdNOTE: MK: show the results

- ▷ 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 8.8.5** 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 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 8.8.6** So the final form of insertion in Example 8.5.1 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
```



8.9 Project: A Web GUI for a Books Database

We now 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.

The Books Application: Setup

- ▷ We have already seen how to set up the database in slide 227.
- ▷ But we want to receive result rows as dictionaries, not as tuples, so we add


```
db.row_factory = sqlite3.Row
```
- ▷ And of course, start the server, and closed the database in the end


```
run(host='localhost', port=8080, debug=True)
db.close()
```
- ▷ We only need to add the Bottle routes for the various sub-pages.



©: Michael Kohlhasse

228



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():
    s = 'SELECT rowid,Last,First,YOB,YOD,Title,YOP,Publisher,City FROM Books'
    cursor.execute(s)
    booklist = cursor.fetchall()
```

- ▷ 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">add a book</a></th></tr>
</table>
```

It inserts the table header from the template file th.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>
</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}}">edit</a></td>
    <td><a href="/drop/{{rowid}}">delete</a></td>
</tr>
```



The Books Application Routes: Adding Book Records

- ▷ We add a route for adding books record (for the add button)

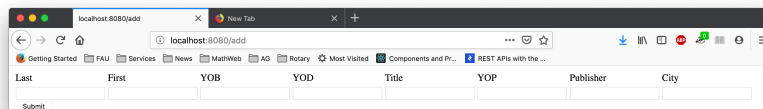
```
@get('/add')
def add():
    return template('add')
```

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 result is



- ▷ Here the action 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')
```

- ▷ 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'),
```

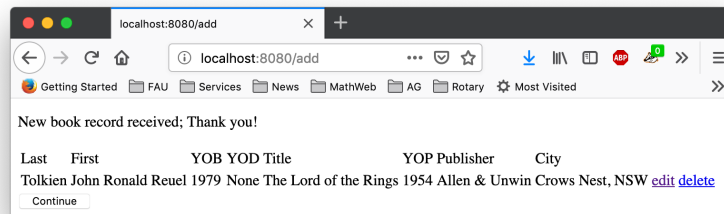


```
'Publisher': request.forms.get('Publisher'),
'City': request.forms.get('City')}]
return data
```

▷ and the template repsonse.tpl:

```
<form action="/">
  <p>{{text}}; Thank you!</p>
  <table>
    % include('th.tpl')
    % include('book.tpl',**data)
  </table>
  <input type="submit" value="Continue"/>
</form>
```

▷ Here is the result after filling in Tolkien's "*Lord of the Rings*":



The Books Application Routes: Deleting Book Records

▷ We add a route for deleting book records (for the green button)

```
@get('/delete/<id>')
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>
```



The Books Application Routes: Editing Book Records

▷ 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:

```
@get('/edit/<id:int>')
def edit(id):
    cursor.execute('SELECT * FROM Books WHERE rowid = ?',(id,))
    return template('edit',cursor.fetchone(), id = id)

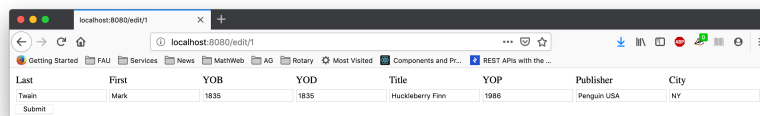
@post('/edit/<id:int>')
def editResponse(id):
    data = parseResponse()
    up = """UPDATE Books
        SET Last = :Last, First = :First, YOB = :YOB, YOD = :YOD,
            Title = :Title, YOP = :YOP,
            Publisher = :Publisher, City = :City
        WHERE rowid = :rowid"""
    dr = data # so we can extend it by rowid.
    dr.update({'rowid': id})
    cursor.execute(up,dr)
    return template('response', data = data,
                    rowid = id,
                    text = 'Updated book record')
```

In this case we have a small subtlety: the update instruction needs a rowid key/value pair, whereas the template `response.tpl` does not. We solve this by making a copy `dr` of the data dictionary and updating this suitably.

- ▷ 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>
```

- ▷ The result is



- ▷ Again, we use the template `response.tpl`, which we fill with a different message.

Chapter 9

Image Processing

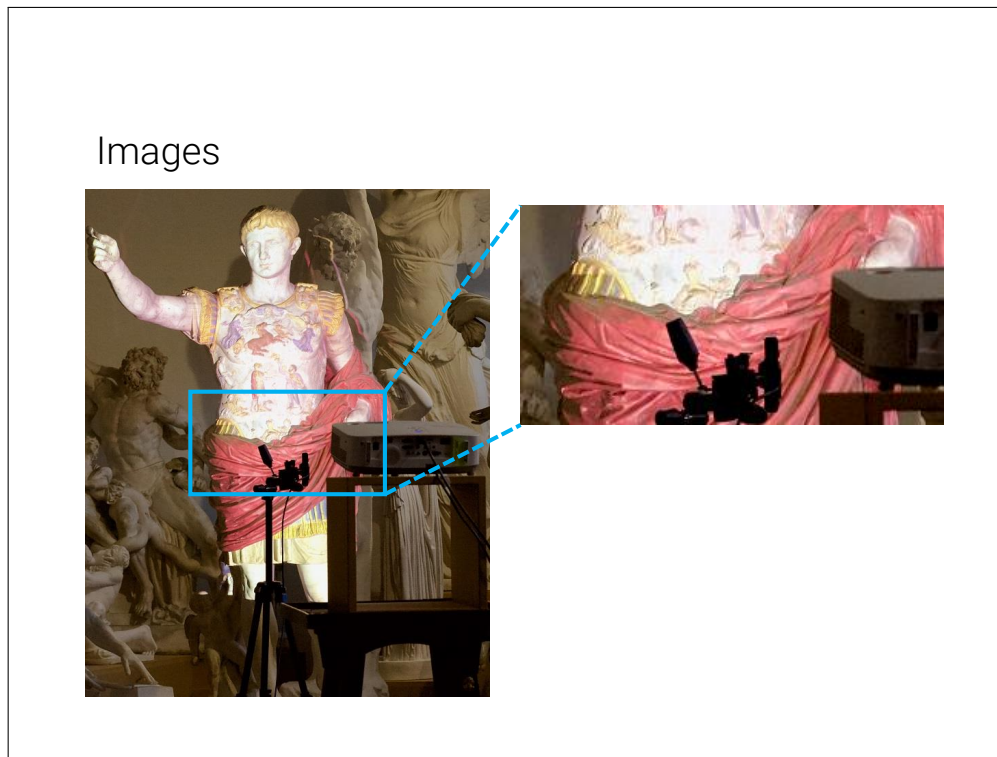
We will now begin a new topic on our way to a useful image database. In particular we will see how computer scientists think about images, how images are represented in computer memory and what we can do with them.

Images



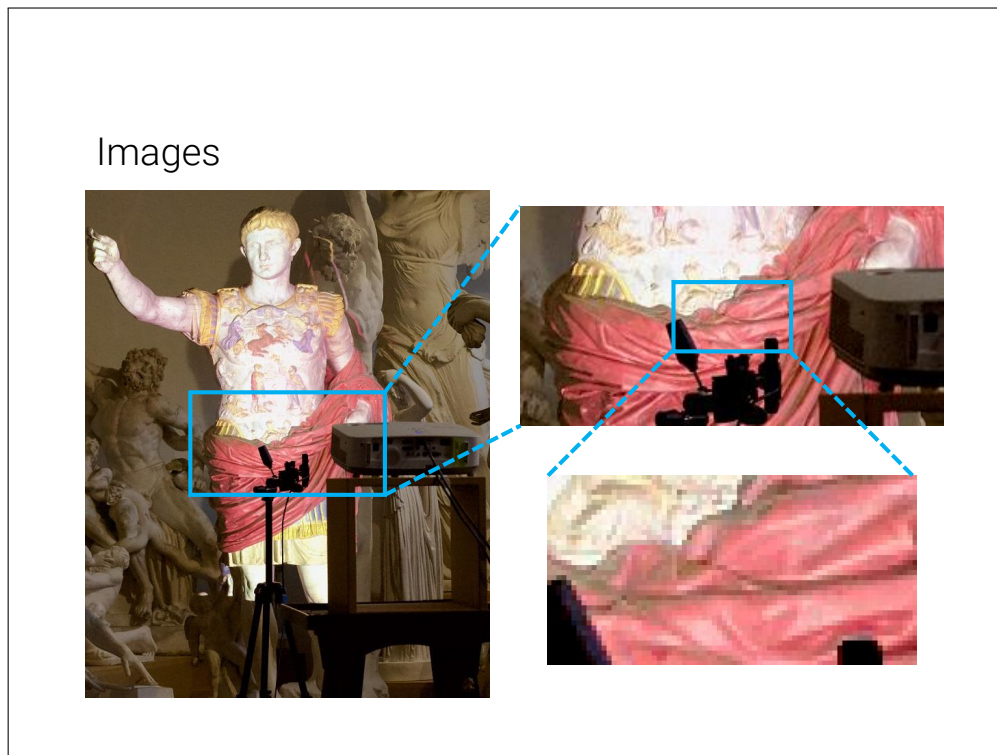
Slide 233

We see here an image taken by a standard DSLR camera. Let's zoom in on it.



Slide 234

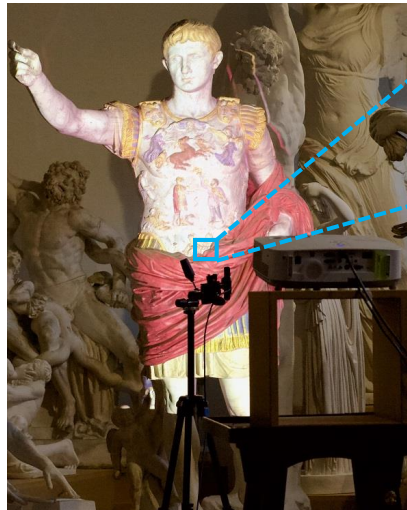
And a bit more...



Slide 235

When zooming in on an image, we start to see blocks of colors, which are organized in a regular grid.

Raster (Pixel) Graphics



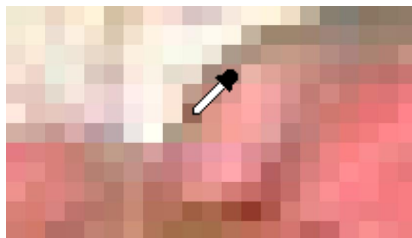
Colors are arranged in a two-dimensional **grid** (raster).

A single entry in this grid is called **pixel**.

Slide 236

We call the grid **raster** and each entry in it **pixel**.

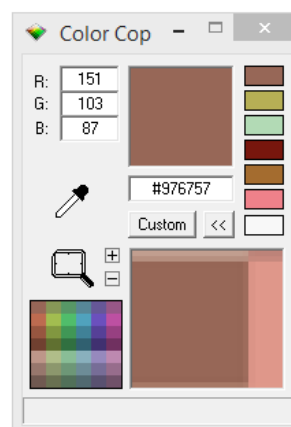
Colors



Colors are usually stored in (R,G,B) format.
(3 **channels**)

$R, G, B \in [0, 255]$ -> One Byte per channel per pixel.

Images in this format can store
 $256 \times 256 \times 256 = 256^3 \approx 16$ million colors.



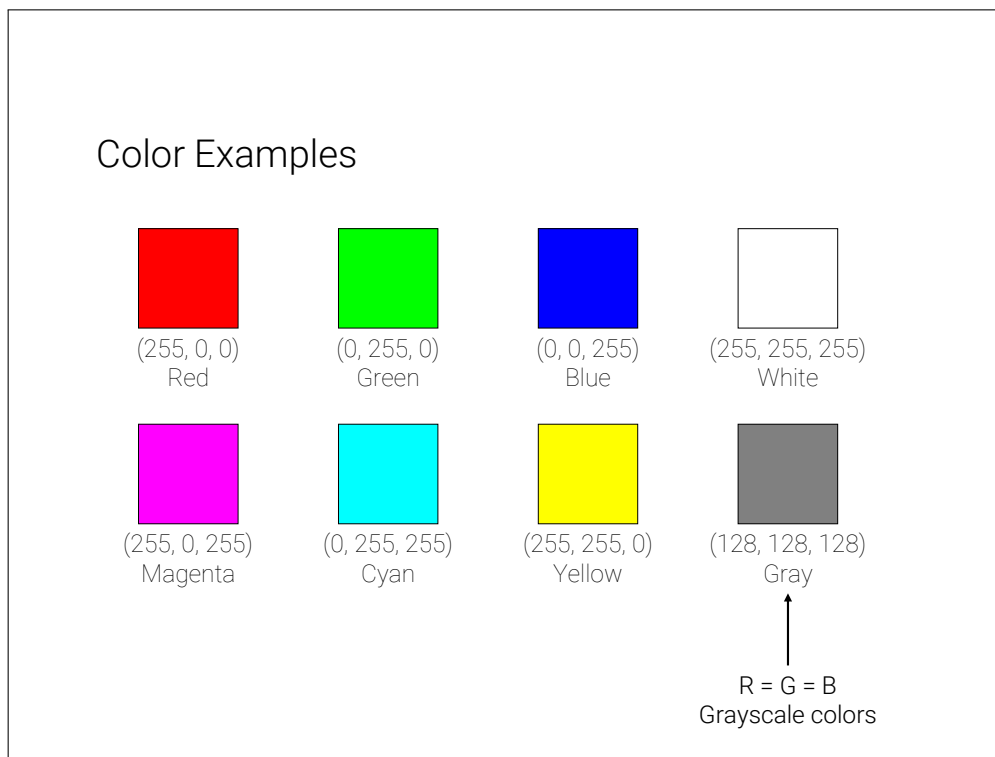
Slide 237

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 ¹.

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.



Slide 238

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).

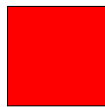
¹<http://colorcop.net/>

Normalized Color Values

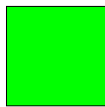
Rather than thinking of a pixel value of being between 0 and 255, it is beneficial to think in terms of **normalized color values**, between 0 and 1.

Values are still stored as Bytes, but normalized before use:

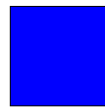
$$v' = v / 255$$



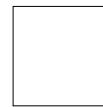
(1, 0, 0)
Red



(0, 1, 0)
Green



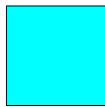
(0, 0, 1)
Blue



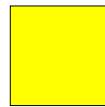
(1, 1, 1)
White



(1, 0, 1)
Magenta



(0, 1, 1)
Cyan



(1, 1, 0)
Yellow



(0.5, 0.5, 0.5)
Gray

Slide 239

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.

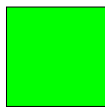
HTML Color Codes

Shorthand notation for colors.

Encode (R,G,B) as hexadecimal numbers.



#FF0000
Red



#00FF00
Green



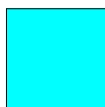
#0000FF
Blue



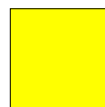
#FFFFFF
White



#FF00FF
Magenta



#00FFFF
Cyan



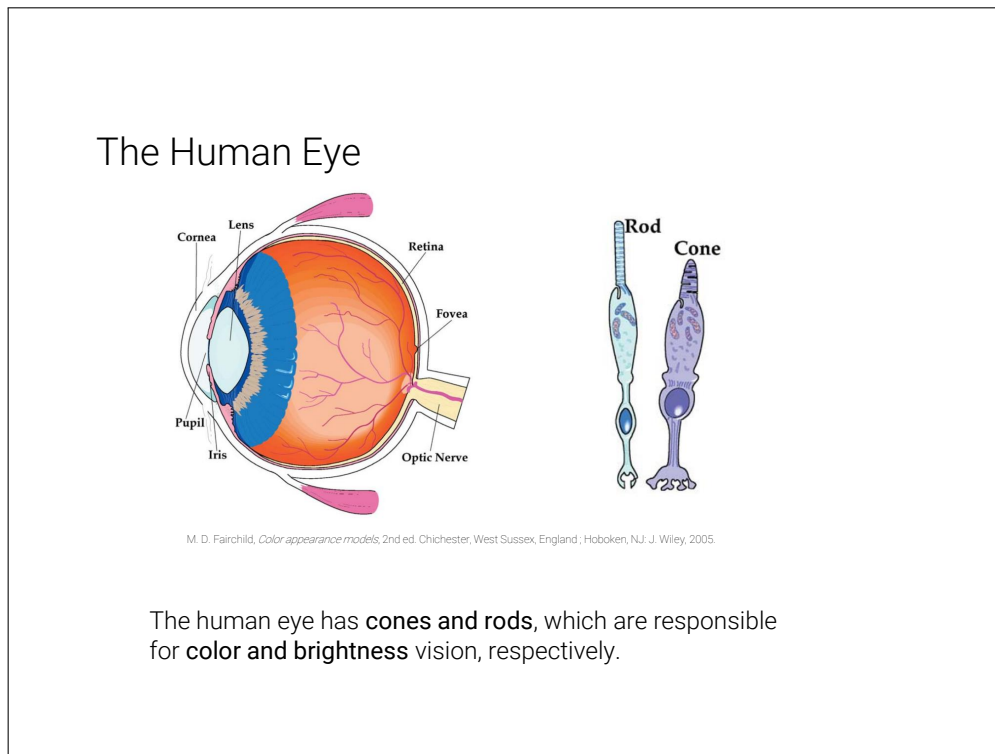
#FFFF00
Yellow



#808080
Gray

Slide 240

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.

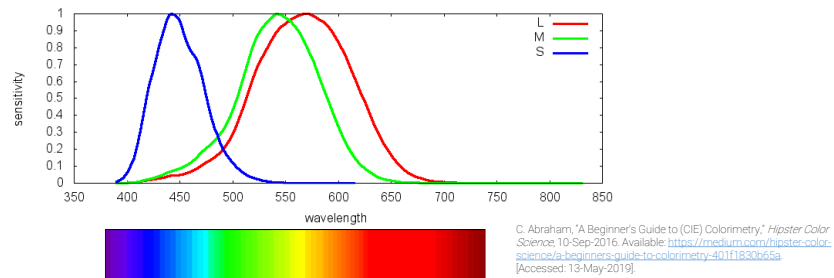


Slide 241

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.

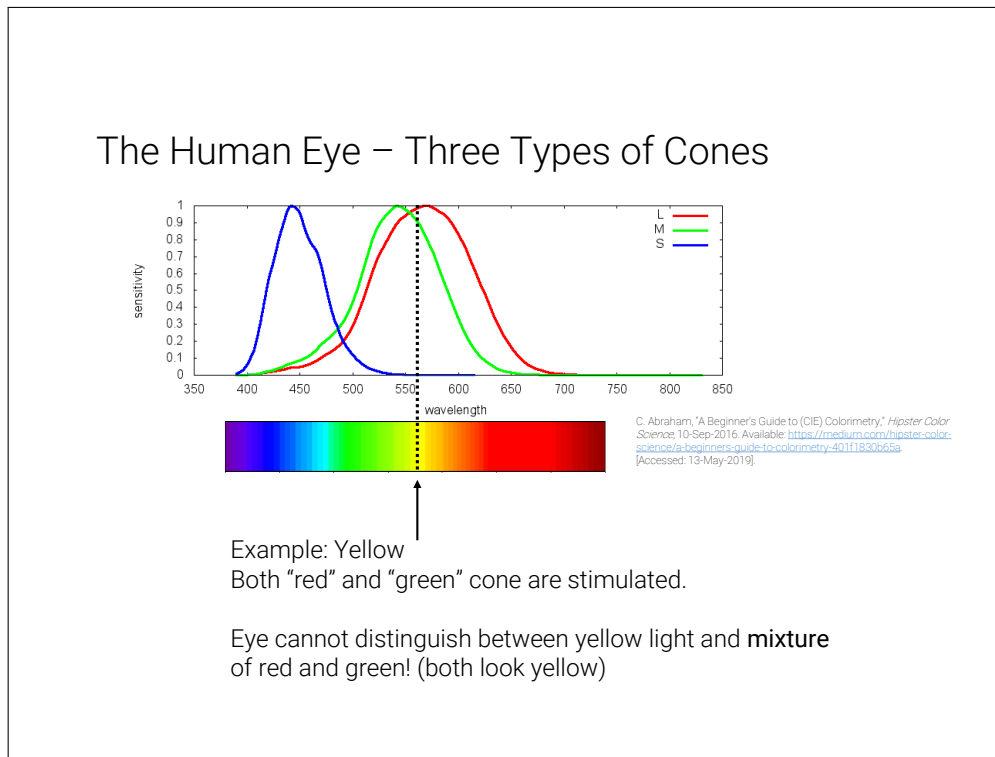
The Human Eye – Three Types of Cones



Slide 242

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).



Slide 243

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.

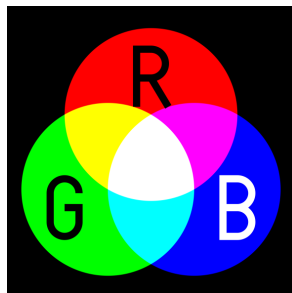
However, let's imagine we perceive a mixture from red and green light. In this case these two cones will be stimulated, too! Our brain is incapable of distinguishing between these two scenarios, since the physical stimulus on our eye is the exact same!

It turns out that we can create all colors as a mixture of red, green, and blue light.

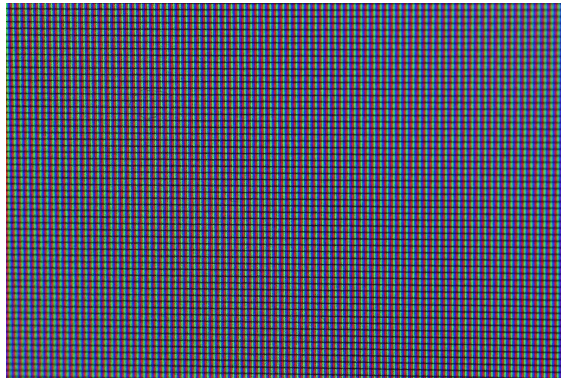
Monitors

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.png> [Accessed 06-June-2019].



Devcore, TFT Bildschirm RGB Pixel 2012 Available:
https://commons.wikimedia.org/wiki/File:TFT-Bildschirm_RGB_Pixel.JPG [Accessed 06-June-2019].

Slide 244

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 x 746 x 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.

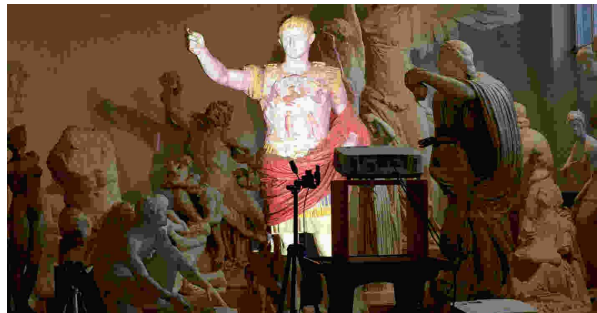
Take our Augustus image again. It is 1440 pixels wide and 746 pixels high. Since each pixel stores three channels, which each measure one Byte, we can calculate the image size: $1440 \times 746 \times 3 = 3222720$ Bytes. On disk however images are usually smaller.

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).



AugustusCompressed.jpg

6/7/2019 9:11 AM

JPEG image

27 KB

In this example we turned the JPEG compression very high, which leads to a tiny file size but strong artifacts 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))
```

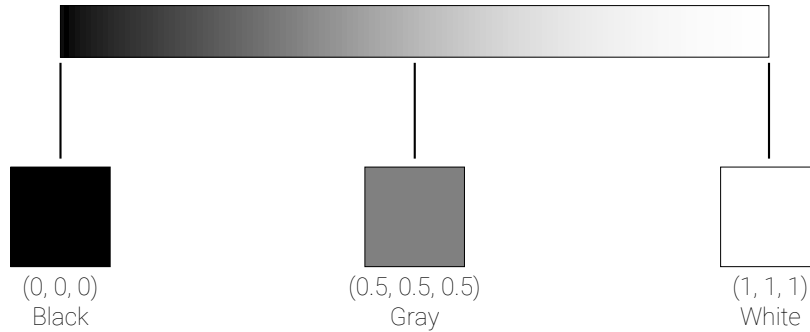
Slide 247

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.

Grayscale Images

$$R = G = B$$



If all channels have the same value, why store all three?
Grayscale images usually have **only one channel**.

Slide 248

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



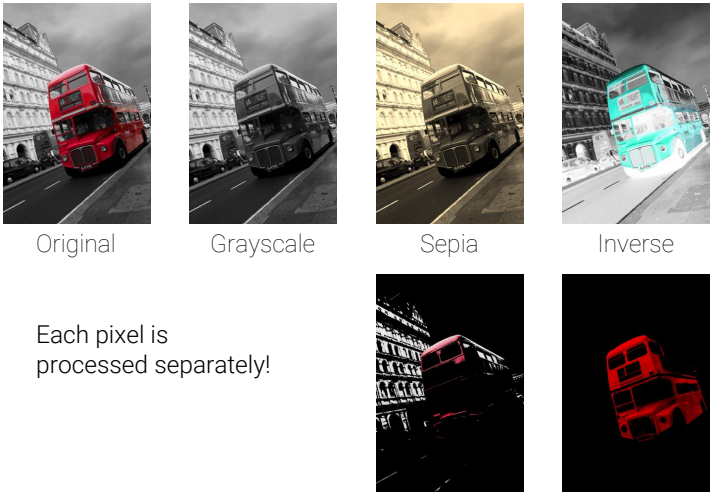
$$\text{Gray} = 0.21 \times R + 0.71 \times G + 0.08 \times B$$

Humans are very sensitive to green.
Green is therefore weighted higher than red and blue.

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

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 251

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!




Slide 252

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



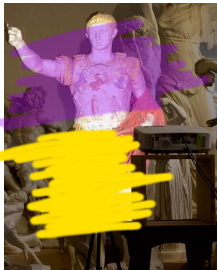
+

$(R,G,B,A) = (0.6, 0.0, 1.0, 0.5)$
Half transparent purple



=

$(R,G,B,A) = (1, 1, 0, 1)$
Full yellow



$$R_{\text{target}} = (1-A) \times R_{\text{augustus}} + A \times R_{\text{purple,yellow}}$$

$$G_{\text{target}} = (1-A) \times G_{\text{augustus}} + A \times G_{\text{purple,yellow}}$$

$$B_{\text{target}} = (1-A) \times B_{\text{augustus}} + A \times B_{\text{purple,yellow}}$$

Slide 253

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.

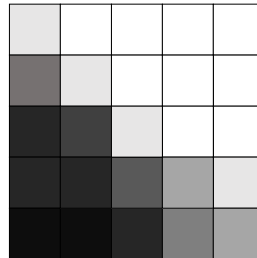
Note that this is again a per-pixel operation, which can easily be implemented with a simple for-loop.

Edge Detection

Goal: Find interesting parts of image (features).

Example: Find **edges**, i.e. sections, where **color changes rapidly**.

Example Image:



Slide 254

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.

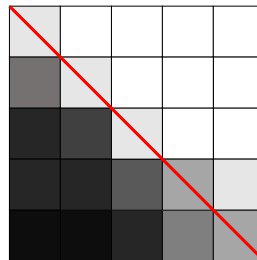
For example, let's say we want to find so-called **edges** in our image, i.e. areas where the color changes rapidly. Edges often correspond to object outlines. We will see an example later.

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 255

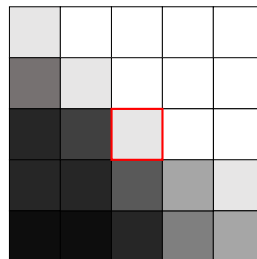
In this (admittedly simple) example image, we can clearly see, that there is an edge present, where the color shifts fast from dark to light. We will now explore, how we can detect such an edge automatically.

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?

Slide 256

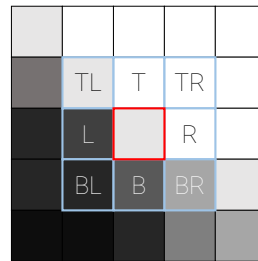
The idea is to decide for each pixel if it is part of an edge or not (binary decision, yes or no). Let's take the marked pixel as example, but remember that the following operations are performed on each pixel in the image.

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.

Slide 257

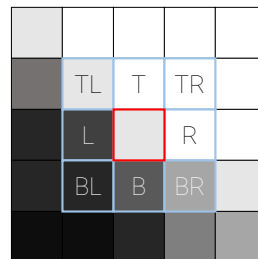
Let's consider the neighbors of our marked pixel.

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:

$$[I_R - I_L] + [I_{TR} - I_{TL}] + [I_{BR} - I_{BL}] > \text{Threshold}$$

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:

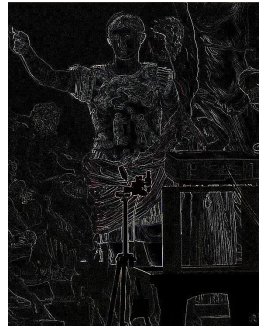
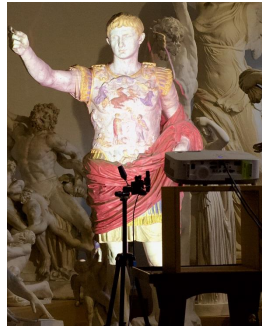
	-1	0	1	
	-2	0	2	
	-1	0	1	

The operation we described here is called **Sobel filter**², 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.

²https://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()
```

Slide 260

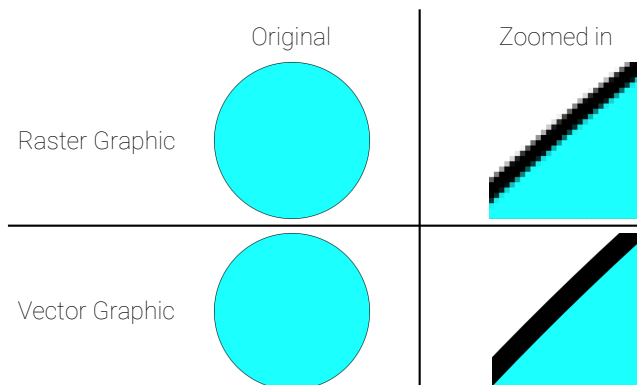
Here we see an example of edge detection. White pixels in the right image are pixels, which were classified as edge pixels, i.e. pixels where large changes in color are present. Black pixels are no edges.

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!



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)!

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.

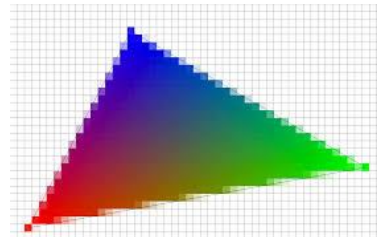
Vector Graphics Display

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 263

Note that most monitors cannot display vector graphics. There are vector monitors, but they are not common.

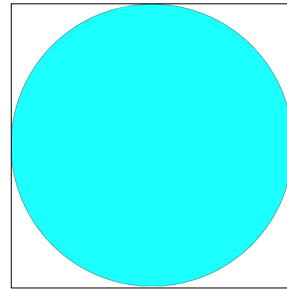
The monitor displayed here does not have pixels. It instead moves a laser and traces a polygon (the asteroids and spaceship). The laser stimulates a phosphor layer, which then glows.

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 264

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:

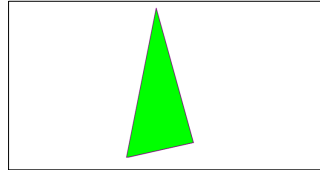
```
<image xlink:href="..." x="..." y="..." width="..." height="..." />
```

Slide 265

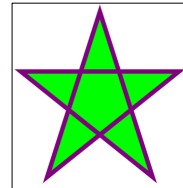
Here are some examples of SVG primitives.

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>
```



Slide 266

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>
```

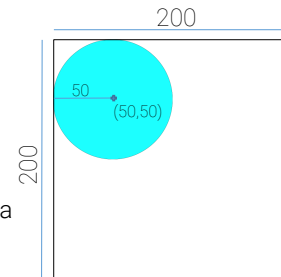
Slide 267

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.

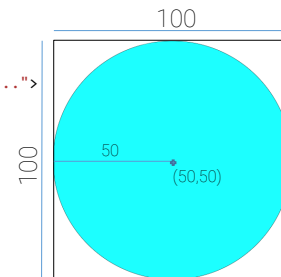


Introducing **viewBox**:

```
<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 268

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:

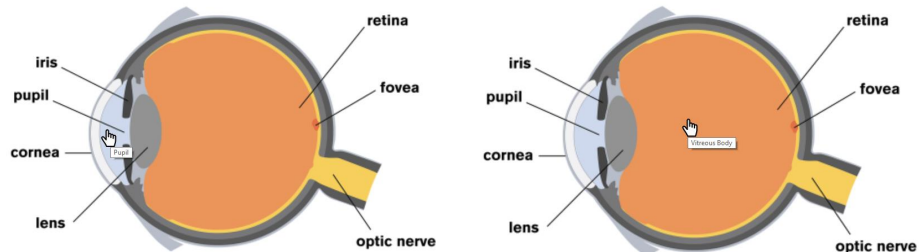
- Mark interesting areas and provide meta data.
- Display annotated information.

Slide 269

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

Tobii AB, "The human eye," <https://www.tobii.com>, 2019.
Available: https://www.tobii.com/imagevault/publishedmedia/4a2hi9g5oq7422kscdn8/Structures_Of_The_Human_Eye.png
[Accessed: 06-June-2019].

Slide 270

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>
```

**** 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 271

We add a new attribute to our **** 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 272

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].

Slide 273

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>`.

Slide 274

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-von-unten-1536x1024.jpg>
[Accessed: 11-June-2019].

Slide 275

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).

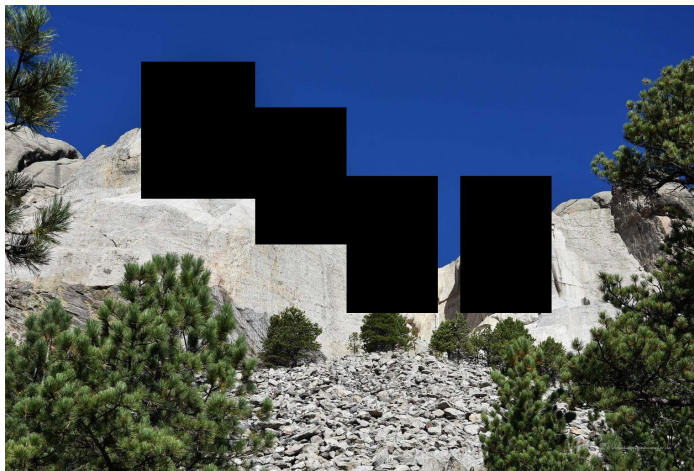
Slide 276

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 278

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;
}
```

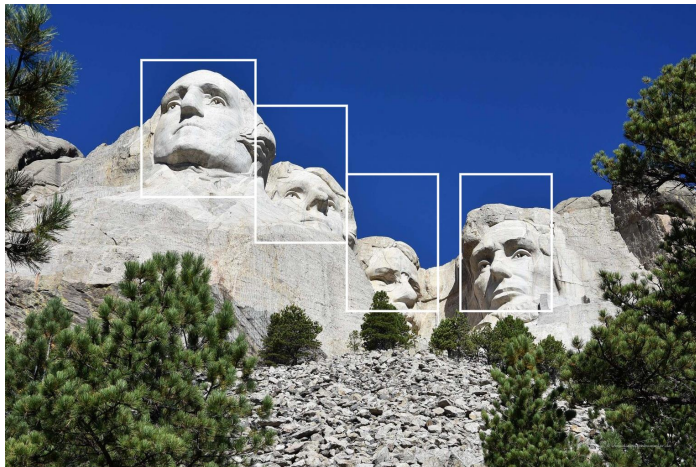
Simple CSS stylesheet. `<rect>`s are given no fill, and a white stroke.

Slide 279

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].

Slide 280

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 281

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-von-unten-1536x1024.jpg>
[Accessed: 11-June-2019].

Slide 282

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 283

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 284

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-von-unten-1536x1024.jpg>
[Accessed: 11-June-2019].

Slide 285

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!

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-unten-1536x1024.jpg>
[Accessed: 11-June-2019]

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 288

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); }
```

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>
```



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 291

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.

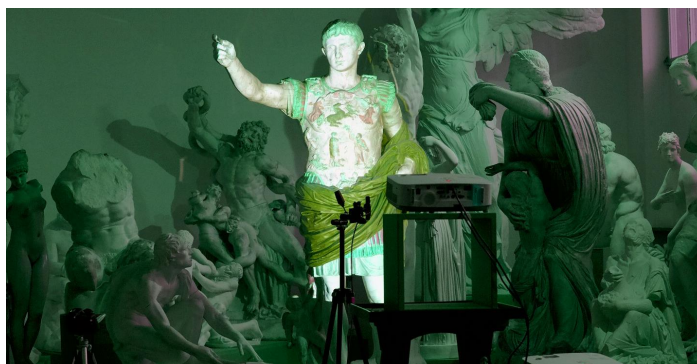
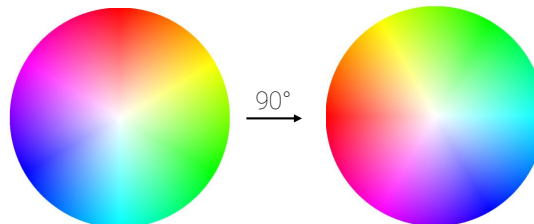
CSS Hue Rotate

```
<html>
<body>

<style>
  img { filter: hue-rotate(90deg); }
</style>



</body>
</html>
```



Slide 292

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!

Slide 293

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 294

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 ``. 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 295

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>
```

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 10

Legal Foundations of Information Technology

In this Chapter, we cover a topic that is a very important secondary aspect of our work as Computer Scientists: the legal foundations that regulate how the fruits of our labor are appreciated (and 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](#).

10.1 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 10.1.1** The concept of [intellectual property](#) motivates a set of laws that regulate [property rights](#) 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 10.1.2** **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 10.1.3** Ownership involves multiple rights (the **property rights**), which may be separated and held by different parties.
- ▷ **Definition 10.1.4** 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 10.1.5** 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 10.1.6** 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* [Smith:WoN1776] 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 Kohlhasse

299



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 juridicative 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 10.1.7** Legal systems in the **common law tradition** are usually based on case law, they are often derived from the British system.
- ▷ **Definition 10.1.8** 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 Kohlhasse

300



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 10.1.9** 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 10.1.10** 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



10.2 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 10.2.1** 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**)

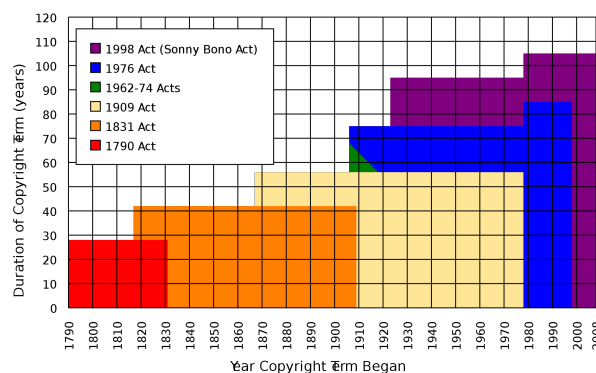


In short almost all products of intellectual work are copyrightable, but this does not mean copyright

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 can 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 10.2.2** A work is said to be in the **public domain**, if no copyright applies, otherwise it is called **copyrighted**.
- ▷ **Example 10.2.3** 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 10.2.4 (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 10.2.5** 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 10.2.6** Formally, it is not the copyrightable work that can be owned itself, but the copyright.

▷ **Definition 10.2.7** 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.



©: Michael Kohlhasse

304



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 10.2.8** 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 10.2.9 (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.



©: Michael Kohlhasse

305



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 10.2.10 (Fair Use/Fair Dealing Doctrines)** Case law in common law jurisdictions 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.



10.3 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 10.3.1** 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

307



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 10.3.2** **Free/Libre/Open-Source Software (FLOSS)** is software that is and licensed via licenses that ensure that its source 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 Kohlhasse

308



The relatively complex name Free/Libre/Open Source comes from the fact that the English¹ 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 bringing 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 10.3.3** A **copyleft** license is a license which requires that allows derivative works, but requires that they be licensed with the same license.
- ▷ **Definition 10.3.4** 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 Kohlhasse

309



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.


¹the movement originated in the USA

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 via Open Content Licenses

- ▷ **Recall:** FLOSS licenses have created a vibrant public domain for software.
- ▷ **How about:** other copyrightable works: music, video, literature, technical documents

Definition 10.3.5 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)



10.4 Information Privacy

Information/Data Privacy

- ▷ **Definition 10.4.1** The principle of **information privacy** comprises the idea that humans have the right to control who can access their personal data when.
- ▷ 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 search engines that make access to information easy and efficient.
- ▷ The “reasonable expectation of privacy” is regulated by special laws.
- ▷ These laws differ considerably by jurisdiction; Germany has particularly stringent regulations (and you are subject to these.)
- ▷ **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.



Organizational Measures or Information Privacy (under German Law)

- ▷ **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.



The General Data Protection Regulation (GDPR)

- ▷ **Definition 10.4.2** The **General Data Protection Regulation (GDPR)** is a **EU regulation** created in 2016 to unite principals of data privacy 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 10.4.3** 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



The General Data Protection Regulation (GDPR)

- ▷ **Definition 10.4.4** **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** 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.



GDPR Customer-Service Requirements

- ▷ 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 11

Ontologies, Semantic Web, & WissKI

In the last Chapter of 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.

11.1 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.

Documenting our Cultural Heritage

- ▷ **Definition 11.1.1** **Cultural heritage** is the legacy of physical artifacts and intangible attributes 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

316



Categories of Data in DigihumS and their IWGS 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 9)

- ▷ e.g. books identified by author/title/publisher/pubyear. (see Chapter 8)
- ▷ **Narrative data**: documents and text fragments used for communicating knowledge to humans.
 - ▷ e.g. plain text and formatted text with **markup codes** (see Chapter 3)
- ▷ **Symbolic data**: descriptions of object and facts in a formal language
 - ▷ e.g. 3+5 in python (see Chapter 2)
- ▷ **Definition 11.1.2 Metadata**: “data about data”, e.g. who has created these facts, images, or documents. (not covered yet)
- ▷ **Observation 11.1.3** *We will need all of these – and their combinations – to do DigiHumS.*



WissKI: a Virtual Research Env. for Cultural Heritage

- ▷ **Requirements**: For a virtual research environment for Cultural Heritage
 - ▷ 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
- ▷ Development on the WissKI started at FAU Erlangen-Nürnberg by the research group of Prof. Günther Görtz and is now used in more than 100 DH projects across Germany.



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.

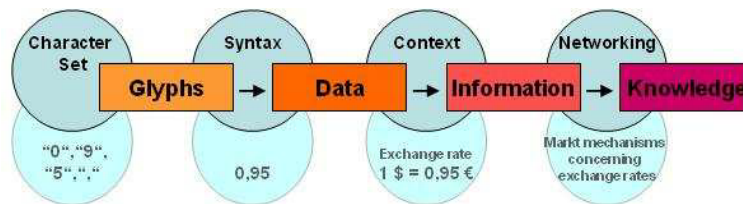


11.2 Semantic Web Technologies

11.2.1 The Semantic Web

The Semantic Web

- ▷ **Definition 11.2.1** 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 11.2.2** We want to find 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 can read (and understand) the text and combine the information to get the answer.



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 to 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

meaning. Conventional wisdom is that we add some semantic/functional markup. Here we pick XML without loss of generality, and characterize some fragments of text e.g. as dates.

Solution: XML markup with “meaningful” Tags

[illegible]

©: Michael Kohlhasse

323



What can we do with this?

▷ **Example 11.2.3** Consider the following fragments:

<title>WWW€//€
 $\mathcal{T}(\downarrow)\dagger\sqsubseteq\sqcup(\mathcal{I}\sqcup\downarrow\nabla\backslash\cup)\chi\backslash\dagger\mathbb{W}\nabla\ddagger\mathbb{W}\prod\mathbb{W}[C]\{\downarrow\nabla\}\sqcup$
</title>
<place> $S(\downarrow\nabla\cup\mathbb{L}\mathbb{W})||)||\mathbb{H}\sqcup\dagger\mathbb{H}\chi\ddagger\cap\ddagger\cap\Leftarrow\mathcal{H}\sqsupseteq\vdash\rangle\Leftarrow USA$
</place>
<date> $\aleph_{\infty\infty}\mathcal{M}\vdash\Leftarrow//€$
</date>

Given the markup above, we can

- ▷ parse $\mathbb{R}_{\infty\infty\mathcal{M}\dashv\vdash\in}\mathbb{N}$ as the date May 7-11 2002 and add this to the user's calendar.
- ▷ parse $\mathcal{S}(\langle \nabla \dashv \sqcup \mathcal{W} \rangle \parallel \parallel) \mathcal{H} \sqcup \uparrow \mathcal{H} \setminus \downarrow \uparrow \uparrow \sqcap \Leftrightarrow \mathcal{H} \dashv \sqsubseteq \dashv \rangle \Leftrightarrow \mathcal{U} \mathcal{S} \mathcal{A}$ as a destination and find flights.

But: do not be deceived by your ability to understand English



©: Michael Kohlhasse

324

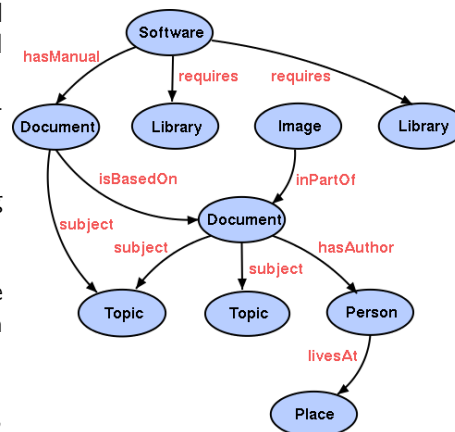


We have to obfuscate the markup as well, since it does not carry any meaning to the machine intrinsically either.

- ▷ What the machine sees of the XML

$$\langle U \rangle U \updownarrow \rangle WWW \in \mathbb{H} \in$$

- ▷ **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 Kohlhasse

327



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 [w3c:rdf-concepts] or RDFa [w3c:rdfa-primer] and ontologies on OWL [w3c:owl2-overview]
 - ▷ 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)



©: Michael Kohlhase

328



11.2.2 Semantic Networks

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.

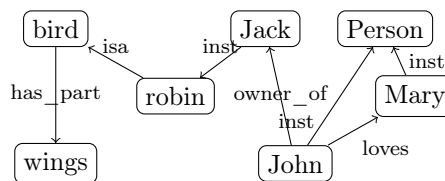
Semantic networks are a very simple way of arranging concepts and their relations in a graph.

Semantic Networks [ColQui:rtsm69]

▷ **Definition 11.2.4** 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 11.2.5** 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



©: Michael Kohlhase

329

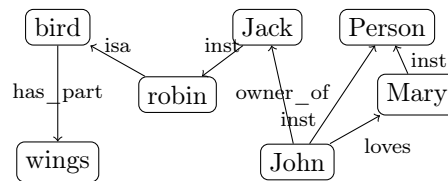


Even though the network in Example 11.2.5 is very intuitive (we immediately understand the concepts depicted), it is unclear how we (and more importantly a machine that does not associate meaning with the labels of the nodes and edges) can draw inferences from the “knowledge” represented.

Deriving Knowledge Implicit in Semantic Networks

▷ **Observation 11.2.6** *There is more knowledge in a semantic network than is explicitly written down.*

▷ **Example 11.2.7** 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.



©: Michael Kohlhase

330



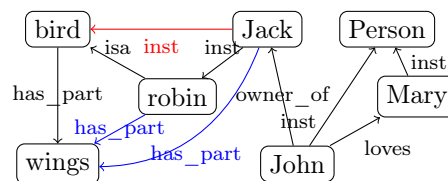
▷ Deriving Knowledge Implicit in Semantic Networks

- ▷ **Definition 11.2.8 (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 .

- ▷ **Intuition:** Derived relations represent knowledge that is implicit in the network; they could be added, but usually are not to avoid clutter.

▷ Example 11.2.9



Slogan: Get out more knowledge from a semantic networks than you put in.



©: Michael Kohlhase

331



Note that Definition 11.2.8 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 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 that we have to keep an eye on.

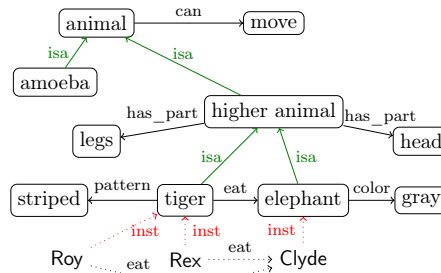
Another problem is that semantic nets (e.g. in in Example 11.2.5) confuse two kinds of concepts: individuals (represented by proper names like *John* and *Jack*) and concepts (nouns like *robin* and *bird*). Even though the “isa” and “inst” links already acknowledge this distinction, the “has_part” and “loves” relations are at different levels entirely, but not distinguished in the networks.

▷ Terminologies and Assertions

- ▷ **Remark 11.2.10** We should keep the “inst” and “isa” links apart – and

distinguish concepts from individuals/objects.

▷ **Example 11.2.11** From the network



infer that *elephants* have *legs* and that *Clyde* is *gray*.

▷ **Definition 11.2.12** 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 .



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.

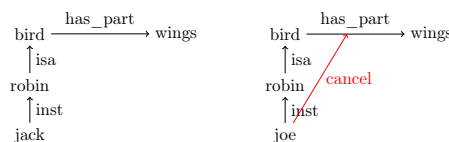
Limitations of Semantic Networks

▷ 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 11.2.13** Consider a robin that has lost its wings in an accident



Cancel-links have been proposed, but their status and process model are debatable.



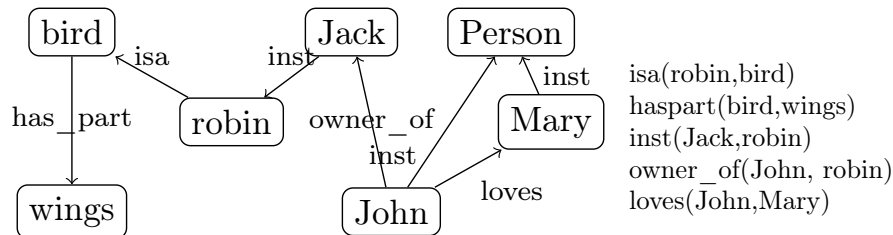
To alleviate the perceived drawbacks of semantic networks, we can contemplate another notation that is more linear and thus more easily implemented: function/argument notation.

Another Notation for Semantic Networks

▷ **Definition 11.2.14** **Function/argument notation** for semantic networks

- ▷ interprets **node** as arguments (reification to individuals)
- ▷ interprets **link** as functions (logical relations)

▷ **Example 11.2.15**



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)



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.

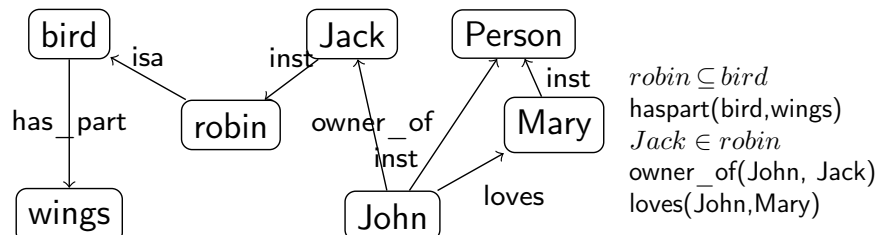
The next slide is a bit outside of the scope of IWGS, but we want to go into this anyway.

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



- ▷ **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.

11.2.3 Ontologies

Based on the intuitions from semantic networks we can now come to general (Semantic Web) ontologies and contrast them to database systems. We will still keep our presentation of the material at a general level without committing to a particular ontology language or system.

What is an Ontology

- ▷ **Definition 11.2.16** 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 11.2.17** A vocabulary often includes names for **classes** and **relationships** (also called **concepts**, and **properties**).
- ▷ **Remark 11.2.18** If the description logic has a reasoner, we can automatically
 - ▷ detect inconsistent axiom systems
 - ▷ compute class membership and taxonomies.



Example: Hogwarts Ontology

- ▷ **Example 11.2.19 Axioms** describe the structure of the model,

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 11.2.20 Facts** describe some particular concrete situation,

Individual: Hedwig

Types: Owl

Individual: HarryPotter

Types: HogwartsStudent

Facts: hasPet Hedwig

Individual: Fawkes

Types: Phoenix

Facts: isPetOf Dumbledore



Ontologies vs. Databases

- ▷ **Obvious Analogy:** Ontology facts analogous to DB data: (structure and constraints on data)

- ▷ **Another one:** Ontology axioms analogous to DB schema

- ▷ Instantiates schema
- ▷ Consistent with schema constraints

- ▷ **But there are also important differences:**

Database:

- ▷ **Closed world assumption (CWA)**
 - ▷ Missing information treated as false
- ▷ **Unique name assumption (UNA)**
 - ▷ Each individual has a single, unique name
- ▷ Schema behaves as constraints on structure of data
 - ▷ Define legal database states

Ontology:

- ▷ **Open world assumption (OWA)**
 - ▷ Missing information treated as unknown
- ▷ No UNA
 - ▷ Individuals may have more than one name
- ▷ Ontology axioms behave like implications (inference rules)
 - ▷ Entail implicit information



DB vs. Ontology Example

▷ 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)

▷ 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



DB vs. Ontology Example

▷ Given: the ontology from the Hogwarts axioms and facts insert

Individual: Dumbledore
Individual: Fawkes
Types: Phoenix
Facts: isPetOf Dumbledore

▷ System Response:

- ▷ DB: Update rejected: constraint violation
 - ▷ Range of hasPet isHuman; 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)



©: Michael Kohlhase

340



DB vs. Ontology: 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 ($\hat{=}$ terrible runtimes)
Implementations may still behave well in typical cases
- ▷ **Definition 11.2.21** We call a query language *semantic*, iff query answering involves derived axioms and facts.
- ▷ **Observation 11.2.22** *Ontology queries are semantic, while database queries are not.*



©: Michael Kohlhase

341



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.



©: Michael Kohlhase

342



11.2.4 The Semantic Web Technology Stack

In this Subsection 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 [RDF1.1primer; RDFS1.1primer; OW2-primer] 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 11.2.23** 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 11.2.24** RDF can be used for describing (all “objects on the WWWeb”)
 - ▷ 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 11.2.25** A **resource** is anything that can have a URI, such as <http://www.fau.de>
- ▷ **Definition 11.2.26** 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 11.2.27** A RDF **statement** (also known as a **triple**) s consists of a resource (the **subject**), a property (the **predicate**), and a property value (the **object** of s). A set of RDF **triples** is called an **RDF graph**.

- ▷ **Example 11.2.28** Statement: *[This slide]^{subj} has been [author]^{pred}ed by [Michael Kohlhase]^{obj}*



©: Michael Kohlhase

344



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 11.2.29** 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

345



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 [DCMI:dcmi-terms:tr].

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.

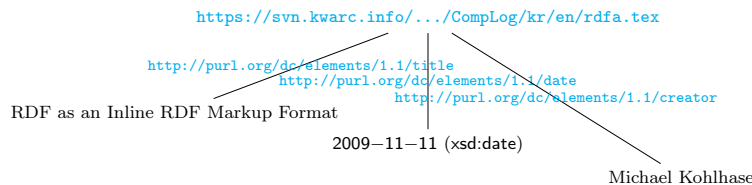
RDFa gives authors a way to embed RDF triples into web resources and make keeping RDF statements about them more in sync.

▷ RDFa as an Inline RDF Markup Format

▷ **Problem:** RDF is a standoff markup format (annotate by URIs pointing into other files)

▷ **Example 11.2.30**

```
<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>
```



©: Michael Kohlhase

346



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 11.2.31** h is the resource for Ian Horrocks, s is the resource for Ulrike Sattler, R is the relation “hasColleague”, and φ 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 an similar language for TBoxes (based on *ALC*)



©: Michael Kohlhase

347



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

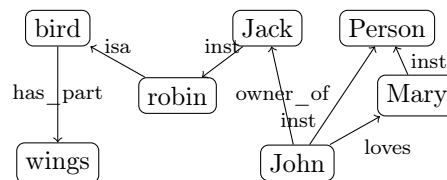
- ▷ **Task:** Complement RDF (ABox) with a TBox language
- ▷ **Idea:** Make use of resources that are values in `rdf:type` (called **Classes**)
- ▷ **Definition 11.2.32** OWL (the **ontology web language**) is a language for encoding TBox information about RDF classes.
- ▷ **Example 11.2.33 (A concept definition for “Mother”)**
 $\text{Mother} = \text{Woman} \sqcap \text{Parent}$ is represented as

XML Syntax	Functional Syntax
<pre><EquivalentClasses> <Class IRI="Mother"/> <ObjectIntersectionOf> <Class IRI="Woman"/> <Class IRI="Parent"/> </ObjectIntersectionOf> </EquivalentClasses></pre>	<pre>EquivalentClasses(:Mother ObjectIntersectionOf(:Woman :Parent))</pre>



Extended OWL Example in Functional Syntax

- ▷ **Example 11.2.34** The semantic network from above can be expressed in OWL (in **functional syntax**)



```

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 11.2.35** A database that stores RDF data is called a **triple store**

▷ **Definition 11.2.36** 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 [PruSea08:sparql].

▷ SPARQL is pronounced like the word “sparkle”.

▷ **Definition 11.2.37** A triple store is called a **SPARQL endpoint**, iff it answers SPARQL queries.

▷ **Example 11.2.38**

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.

}



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.

SPARQL Applications: DBPedia

▷ **Typical Application:** DBPedia screen-scrapes Wikipedia fact boxes for RDF triples and uses SPARQL for querying the induced triple store.

▷ **Example 11.2.39 (DBPedia Query)**

People who were born in Berlin before 1900 (<http://dbpedia.org/sparql>)

PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT ?name ?birth ?death ?person **WHERE** {

 ?person dbo:birthPlace :Berlin .

 ?person dbo:birthDate ?birth .

 ?person foaf:name ?name .

```

?person dbo:deathDate ?death .
  FILTER (?birth < "1900-01-01"^^xsd:date) .
}
ORDER BY ?name

```



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 > 10000000)
} order by ?soccerplayer

```

Results:

SPARQL results:

soccerplayer	countryOfBirth	team	countryOfTeam	stadiumcapacity
:Abdesslam_Benabdellah	:Algeria	:Wydad_Casablanca	:Morocco	67000
:Ailton_Moraes_Michellon	: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_Ferreyra	:Argentina	:FBC_Melgar	:Peru	60000
:David_Bílik	: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_Bolívar	: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



Triple Stores: the Semantic Web Databases

▷ **Definition 11.2.40** A **triplestore** or **RDF store** is a purpose-built database for the storage **RDF graphs** and retrieval of RDF **triples** through **semantic queries**, usually variants of SPARQL.



11.2.5 The Linked Open Data Cloud

Linked Open Data

- ▷ **Definition 11.2.41** **Linked data** is a structured data which is interlinked via **relations** with other data so that become more useful through semantic queries.
- ▷ **Definition 11.2.42** **Linked open data (LOD)** is linked data which is released under an open license, which does not impede its reuse for free.
- ▷ **Definition 11.2.43** Given the Semantic Web technology stack, we can create interoperable ontologies and interlinked data sets, we call their totality the **linked open data cloud**.



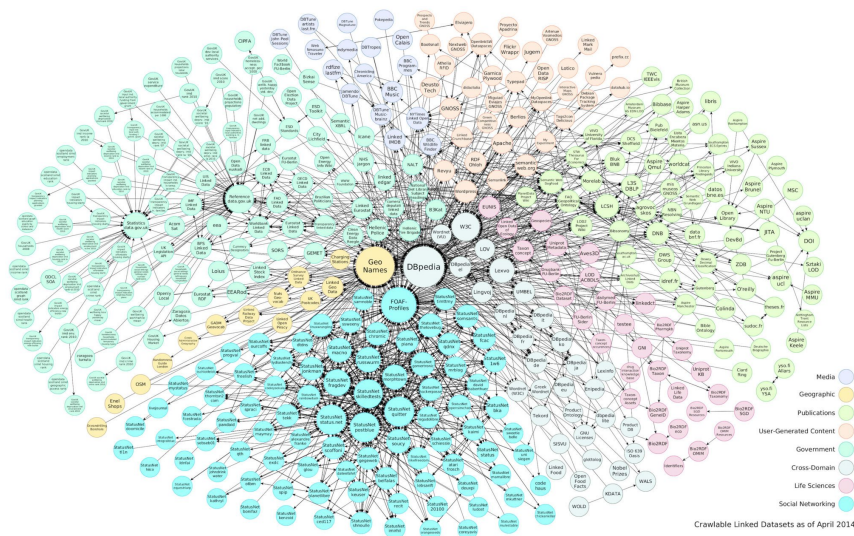
©: Michael Kohlhasse

354



The Linked Open Data Cloud

- ▷ The Linked Open Data Cloud in 2014 (today much bigger, but unreadable)



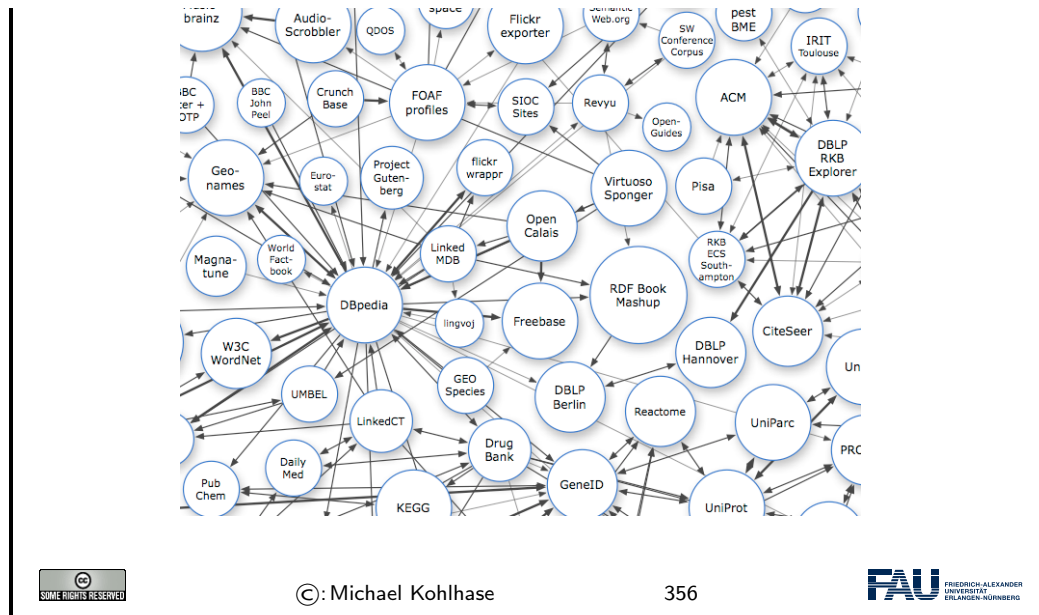
©: Michael Kohlhasse

355



The Linked Open Data Cloud

- ▷ zooming in (data sets and their – interlinked – ontologies)



11.3 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.

Acknowledgements: The presentation in this Section has been derived from WissKI presentations by Martin Scholz and Sarah Wagner. In particular, most of the image are copied from those.

WissKI: a Virtual Research Env. for Cultural Heritage

- ▷ **Requirements:** For a virtual research environment for Cultural Heritage
 - ▷ 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
- ▷ Development on the WissKI started at FAU Erlangen-Nürnberg by the research group of Prof. Günther Görtz and is now used in more than 100 DH projects across Germany.

Cultural Artefacts in Databases I

- ▷ **Example 11.3.1** A typical database for cultural artefacts.

Freitext: unerschlossene Information

HiDA/MIDAS-Datenbank
Projekt zur Nürnberger Goldschmiedekunst

Cultural Artefacts in Databases II

▷ **Example 11.3.2** Another database for cultural artefacts

14 von 127
Neu
Erstellen
Suchen
MS Word
PowerPoint
Drucken
Drucken

Neu
Erstellen
Suchen
MS Word
PowerPoint
Drucken
Drucken

ID	Titel	Genre
10	Einzelbild	Gemälde
11	Einzelbild	Gemälde
12	Einzelbild	Gemälde
13	Einzelbild	Gemälde
14	Einzelbild	Gemälde
15	Einzelbild	Gemälde
16	Einzelbild	Gemälde
17	Einzelbild	Gemälde
18	Einzelbild	Gemälde
19	Einzelbild	Gemälde
20	Einzelbild	Gemälde
21	Einzelbild	Gemälde
22	Einzelbild	Gemälde
23	Einzelbild	Gemälde
24	Einzelbild	Gemälde
25	Einzelbild	Gemälde
26	Einzelbild	Gemälde
27	Einzelbild	Gemälde
28	Einzelbild	Gemälde
29	Einzelbild	Gemälde
30	Einzelbild	Gemälde
31	Einzelbild	Gemälde
32	Einzelbild	Gemälde
33	Einzelbild	Gemälde
34	Einzelbild	Gemälde
35	Einzelbild	Gemälde
36	Einzelbild	Gemälde
37	Einzelbild	Gemälde
38	Einzelbild	Gemälde
39	Einzelbild	Gemälde
40	Einzelbild	Gemälde
41	Einzelbild	Gemälde
42	Einzelbild	Gemälde
43	Einzelbild	Gemälde
44	Einzelbild	Gemälde
45	Einzelbild	Gemälde
46	Einzelbild	Gemälde
47	Einzelbild	Gemälde
48	Einzelbild	Gemälde
49	Einzelbild	Gemälde
50	Einzelbild	Gemälde

Titel: Einzelbild
Genre: Gemälde

Definition: 163

Maße (H): 163 cm

Maße (B): 163 cm

Maße (T): 163 cm

Maße (L): 163 cm

Maße (W): 163 cm

Maße (D): 163 cm

Maße (R): 163 cm

Maße (F): 163 cm

Maße (S): 163 cm

Maße (M): 163 cm

Maße (N): 163 cm

Maße (O): 163 cm

Maße (P): 163 cm

Maße (Q): 163 cm

Maße (U): 163 cm

Maße (V): 163 cm

Maße (X): 163 cm

Maße (Y): 163 cm

Maße (Z): 163 cm

Maße (AA): 163 cm

Maße (AB): 163 cm

Maße (AC): 163 cm

Maße (AD): 163 cm

Maße (AE): 163 cm

Maße (AF): 163 cm

Maße (AG): 163 cm

Maße (AH): 163 cm

Maße (AI): 163 cm

Maße (AJ): 163 cm

Maße (AK): 163 cm

Maße (AL): 163 cm

Maße (AM): 163 cm

Maße (AN): 163 cm

Maße (AO): 163 cm

Maße (AP): 163 cm

Maße (AQ): 163 cm

Maße (AR): 163 cm

Maße (AS): 163 cm

Maße (AT): 163 cm

Maße (AU): 163 cm

Maße (AV): 163 cm

Maße (AW): 163 cm

Maße (AX): 163 cm

Maße (AY): 163 cm

Maße (AZ): 163 cm

Maße (BA): 163 cm

Maße (BB): 163 cm

Maße (BC): 163 cm

Maße (BD): 163 cm

Maße (BE): 163 cm

Maße (BF): 163 cm

Maße (BG): 163 cm

Maße (BH): 163 cm

Maße (BI): 163 cm

Maße (BJ): 163 cm

Maße (BK): 163 cm

Maße (BL): 163 cm

Maße (BM): 163 cm

Maße (BN): 163 cm

Maße (BO): 163 cm

Maße (BP): 163 cm

Maße (BQ): 163 cm

Maße (BR): 163 cm

Maße (BS): 163 cm

Maße (BT): 163 cm

Maße (BU): 163 cm

Maße (BV): 163 cm

Maße (BW): 163 cm

Maße (BX): 163 cm

Maße (BY): 163 cm

Maße (BZ): 163 cm

Maße (CA): 163 cm

Maße (CB): 163 cm

Maße (CC): 163 cm

Maße (CD): 163 cm

Maße (CE): 163 cm

Maße (CF): 163 cm

Maße (CG): 163 cm

Maße (CH): 163 cm

Maße (CI): 163 cm

Maße (CJ): 163 cm

Maße (CK): 163 cm

Maße (CL): 163 cm

Maße (CM): 163 cm

Maße (CN): 163 cm

Maße (CO): 163 cm

Maße (CP): 163 cm

Maße (CQ): 163 cm

Maße (CR): 163 cm

Maße (CS): 163 cm

Maße (CT): 163 cm

Maße (CU): 163 cm

Maße (CV): 163 cm

Maße (CW): 163 cm

Maße (CX): 163 cm

Maße (CY): 163 cm

Maße (CZ): 163 cm

Maße (DA): 163 cm

Maße (DB): 163 cm

Maße (DC): 163 cm

Maße (DD): 163 cm

Maße (DE): 163 cm

Maße (DF): 163 cm

Maße (DG): 163 cm

Maße (DH): 163 cm

Maße (DI): 163 cm

Maße (DJ): 163 cm

Maße (DK): 163 cm

Maße (DL): 163 cm

Maße (DM): 163 cm

Maße (DN): 163 cm

Maße (DO): 163 cm

Maße (DP): 163 cm

Maße (DQ): 163 cm

Maße (DR): 163 cm

Maße (DS): 163 cm

Maße (DT): 163 cm

Maße (DU): 163 cm

Maße (DV): 163 cm

Maße (DW): 163 cm

Maße (DX): 163 cm

Maße (DY): 163 cm

Maße (DZ): 163 cm

Maße (EA): 163 cm

Maße (EB): 163 cm

Maße (EC): 163 cm

Maße (ED): 163 cm

Maße (EE): 163 cm

Maße (EF): 163 cm

Maße (EG): 163 cm

Maße (EH): 163 cm

Maße (EI): 163 cm

Maße (EJ): 163 cm

Maße (EK): 163 cm

Maße (EL): 163 cm

Maße (EM): 163 cm

Maße (EN): 163 cm

Maße (EO): 163 cm

Maße (EP): 163 cm

Maße (EQ): 163 cm

Maße (ER): 163 cm

Maße (ES): 163 cm

Maße (ET): 163 cm

Maße (EU): 163 cm

Maße (EV): 163 cm

Maße (EW): 163 cm

Maße (EX): 163 cm

Maße (EY): 163 cm

Maße (EZ): 163 cm

Maße (FA): 163 cm

Maße (FB): 163 cm

Maße (FC): 163 cm

Maße (FD): 163 cm

Maße (FE): 163 cm

Maße (FF): 163 cm

Maße (FG): 163 cm

Maße (FH): 163 cm

Maße (FI): 163 cm

Maße (FJ): 163 cm

Using the Internet for the Cultural Heritage I

- ▷ **Idea:** Why not use the Internet as a tool

- ▷ it is inherently distributed and networked
- ▷ gives us instantaneous access to information/images/...
- ▷ allows collaboration and discussion (wikis, fora, blogs)



©: Michael Kohlhasse

360



Documents discussing Cultural Artefacts

- ▷ **Example 11.3.3** A text about a cultural artefact (an etching by Dürer)

Beziehung zu Albrecht Dürer? Wie hängen beide Artikel zusammen?



©: Michael Kohlhasse

361



Using the Internet for the Cultural Heritage II

- ▷ **Problems:** with using the Internet as a resource
 - ▷ are often of dubious quality (imprecise, typos, incomplete, ...)
 - ▷ Informationen are primarily written for human consumption
 - ▷ \leadsto not machine-actionable, but full text search works (e.g. Google)
 - ▷ sometimes we can use established structures (e.g. Infobox in Wikipedia)
- ▷ **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
 - ▷ inference (get out more than you put in!)



RDF for Representing Information about Cultural Artefacts

▷ flexible schemata (OWL)

▷ easy data sharing

▷ open standards, free tools

▷ semantic search via SPARQL

▷ **Idea**: We can use RDF like a Mindmap: RDF can

▷ represent relations between objects

▷ classify objects (web resources)

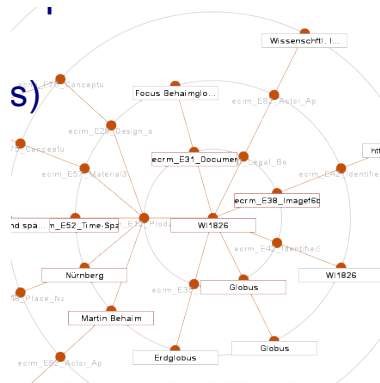
▷ RDFa for document annotation

▷ Reference ontologies for interoperability

▷ SUMO (Suggested Upper Model Ontology) for common knowledge

▷ FOAF (Friend-of-a-Friend) for persons and relations

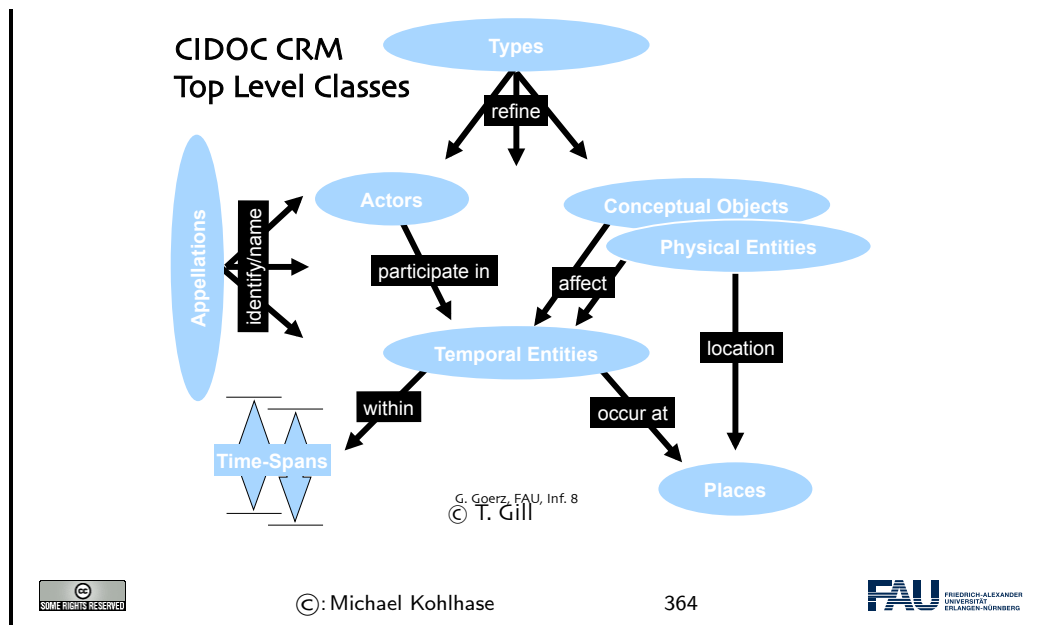
▷ CIDOC-CRM for documentation of cultural heritage



CIDOC CRM (Conceptual Reference Model)

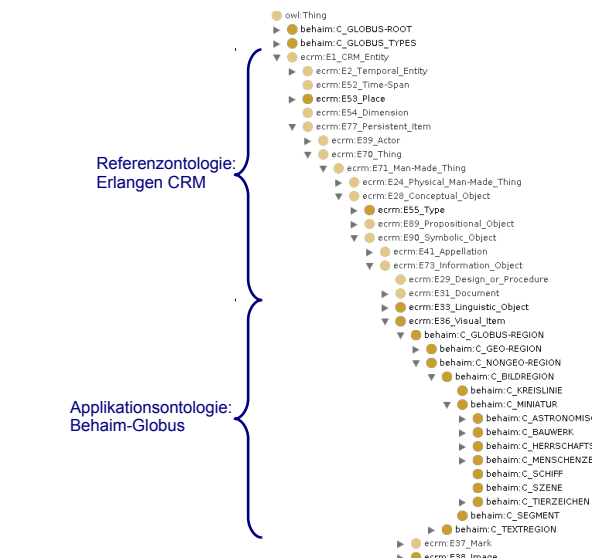
▷ ISO-standardized (ISO-21127)

▷ Reference ontology for the documentation of cultural heritage (museums)



CIDOC CRM (Conceptual Reference Model)

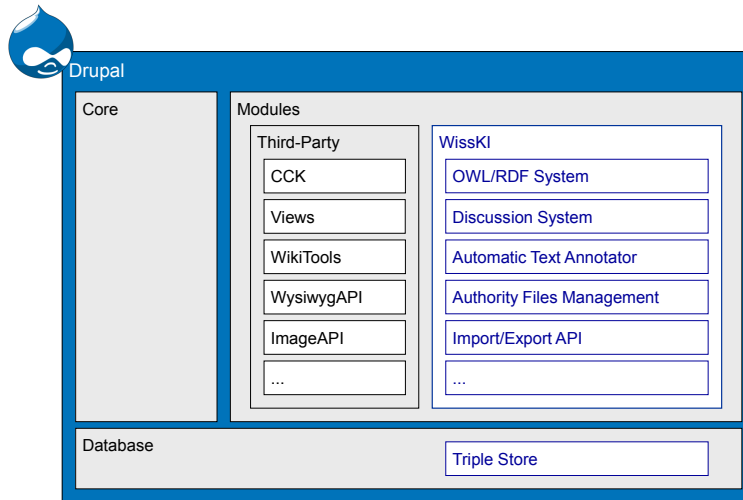
- ▷ Erlangen CRM: Implementation in OWL (enables e.g. Protege support)



WissKI System Architecture

- ▷ Software basis: Drupal CMS (Content Management System)
 - ▷ large, active community, extensible by Drupal modules

- ▷ WissKI $\hat{=}$ WissKI-modules (extend Drupal by semantic functionality)



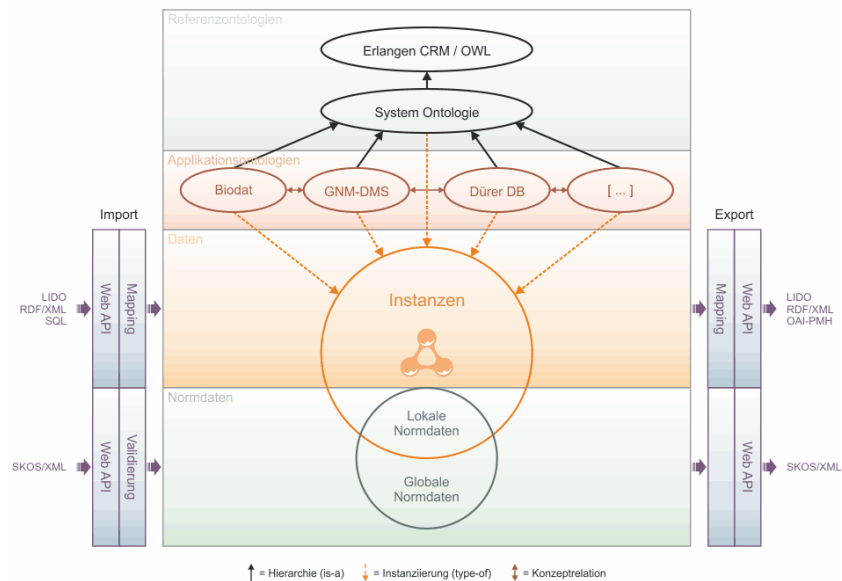
©: Michael Kohlhasse

366



WissKI Information Architecture (Ontologies)

- ▷ Ontologies, instances, and export formats



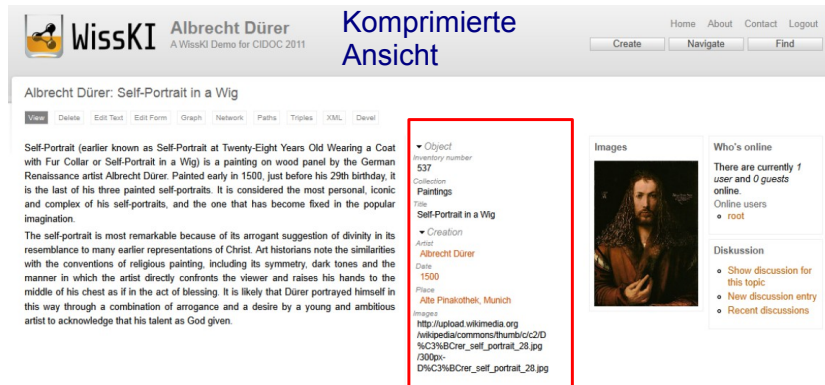
©: Michael Kohlhasse

367

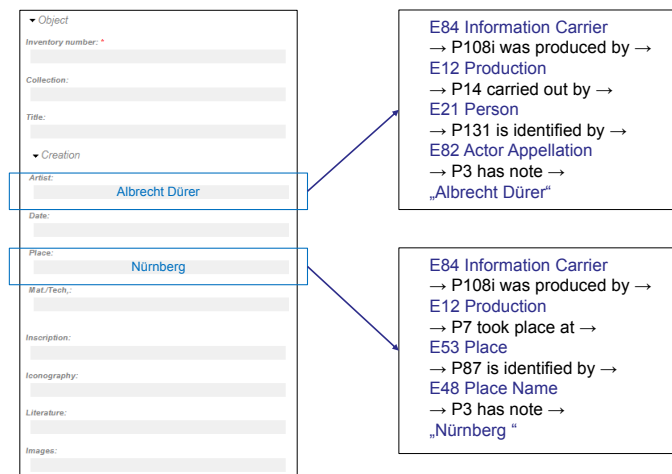


Data Presentation in WissKI

- ▷ **Idea:** hide the complexity induced by the ontology from the user
- ▷ **path constructor:** reduction of ontology constructs to **ontology paths** and **ontology groups**
- ▷ Form-based interaction with categories and fields (as in a RDBMS UI)
- ▷ **Example 11.3.4 (Compressed View)**



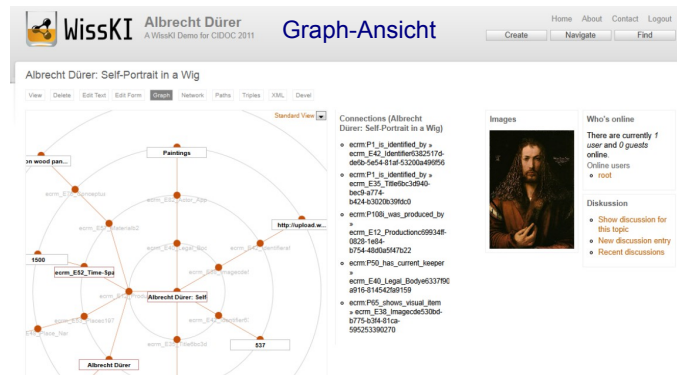
- ▷ **Example 11.3.5 (A WissKI Path Construtor)**



- ▷ **Example 11.3.6 (A WissKI Path Construtor as a Table)**

Menu item	Path	Enabled	Operations
+ Ungrouped		<input checked="" type="checkbox"/>	
+ Museumsobjekt	Group [ecm:E84_Information_Carrier]	<input checked="" type="checkbox"/>	edit delete
+ Inventarnummer	ecm:E84_Information_Carrier -> ecm:P1_is_identified_by -> ecm:E42_Identifier	<input checked="" type="checkbox"/>	edit delete
+ Abteilung	ecm:E84_Information_Carrier -> ecm:P50_has_current_keeper -> ecm:E40_Legal_Body -> ecm:P1_is_identified_by -> ecm:E82_Actor_Appellation	<input checked="" type="checkbox"/>	edit delete
+ Bezeichnung	ecm:E84_Information_Carrier -> ecm:P1_is_identified_by -> ecm:E41_Appellation	<input checked="" type="checkbox"/>	edit delete
+ Titel	ecm:E84_Information_Carrier -> ecm:P1_is_identified_by -> ecm:E35_Title	<input checked="" type="checkbox"/>	edit delete
+ Herstellung	Group [ecm:E84_Information_Carrier -> ecm:P1081_was_produced_by -> ecm:E12_Production]	<input checked="" type="checkbox"/>	
+ Hersteller	ecm:E84_Information_Carrier -> ecm:P1081_was_produced_by -> ecm:E12_Production -> ecm:P14_carried_out_by -> ecm:E21_Person -> ecm:P14_is_identified_by -> ecm:E82_Actor_Appellation	<input checked="" type="checkbox"/>	edit delete
+ Zeit	ecm:E84_Information_Carrier -> ecm:P1081_was_produced_by -> ecm:E12_Production -> ecm:P4_has_time-span -> ecm:E62_Time_Span	<input checked="" type="checkbox"/>	edit delete
+ Ort	ecm:E84_Information_Carrier -> ecm:P1081_was_produced_by -> ecm:E12_Production -> ecm:P7_took_place_at -> ecm:E53_Location	<input checked="" type="checkbox"/>	edit delete
+ Material	ecm:E84_Information_Carrier -> ecm:P1081_was_produced_by -> ecm:E12_Production -> ecm:P32_used_general_technique -> ecm:E57_Material -> ecm:P1_is_identified_by -> ecm:E75_Conceptual_Object_Appellation	<input checked="" type="checkbox"/>	edit delete
+ Technik	ecm:E84_Information_Carrier -> ecm:P1081_was_produced_by -> ecm:E12_Production -> ecm:P33_used_specific_technique -> ecm:E29_Design_or_Procedure -> ecm:P1_is_identified_by -> ecm:E75_Conceptual_Object_Appellation	<input checked="" type="checkbox"/>	edit delete
+ Inhalt Beschreibung	ecm:E84_Information_Carrier -> ecm:P701_is_documented_in -> ecm:E31_Document	<input type="checkbox"/>	edit delete
+ Inschrift	ecm:E84_Information_Carrier -> ecm:P128_carries -> ecm:E84_Inscription	<input checked="" type="checkbox"/>	edit delete
+ Ikonografie	ecm:E84_Information_Carrier -> ecm:P62_depicts -> ecm:E31_CRM_Entity	<input checked="" type="checkbox"/>	edit delete
+ Literatur	ecm:E84_Information_Carrier -> ecm:P701_is_documented_in -> ecm:E31_Document	<input checked="" type="checkbox"/>	edit delete
+ Bilder	ecm:E84_Information_Carrier -> ecm:P65_shows_visual_item -> ecm:E39_Image -> ecm:P1_is_identified_by -> ecm:E42_Identifier	<input checked="" type="checkbox"/>	edit delete

▷ **Example 11.3.7 (A WissKI Path Construtor as a Graph)**



▷ **Example 11.3.8 (A WissKI Path Construtor as Triples)**



WissKI

Albrecht Dürer
A Wikiold Dump for GIDOC 2011

Triples-Ansicht

[Home](#) [About](#) [Contact](#) [Login](#)

Albrecht Dürer: Self-Portrait in a Wig

[View](#)
[Details](#)
[Edit Text](#)
[Edit Form](#)
[Graph](#)
[Network](#)
[Paths](#)
[Tables](#)
[Tools](#)
[Download](#)

Incoming Subject

[y152a65c-118a-8914-4860-05c7acc0d86c_text](#)
[y152a65c-118a-8914-4860-05c7acc0d86c_text](#)

Outgoing predicate

[rdf:type](#)
[acsm:P10b_was_produced_by](#)
[acsm:P11_identified_by](#)
[acsm:P12c_identified_by](#)
[acsm:P50_has_current_keeper](#)
[acsm:P56_shows_visual_form](#)

Incoming Predicate

[acsm:P12b_is_about](#)
[acsm:P12b_refers_to](#)

Outgoing Object

[acsm:E84_Identifier_Canonical](#)
[acsm:E12_ProductionDate_0608_1464-6754-4846a94722](#)
[acsm:E16a_Identifier_3625175_s66b-56d4-81455200a48566](#)
[acsm:E17c_ThumbnailImage_4714-634-6320326660](#)
[acsm:E40_Legal_Body_037800-0714-0346-4161645469169](#)
[acsm:E30_Imagecase_50064b-6755-4384-01a59525300272](#)



Who's online

There are currently 1 user and 0 guests online.

Online users:

- root

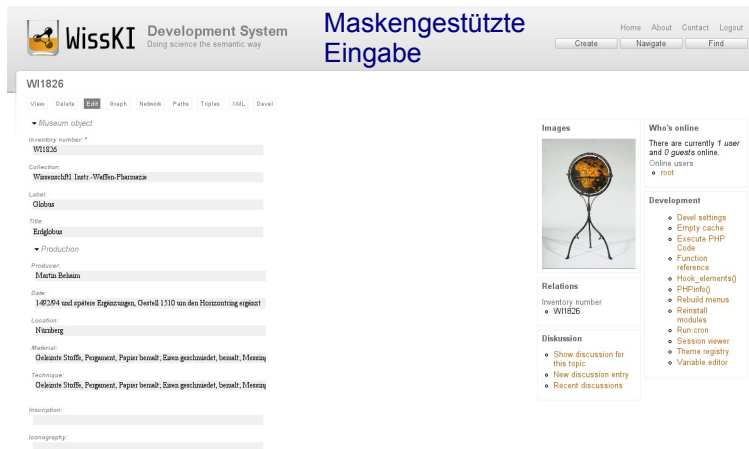
Discussion

- Show discussion for this topic
- New discussion entry

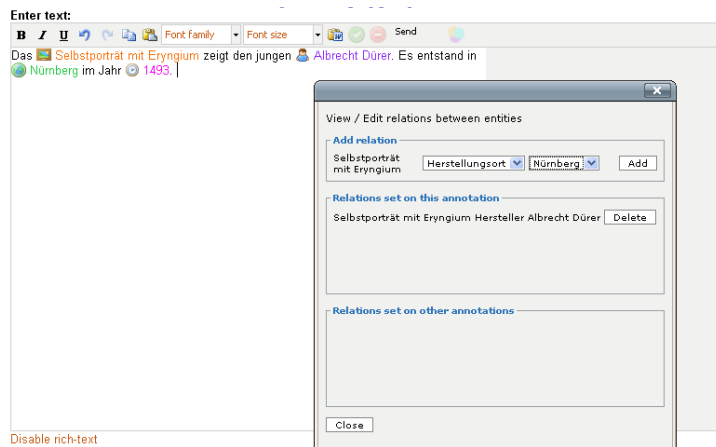
Data Input in WissKI

- ▷ **Idea**: hide the complexity induced by the ontology from the curator
- ▷ **Example 11.3.9 (Form-based data entry)** with autocomplete, external norm authorities

11.3. THE WISSKI SYSTEM: A VIRTUAL RESEARCH ENVIRONMENT FOR CULTURAL HERITAGE²⁴⁵



▷ Example 11.3.10 (Free Text Entry) and fragment annotation



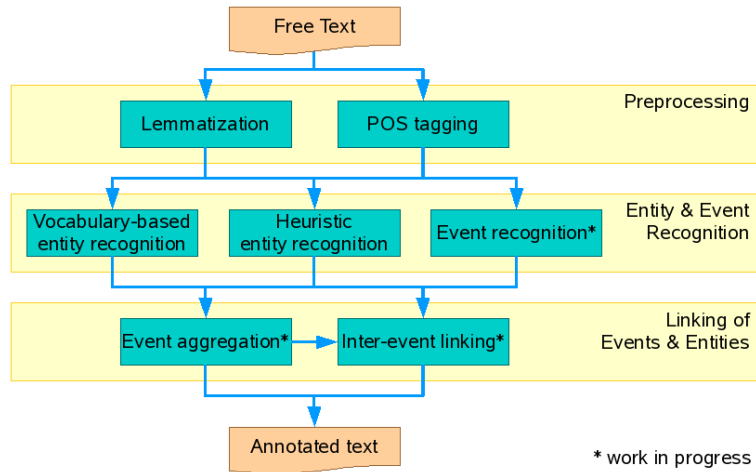
Free Text Entry and Fragment Annotation

- ▷ via a configurable WYSIWYG Editor (Javascript) (MS Word Feeling)
- ▷ Text fragment annotations for
 - ▷ Entities (instances) (categories, relations)
- ▷ Semi-automatic annotation by shallow text analysis (suggestions)
- ▷ Curator can always add, correct, and delete annotations
- ▷ annotations can be harvested and exported.



WissKI: Shallow Text Analysis

▷ Shallow Text Analysis in WissKI



Chapter 12

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
- ▷ Web technologies for interactive documents and applications
 - ▷ Internet infrastructure: web browsers and servers
 - ▷ PHP, dynamic HTML, Javascript, HTML forms
- ▷ Web Application Project (design your own!)



©: Michael Kohlhase

372



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



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).
- [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>.
- [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] *CSS Specificity*. URL: https://en.wikipedia.org/wiki/Cascading_Style_Sheets#Specificity (visited on 12/03/2018).
- [CSSb] *CSS Tutorial*. URL: <https://www.w3schools.com/css/default.asp> (visited on 12/02/2018).
- [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).
- [Ecm] *ECMAScript Language Specification*. ECMA Standard. 5th Edition. Dec. 2009.
- [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>.
- [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/>.
- [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](https://doi.org/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] Folger Karsdorp. *Python Programming for the Humanities*. URL: <http://www.karsdorp.io/python-course/> (visited on 10/14/2018).
- [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^AT_EX 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>.

- [Koh18] Michael Kohlhase. *sTeX: Semantic Markup in T_EX/L^AT_EX*. Tech. rep. Comprehensive T_EX Archive Network (CTAN), 2018. 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).
- [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 (visited on 11/18/2018).
- [PRR97] G. Probst, St. Raub, and Kai Romhardt. *Wissen managen*. 4 (2003). Gabler Verlag, 1997.
- [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>.
- [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).
- [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>.
- [Xam] *apache friends - Xampp*. <http://www.apachefriends.org/en/xampp.html>. URL: <http://www.apachefriends.org/en/xampp.html>.

Index

- sparql
 - endpoint, 234
- ABox, 224, 226
- absolute
 - URI, 80
- academic
 - culture, 4
- algorithm, 13
- American Standard Code for Information Inter-
 - change, 43
- ancestor, 75
- anonymous
 - functions, 35
- Anti-Counterfeiting Trade Agreement, 206
- architectural
 - work, 206
- argument, 35
- arity, 35
- assertions, 224, 226
- assign, 24
- assignee, 131
- attachment, 129
- attribution, 212
- attribute, 73, 135, 141
 - node, 73
- audiovisual
 - work, 206
- authentication, 127
 - factor, 127
- authority, 79
- availability
 - control, 213
- axiom, 226
- balanced
 - bracketing
 - structure, 73
- bare, 120
- base, 41
 - name, 33
- bash
 - console, 21
- basic
 - multilingual
 - plane, 46
- begin
 - tag, 57
- Berne
 - convention, 205
- binary, 14, 42
 - file, 52
 - unit
 - prefix, 54
- bit, 54
- body, 27, 35
- Boolean, 26
 - expression, 27
- border, 65
- Bottle
 - WSGI, 88
- box, 65
- bracketing
 - balanced (structure), 73
- branch, 27, 75, 119
- branching, 27
- browsing, 78
- bug
 - report, 129
- bugtracker, 128
- byte, 54
- called, 35
- Cascading Style Sheets, 62
- central
 - processing
 - unit, 12
- centralized, 120
- character, 46
 - encoding, 46
- checkout, 116
- child, 75
 - table, 141
- choreographic
 - work, 206
- civil
 - law
 - tradition, 205
- class, 36
- clone, 121

- close, 33
- closed
 - world
 - assumption, 227
- closing
 - tag, 73
- code
 - block, 89
 - line, 89
 - point, 46
- column, 135
 - name, 135
 - specification, 138
- comment, 131
- commercial
 - use, 212
- commit, 116
- common
 - law
 - tradition, 205
- compiler, 14
- Complex
 - number, 26
- component, 79
- compose, 15
- composition
 - principle, 15
- computationally
 - universal, 14
- computer
 - hardware, 12
- concept, 222, 226
- class, 226
- concrete
 - data, 215
- condition, 27
- conditional
 - execution, 27
 - statement, 27
- content, 65
- control
 - flow, 27
 - structure, 27
 - word, 56
- cookie, 93
- copyleft, 211
- copyright, 207
 - holder, 208
 - infringement, 208
- copyrightable
 - work, 206
- copyrighted, 207
- CPU, 12
- Creative Commons
 - license, 212
- Cryptography, 127
- cultural
 - heritage, 215
- cursor, 150
- CWA, 227
- cypher
 - text, 127
- dashboard, 21
- data, 12, 133
 - controller, 213
 - definition
 - language, 138
 - language, 14
 - subject, 213
 - transfer
 - control, 213
- database, 133
 - browser, 137
 - management
 - system, 135
 - record, 135
 - schema, 138
 - view, 148
- datum, 133
- DBMS, 135
- DDL, 138
- decimal, 42
- declaration, 62
 - block, 62
- decode, 127
 - key, 127
- default
 - argument, 87
 - value, 87
- DELETE, 83
- delete, 139
- dependency, 131
- derivative
 - works, 212
- derive, 223
- derived, 226
- descendent, 75
- description, 129
 - logic, 226
- development, 125
 - history, 116
- dictionary, 31
- diff
 - file, 116
- digit, 41
- digital
 - text, 52

- digits, 41
- direct
 - identifier, 214
- discussion, 129
- distributed, 120
- document
 - format, 53
 - markup, 56
 - object
 - model, 75, 94
 - renderer, 52
 - root, 73
 - type, 56
- DOM, 75, 94
- dot
 - notation, 37
- downstream, 121
- dramatic
 - work, 206
- DUPLICATE, 131
- dynamic
 - route, 91
- element
 - node, 73
- empty
 - element, 73
 - tag, 57
- encode
 - key, 127
- end
 - tag, 57
- end-user
 - license
 - agreement, 209
- encode, 127
- entity, 141
 - relationship
 - diagram, 141
- ERD, 141
- escape
 - character, 48
 - sequence, 48
- event, 96
- event-handler
 - attribute, 96
- exbi, 54
- exception, 145
- exploitation
 - rights, 208
- expression, 20, 89
- extension, 33
- f-string, 48
- fact, 226
- fair
 - use
 - doctrine, 209
- feature
 - branch, 119, 125
 - request, 129
- fetch, 121
- field), 135
- file, 33
 - name, 33
 - system, 33
- FIXED, 131
- float, 26
- FLOSS, 210
- folder, 33
 - name, 33
- for
 - loop, 31
- foreign
 - key, 141
- fork, 119, 121, 125
- form
 - action, 60
- formatted
 - string
 - literal, 48
 - text, 52
- fragment, 79
- Free/Libre/Open-Source
 - Software, 210
- function, 35
 - object, 35
- function/argument
 - notation, 225
- GDPR, 213
- General
 - Public
 - License, 211
- GET, 83
- gibi, 54
- GIT
 - flow, 125
- group, 126
- handling, 146
- head, 116
 - revision, 116
- headless, 120
- height, 65, 75
- hexadecimal, 42
- higher-order
 - function, 36

- host, 83
- hostname, 83
- http
 - request, 83
- hunk, 117
- hyperlink, 78
- Hypertext
 - Transfer
 - Protocol, 83
- hypertext, 78
- HyperText Markup Language, 57
- idempotent, 83
- information
 - access
 - control, 213
 - privacy, 212
- inheritable, 67
- inheritance
 - factor, 127
- inherits, 67
- input, 12
 - control, 213
- inserting, 139
- integer, 26
- intellectual
 - property, 203
- internal, 126
- Internet, 78
- interpreter, 14
- INVALID, 131
- IRI, 81
- internationalized
 - resource
 - identifier, 81
- Isa-Hierarchy, 224
- ISO-Latin, 45
- issue, 128, 129
 - metadata, 129
 - number, 131
 - tracker, 128
 - tracking
 - system, 128
- iterate, 31
- iteration, 31
- join, 148
- key, 32
- keyword
 - argument, 87
- kibi, 54
- knowledge
 - factor, 127
- label, 131
- leaf, 75
- library, 37
- license, 209
- licensee, 209
- licensor, 209
- linked
 - data, 236
 - open
 - data, 236
- list, 30
- literary
 - work, 206
- local
 - repository, 121
- localhost, 83
- LOD, 236
- loop, 27
- looping, 27
- margin, 65
- markdown, 129
- markup
 - codes, 56
 - format, 56
- master
 - branch, 119, 125
- mebi, 54
- media
 - query, 71
- memory, 12
- merging, 117
- Metadata, 216
- milestones, 131
- musical
 - work, 206
- named
 - wildcard, 91
- namespace, 126
 - declaration, 73
- narrative
 - data, 216
- navigating, 78
- node
 - text, 73
- null
 - value, 135
- object, 36, 226, 231
- obligation
 - of
 - separation, 213
- octal, 42

- ODF, 53
- Office Open XML, 53
- ontology, 226
 - web
 - language, 233
- open, 33
 - world
 - assumption, 227
- Open Office Format, 53
- opening
 - tag, 73
- operator, 20
- output, 12
- OWA, 227
- owner, 204
- ownership, 204
 - factor, 127
- padding, 65
- page
 - inspector, 70
- parameter, 35
 - substitution, 151
- parent, 75, 117
 - table, 141
- participant, 131
- patch, 116
- path, 33, 79
- pebi, 54
- personal
 - group, 126
 - rights, 207
- personally
 - identifiable
 - information, 214
- PGS
 - work, 206
- physical
 - access
 - control, 213
- pictorial, graphic and sculptural
 - work, 206
- PII, 214
- plain
 - text, 52, 127
- positional
 - number
 - system, 41
- POST, 83
- predicate, 231
- primary
 - key, 141
- primitive, 15
- private, 126
- program, 12
- programming, 16
 - language, 13
- properties, 226
- property, 62, 204, 230
 - right, 204
 - value, 230
- pseudonymized, 214
- public, 126
 - domain, 207
- pull
 - request, 125
- punch
 - card, 44
- push, 121
- PUT, 83
- python
 - console, 21
- quasi-identifier, 214
- query, 79, 147
- radix, 41
- raising, 145
- raw
 - string
 - literal, 48
- RDBMS, 135
- RDF
 - graph, 231
 - store, 235
- read, 33
- reference
 - table, 141
- regexp, 49
- regular
 - expression, 49
- relation, 141, 223, 226
- relationship, 226
- relative
 - URI, 79
- release
 - branch, 119, 125
- remote
 - repository, 121
- renewal
 - provision, 209
- repository, 116
 - hosting
 - service, 126
 - management
 - system, 126
 - server, 126
- represented, 226

- resolution, 131
- resource, 230
- Resource Description Framework, 230
- revision, 116
 - control
 - action, 116
 - system, 116
- root, 74
- route, 85, 90
 - filter, 91
 - function, 90
- routing, 85, 90
- row, 135
- RTFM, 17, 38
- rule, 62
- RWD, 71
- responsive
 - web
 - design, 71
- safe, 83
- scheme, 79
- selector, 62
- self-join, 148
- semantic, 229
 - network, 222
 - web, 217
- semantics, 15
- server-side
 - scripting
 - framework, 85
- serverside
 - routing, 90
- share
 - alike, 212
- software, 12
- sound
 - recording, 206
- source, 14
- SQL, 137
 - injection
 - attack, 145
- SQL-sanitizes, 151
- staging, 123
 - branch, 119
- statement, 231
- status, 131
- storage, 12
- stpl
 - python, 89
- stream, 33
- string, 26
 - literal, 48
- structured
 - query
 - language, 137
- subject, 231
- symbolic
 - data, 216
- syntax, 15
- system
 - access
 - control, 213
- table, 135
 - join, 148
 - name, 135
- tag, 57
- TBox, 224, 226
- tebi, 54
- template
 - engine, 86
 - file, 86, 89
 - functions, 89
 - processing, 86
 - processor, 86
- term
 - provision, 209
- terminology, 224, 226
- territory
 - provision, 209
- text
 - editor, 53
 - file, 52
 - node, 73
- textual
 - content, 56
- third party
 - cookies, 93
- three-way
 - merge, 117
- title, 129
- tree, 74
- triple, 231
 - store, 234
- triplestore, 235
- trunk, 119
- Turing
 - complete, 14
- two-way
 - merge, 117
- type, 26
- UCS, 46
- UNA, 227
- unary, 42
 - natural
 - numbers, 40

- unicode
 - Standard, 46
- uniform
 - resource
 - identifier, 79
 - locator, 80
 - name, 80
- unique, 141
 - name
 - assumption, 227
- universal
 - character
 - set, 46
 - tool, 14
- update, 116, 140
- upstream, 121
- URI, 79
 - decoding, 81
 - encoding, 81
- URL, 80
- URN, 80
- user
 - agent, 83
- value, 20, 32, 62, 135
- variable, 24
 - assignment, 24
 - name, 24
- view, 148
- vocabulary, 226
- web
 - application, 77
 - browser, 82
 - IDE, 20
 - page, 78
 - resource, 79
 - server, 83
 - site, 78
- WFH, 208
- width, 65
- WONTFIX, 131
- word
 - processor, 53
- work made for hire, 208
- working
 - copy, 116
- WORKSFORME, 131
- World Wide Web, 78
- write, 33
- WWW, 78
- WWWeb, 78
- document
 - tree, 73
- path
 - language, 76
- yobi, 54
- zebi, 54