Contents

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

Prof. Dr. Michael Kohlhase

Knowledge Representation and -Processing
Computer Science, FAU Erlangen-Nürnberg
https://kwarc.info/kohlhase

2025-06-05

Contents

0.1 Preface

0.1.1 Course Concept

Objective: The course aims at giving students an overview over the variety of digital tools and methods at the disposal of practitioners of the humanities and social sciences, explaining their intuitions on how/why they work (the way they do). The main goal of the course is to empower students for their for the emerging discipline of "digital humanities and social sciences". In contrast to a classical course in computer science which lays the mathematical and computational foundations which will become useful in the long run, we want to introduce methods and tools that can become *useful in the short term* and thus generate immediate success and gratification, thus alleviating the "programming shock" (the brain stops working when in contact with computer science tools or computer scientists) common in the humanities and social sciences.
Original Context: The course "Informatische Werkzeuge in den Geistes- und Sozialwissenschaften" is a first-year, two-semester course in the bachelor program "Digitale Geistes- und Sozialwissenschaften" (Digital Humanities and Social Sciences: DigiHumS) at FAU Erlangen-Nürnberg.
Open to External Students: Other bachelor programs are increasingly co-opting the course as specialization option or a key skill. There is no inherent restriction to DHSS students in this course.
Prerequisites: There are no formal prerequisites – after all it starts in the first semester for DigiHumS – but a good deal of motivation, openness towards exploring the weird and wonderful world of 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.

0.1.2 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 ??? and ???. 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, markupsystems, HTML, and XML; see chapter 4.
2. basic concepts of the World Wide Web; see ???
3. Web technologies for interactive documents and their applications; in particular internet infrastructure, web browsers and servers, PHP, dynamic HTML, JavaScript, and CSS; see ???.

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

1. Databases; in particular entity relationship diagrams, CRUD operations, and querying; see ???.
2. Image processing tools, see ???

3. Using the ontologies and the semantic web for Cultural Heritage; see ???

4. The WissKI System: A Virtual Research Environment for Cultural Heritage; see ???

5. Copyright and Data Privacy as legal foundations of DH tools; see ???

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

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

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

0.1.3 Programming Exercises and JuptyterLab as a Web IDE

Programming Exercises: Most of the computer tools introduced in this course require programming e.g. for configuration, extension, or input preprocessing or work much better when the user understands the basic underlying concepts at the program level. Therefore we accompany the course with a set of (programming) exercises (given as homework to the IWGS students) that allow practicing that.

Web IDEs: In the IWGS course at FAU, which is adressed to students from the humanities and social sciences, we do not have access to a pool of standardized hardware. Students have to use their own computing devices for the programming exercises. In any group with diverse hardware, installing software, standardizing software versions, …becomes a serious problem, even if the group only has 50 members; in IWGS, we need the Python interpreter, a text editor or integrated development environment (IDE), and various Python libraries. In IWGS we solve this by using a web IDE, which only presupposes a web browser on student hardware.

Jupyterlab: After experimenting with commercial web IDEs we settled on jupyterLab, even though it does not focus on IDE features. Jupyter notebooks allow to mix documentation, code snippets, and exercise text of programming exercises and package them into learning objects that can be downloaded, interacted with, and submitted easily. jupyterLab acts as the user interface for managing and editing jupyter notebooks and supplies standardized shell and Python REPLs for students. The jupyterLab server runs as a virtual machine on the instructor's hardware. Resource consumption is minimal in our experience (except in the week before the exam). See [JKI] for a documentation of how to set up a server for a small course like IWGS.

Limitations of JupyterLab: Of course, students who want to engage in more serious software development will eventually have to "graduate" to a regular IDE when programs become larger and more long-lived. But this – and the necessary software engineering skills – is emphatically not the focus of the IWGS course.

Exercise Notebooks: The exercise notebooks (in notebook format and PDF – unfortunately only in German) can be found at `https://kwarc.info/teaching/IWGS/NB`. They comprise

- outright programming exercises that introduce the Python language or allow to play with the respective concepts in Python

- code reading/debugging exercises where the character of Beatrice Beispiel almost solves interesting problems, and

- development steps towards larger applications, which often involve completing Python skeletons using the concepts taught in the lectures.

In all cases, the necessary increments to be supplied by the students are designed to not let the Python skills become a barrier, but give students the opportunity to develop the necessary programming skills in passing.

We have themed the exercises with DigiHumS topics to keep them interesting for our students.

0.1.4 This Document

Presentation: The document mixes the slides presented in class with comments of the instructor to give students a more complete background reference. **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 sTeX package [Koh08; sTeX], 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 lecture 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`.

0.1.5 Acknowledgments

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

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

Teaching Assistants: The organization and material choice in the IWGS has significantly been influenced by Jonas Betzendahl and Philipp Kurth, who have been very active and dedicated teaching assistants and have given feedback on all aspects of the course. They have also provided almost all of the IWGS exercises – see subsection 0.1.3.

DigiHumS Administrators: Jacqueline Klusik-Eckert and Philipp Kurth who used to administrate the DigiHumS major at FAU together have been helpful in navigating the administrative waters of an unfamiliar faculty.

WissKI Specialists and Colleagues: ??? has profited from discussions with Peggy Große and Juliane Hamisch, then two WissKI specialists at FAU. My colleagues Prof. Peter Bell has provided the idea and data for the “Kirmes Pictures Project” that grounds some of the second semester.

JupyterLab: The JupyterLab server at `https://juptyter.kwarc.info` (see ???) has been developed, operated, and maintained by Jonas Betzendahl. For details see [JKI].

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

0.2 Recorded Syllabus

The recorded syllabus – a record the progress of the course in the 1 – is in the course page in the ALeA system at `https://courses.voll-ki.fau.de/course-home/iwgs-1`. The table of contents in the IWGS lecture notes at `https://kwarc.info/teaching/IWGS` indicates the material covered to date in yellow.

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

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

FAU : 1 2025-06-05

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

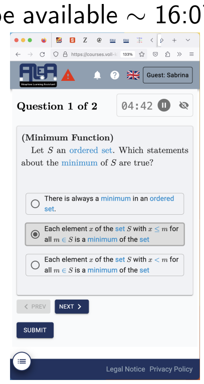
- **Grading Background/Theory:** Only modules are graded! (by the law)
  - Module "DH-Einführung" (DHE) $\widehat{=}$ courses IWGS1/2, DH-Einführung. (7.5 ECTS)
  - DHE module grade $\leadsto$ pass/fail determined by "portfolio" $\widehat{=}$ collection of contributions/assessments.
  - Module "DH-Einführung mit Übungen" (DHÜ) $\widehat{=}$ courses IWGS1/2, (10 ECTS)
  - DHÜ module grade $\leadsto$ 1-5 50% exam, 50% homework assignments, 10% bonus points from prepquizzes.
- **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. 2025.

- **Retake Exam:** 60 min exam at the end of the exam break. (∼ May 4. 2025)
- **To help you succeed:** We offer you
  - **External motivation**: informal points for homeworks and a grade for exam, (even though only pass/fail relevant in the end)
  - weekly online prepquizzes that help you prepare for the course. (check understanding/preparation)

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 IWGS concepts. We will go into the details next.

Preparedness Quizzes

- **PrepQuizzes:** Before every lecture we offer a 10 min online quiz – the PrepQuiz – about the material from the previous week. (∼ 16:0?-16:15 (check on ALeA); starts in week 2)
- **Motivations:** We do this to
  - keep you prepared and working continuously. (primary)
  - bonus points if the exam has $\geq 50\%$ points (potential part of your grade)
  - update the ALeA learner model. (fringe benefit)
- The prepquizes will be given in the ALeA system
  - `https://courses.voll-ki.fau.de/quiz-dash/iwgs-1`
  - You have to be logged into ALeA! (via FAU IDM)
  - You can take the prepquiz on your laptop or phone, . . .
  - . . . in the lecture or at home . . .
  - . . . via WLAN or 4G Network. (do not overload)
  - Prepquizzes will only be available ∼ 16:0?-16:15 (check on ALeA)!



1.2 Getting Most out of IWGS

In this section we will discuss a couple of measures that students may want to consider to get most out of the IWGS course.

None of the things discussed in this section – homeworks, tutorials, study groups, and attendance – are mandatory (we cannot force you to do them; we offer them to you as learning opportunities), but most of them are very clearly correlated with success (i.e. passing the exam and getting a good grade), so taking advantage of them may be in your own interest.

IWGS Homework Assignments

- ▷ **Goal:** Homework assignments reinforce what was taught in lectures.
- ▷ **Homework Assignments:** Small individual problem/programming/proof task
  - ▷ but take time to solve (at least read them directly ⇝ questions)
- ▷ **Didactic Intuition:** Homework assignments give you material to test your understanding and show you how to apply it.
- ▷ ⚠ Homeworks give no points, but without trying you are unlikely to pass the exam.
- ▷ **Homework Workflow:** in ALeA (see below)
  - ▷ Homework assignments will be published on thursdays: see `https://courses.voll-ki.fau.de/hw/iwgs-1`
  - ▷ Go to the Tutorials to discuss them.
  - ▷ Submission of solutions via the StudOn system in the week after
  - ▷ graded by the TA.
- ▷ **Homework/Tutorial Discipline:**
  - ▷ Start early! (many assignments need more than one evening's work)
  - ▷ Don't start by sitting at a blank screen (talking & study groups help)
  - ▷ Humans will be trying to understand the text/code/math when grading it.
  - ▷ Go to the tutorials, discuss with your TA! (they are there for you!)

FAU : 4 2025-06-05

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 very little 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 lectures, 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)

▷ **Tutor:** (Master Student in CS)

▷ Dirk Böhme: `dirk.boehme@fau.de`

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

▷ Dirk will also grade the homework assignments for the DFÜ students. (grade-relevant)

▷ **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 lecture notes and the homework assignments

▷ **Inverted Classroom:** the latest craze in didactics (works well if done right)

in IWGS: lecture + homework assignments + tutorials $\widehat{=}$ inverted classroom

Collaboration

▷ **Definition 1.2.1.** Collaboration (or cooperation) is the process of groups of agents acting together for common, mutual benefit, as opposed to acting in competition for selfish benefit. In a collaboration, every agent contributes to the common goal and benefits from the contributions of others.

▷ In learning situations, the benefit is "better learning".

▷ **Observation:** In collaborative learning, the overall result can be significantly better than in competitive learning.

▷ **Good Practice:** Form study groups. (long- or short-term)

1. ⚠ Those learners who work/help most, learn most!
2. ⚠ Freeloaders – individuals who only watch – learn very little!

▷ It is OK to collaborate on homework assignments in IWGS! (no bonus points)

▷ Choose your study group well! (ALeA helps via the study buddy feature)

As we said above, almost all of the components of the IWGS course are optional. That even applies to attendance. But make no mistake, attendance is important to most of you. Let me explain, . . .

Do I need to attend the IWGS Lectures

▷ Attendance is not mandatory for the IWGS course. (official version)

▷ **Note:** There are two ways of learning: (both are OK, your mileage may vary)

- ▹ Approach B: Read a book/papers (here: lecture notes)
- ▹ Approach I: come to the lectures, be involved, interrupt the instructor whenever you have a question.

The only advantage of I over B is that books/papers do not answer questions

▷ Approach S: come to the lectures and sleep does not work!

▷ The closer you get to research, the more we need to discuss!

1.3 Learning Resources for IWGS

Course Notes, Forum, Matrix

▷ **Lecture notes** will be posted at `https://kwarc.info/teaching/IWGS`

- ▹ We mostly prepare/update them as we go along (semantically preloaded ⇝ research resource)
- ▹ Please report any errors/shortcomings you notice. (improve for the group/successors)

▷ **StudOn Forum:** For announcements – `https://www.studon.fau.de/studon/goto.php?target=lcode_3oqqBg7g`

▷ **Matrix Channel:** `https://matrix.to/#/#iwgs:fau.de` for questions, discussion with instructors and among your fellow students. (your channel, use it!)

Login via FAU IDM ⇝ instructions

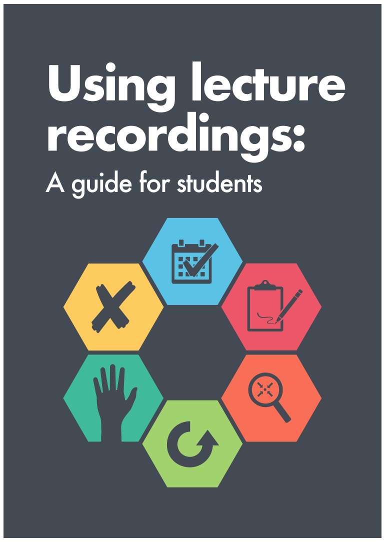
▷ **Course Videos** are at at `https://www.fau.tv/course/id/4020`.

▷ **Do not let the videos mislead you:** Coming to class is highly correlated with passing the exam!

FAU has issued a very insightful guide on using lecture videos. It is a good idea to heed these recommendations, even if they seem annoying at first.

Practical recommendations on Lecture Videos

▷ **Excellent Guide:** [Nor+18a] (German version at [Nor+18b])



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

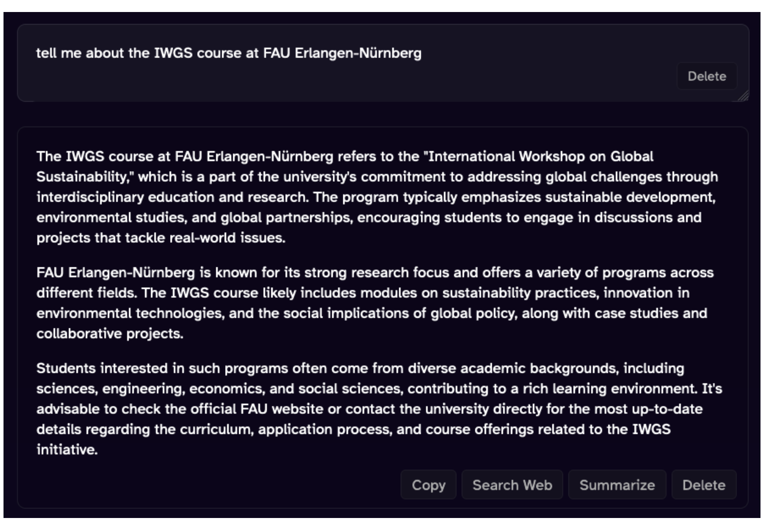
NOT a Resource for : LLMs – AI-tools like ChatGPT

▷ **Definition 1.3.1.** A large language model (LLM) is a computational model capable of language generation or other natural language processing tasks.

▷ **Example 1.3.2.** OpenAI's GPT, Google's Bard, and Meta's Llama.

▷ **Definition 1.3.3.** A chatbot is a software application or web interface that is designed to mimic human conversation through text or voice interactions. Modern chatbots are usually based on LLMs.

▷ **Example 1.3.4 (ChatGPT talks about IWGS).** (Aha, where does this come from?)



▷ **Example 1.3.5 (In the IWGS exam).** ChatGPT scores almost perfectly (master solution quality)

- ▷ ChatGPT can pass the exam … (We could award it a Master's degree)
- ▷ But can you? (the IWGS exams will be in person on paper)

You will only pass the exam, if you can do IWGS yourself!

▷ **Intuition:** AI tools like GhatGPT, CoPilot, etc. (see also [She24])

- ▷ can help you solve problems, (valuable tools in production situations)
- ▷ hinders learning if used for homeworks/quizzes, etc. (like driving instead of jogging)

▷ **What (not) to do:** (to get most of the brave new AI-supported world)

- ▷ try out these tools to get a first-hand intuition what they can/cannot do
- ▷ challenge yourself while learning so that you can also do it (mind over matter!)

FAU : 10 2025-06-05

1.4 Goals, Culture, & Outline of the Course

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/tools that can become *useful in the short term*

- ▷ generate immediate success and gratification, (important for motivation)
- ▷ 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.

FAU : 11 2025-06-05

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 1.4.1.** The academic culture is the overall style of working, research, and discussion in an academic field.

- **Observation 1.4.2.** *There are significant differences in the academic culture between computer science, the humanities and the 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 an important concern
- Almost all technical literature is in English. (technical vocabulary too)
- CSlings love shallow hierarchies. (no personality cult; alle per Du)

FAU : 12 2025-06-05

Please keep in mind that – self-awareness is always difficult – the list above may be incomplete and clouded by mirror-gazing. We now come to the concrete topics we want to cover in IWGS. 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 wordsstring, character encodings, unicode, and regular expressions
- Electronic documents and document processing:
  - text files
  - markup systems, HTML, and CSS
  - XML: Documents are trees.
- Web technologies for interactive documents and web applications
  - internet infrastructure: web browsers and server
  - server-side computation: bottle routing and
  - client-side interaction: dynamic HTML, JavaScript, HTML forms
- Web application project (fill in the blanks to obtain a working web app)

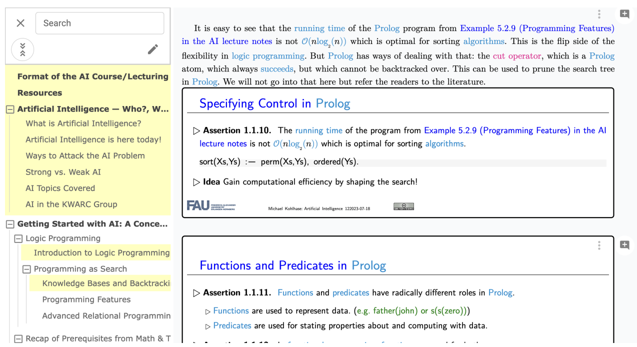
FAU : 13 2025-06-05

1.5 ALeA – AI-Supported Learning

In this section we introduce the ALeA (Adaptive Learning Assistant) system, a learning support system we will use to support students in IWGS.

ALeA: Adaptive Learning Assistant

- ▷ **Idea:** Use AI methods to help teach/learn AI (AI4AI)
- ▷ **Concretely:** Provide HTML versions of the IWGS slides/lecture notes and embed learning support services into them. (for pre/postparation of lectures)
- ▷ **Definition 1.5.1.** Call a document active, iff it is interactive and adapts to specific information needs of the readers. (lecture notes on steroids)
- ▷ **Intuition:** ALeA serves active course materials. (PDF mostly inactive)
- ▷ **Goal:** Make ALeA more like a instructor + study group than like a book!
- ▷ **Example 1.5.2 (Course Notes).** $\widehat{=}$ Slides + Comments



⇝ yellow parts in table of contents (left) already covered in lectures.

FAU : 14 2025-06-05

The central idea in the AI4AI approach – using AI to support learning AI – and thus the ALeA system is that we want to make course materials – i.e. what we give to students for preparing and postparing lectures – more like teachers and study groups (only available 24/7) than like static books.

VoLL-KI Portal at `https://courses.voll-ki.fau.de`

- ▷ **Portal for ALeA Courses:** `https://courses.voll-ki.fau.de`

- ▷ **IWGS in ALeA:** `https://courses.voll-ki.fau.de/course-home/iwgs-1`

- ▹ All details for the course.
- ▹ recorded syllabus (keep track of material covered in course)
- ▹ syllabus of the last semesters (for over/preview)

▷ **ALeA Status:** The ALEA system is deployed at FAU for over 1000 students taking eight courses

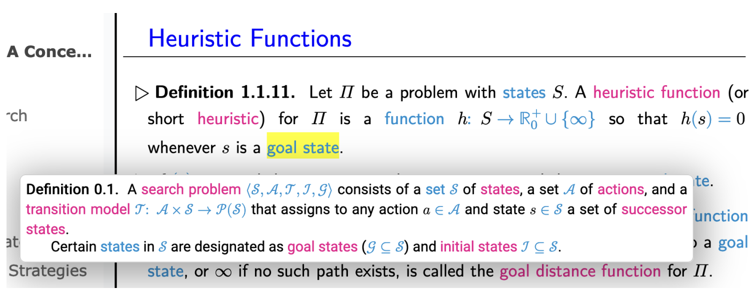
- ▹ (some) students use the system actively (our logs tell us)
- ▹ reviews are mostly positive/enthusiastic (error reports pour in)

The ALEA IWGS page is the central entry point for working with the ALeA system. You can get to all the components of the system, including two presentations of the course contents (notes- and slides-centric ones), the flashcards, the localized forum, and the quiz dashboard.
We now come to the heart of the ALeA system: its learning support services, which we will now briefly introduce. Note that this presentation is not really sufficient to undertstand what you may be getting out of them, you will have to try them, and interact with them sufficiently that the learner model can get a good estimate of your competencies to adapt the results to you.

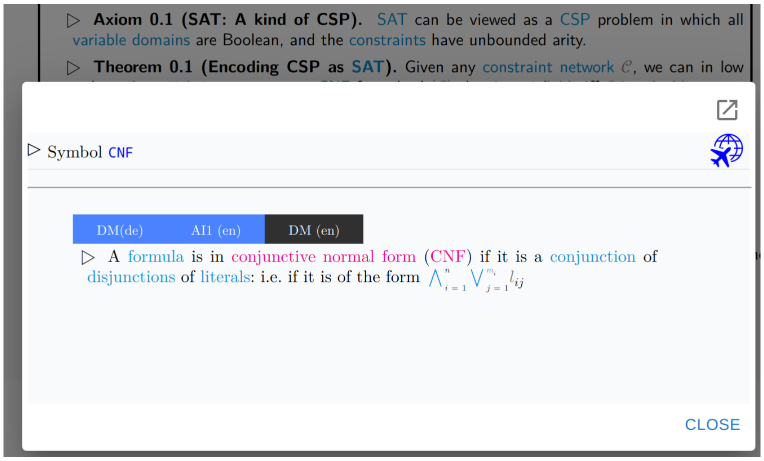
Learning Support Services in ALEA

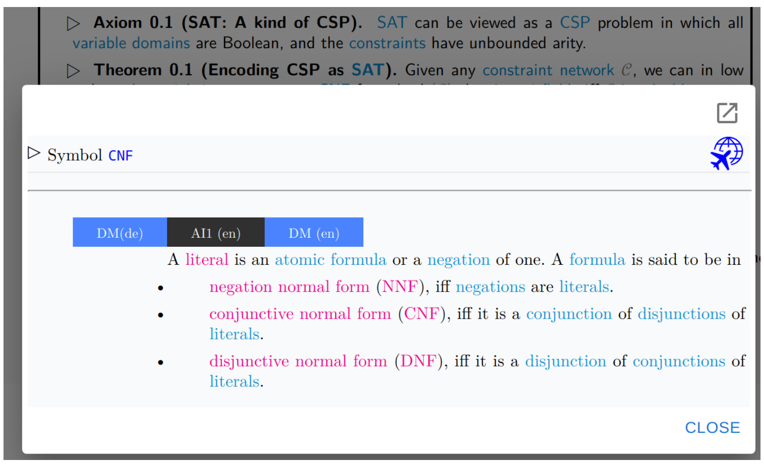
▷ **Idea:** Embed learning support services into active course materials.

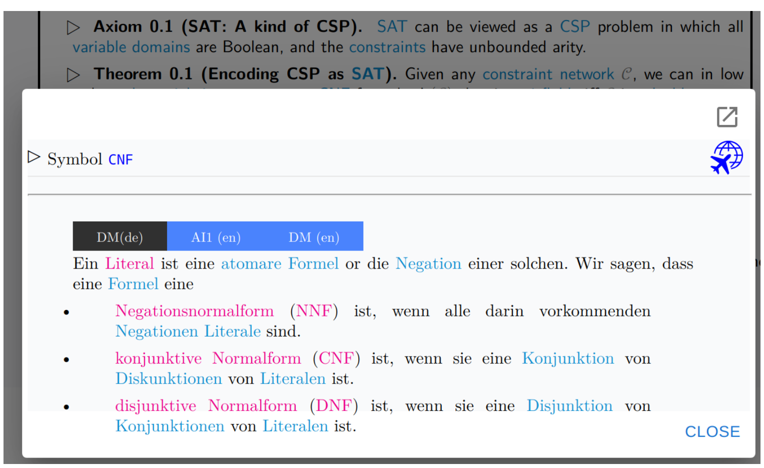
▷ **Example 1.5.3 (Definition on Hover).** Hovering on a (cyan) term reference reminds us of its definition. (even works recursively)



▷ **Example 1.5.4 (More Definitions on Click).** Clicking on a (cyan) term reference shows us more definitions from other contexts.







▷ **Example 1.5.5 (Guided Tour).** A guided tour for a concept c assembles definitions/etc. into a self-cont

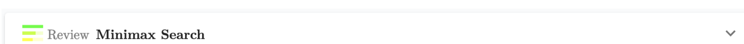
c = countable ⇝

▷ . . . your idea here . . . (the sky is the limit)

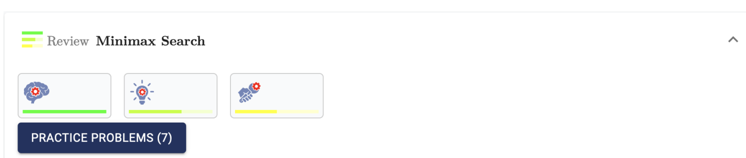
Note that this is only an initial collection of learning support services, we are constantly working on additional ones. Look out for feature notifications () on the upper right hand of the ALeA screen.

(Practice/Remedial) Problems Everywhere

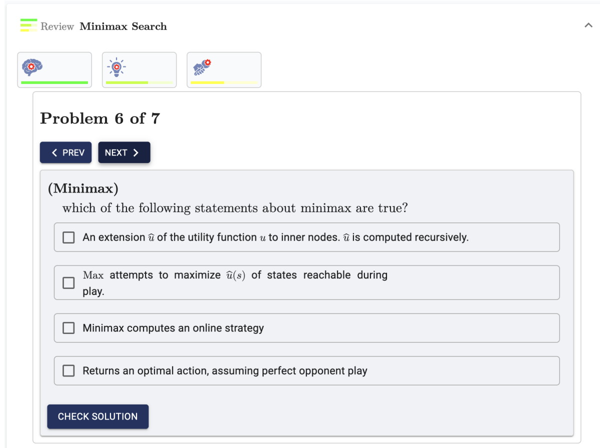
- ▷ **Problem:** Learning requires a mix of understanding and test-driven practice.
- ▷ **Idea:** ALeA supplies targeted practice problems everywhere.
- ▷ **Concretely:** Revision markers at the end of sections.
  - ▷ A relatively non-intrusive overview over competency



  - ▷ Click to extend it for details.



  - ▷ Practice problems as usual. (targeted to your specific competency)



: 17 2025-06-05

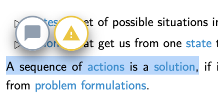
While the learning support services up to now have been adressed to individual learners, we now turn to services addressed to communities of learners, ranging from study groups with three learners, to whole courses, and even – eventually – all the alumni of a course, if they have not de-registered from ALeA.

Currently, the community aspect of ALeA only consists in localized interactions with the course materials.
The ALeA system uses the semantic structure of the course materials to localize some interactions that are otherwise often from separate applications. Here we see two:

1. one for reporting content errors – and thus making the material better for all learners – and‘’
2. a localized course forum, where forum threads can be attached to learning objects.

Localized Interactions with the Community

▷ Selecting text brings up localized – i.e. anchored on the selection – interactions:



▷ post a (public) comment or take (private) note

▷ report an error to the course authors/instructors

▷ Localized comments induce a thread in the ALeA forum (like the StudOn Forum, but targeted towards specific learning objects.)

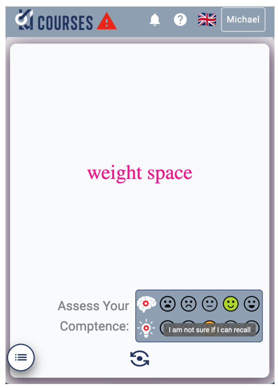
- ▷ Answering questions gives karma $\widehat{=}$ a public measure of user helpfulness.
- ▷ Notes can be anonymous (⇝ generate no karma)

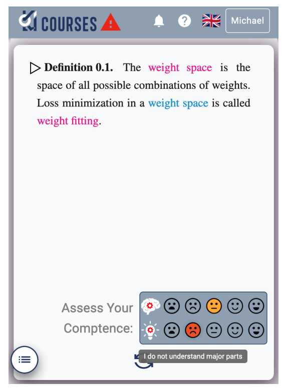
FAU : 18 2025-06-05

We can use the same four models discussed in the space of guided tours to deploy additional learning support services, which we now discuss.

New Feature: Drilling with Flashcards

- ▷ Flashcards challenge you with a task (term/problem) on the front. . .





. . . and the definition/answer is on the back.

- ▷ Self-assessment updates the learner model (before/after)
- ▷ **Idea:** Challenge yourself to a card stack, keep drilling/assessing flashcards until the learner model eliminates all.
- ▷ **Bonus:** Flashcards can be generated from existing semantic markup (educational equivalent to free beer)

FAU : 19 2025-06-05

We have already seen above how the learner model can drive the drilling with flashcards. It can also be used for the configuration of card stacks by configuring a domain e.g. a section in the course materials and a competency threshold. We now come to a very important issue that we always face when we do AI systems that interface with humans. Most web technology companies that take one the approach "the user pays for the services with their personal data, which is sold on" or integrate advertising for renumeration. Both are not acceptable in university setting.

But abstaining from monetizing personal data still leaves the problem how to protect it from intentional or accidental misuse. Even though the GDPR has quite extensive exceptions for research, the ALeA system – a research prototype – adheres to the principles and mandates of the GDPR. In particular it makes sure that personal data of the learners is only used in learning support services directly or indirectly initiated by the learners themselves.

Learner Data and Privacy in ALeA

- **Observation:** Learning support services in ALEA use the learner model; they
  - need the learner model data to adapt to the invidivual learner!
  - collect learner interaction data (to update the learner model)
- **Consequence:** You need to be logged in (via your FAU IDM credentials) for useful learning support services!
- **Problem:** Learner model data is highly sensitive personal data!
- **ALeA Promise:** The ALEA team does the utmost to keep your personal data safe. (SSO via FAU IDM/eduGAIN, ALEA trust zone)
- **ALeA Privacy Axioms:**
  1. ALEA only collects learner models data about logged in users.
  2. Personally identifiable learner model data is only accessible to its subject (delegation possible)
  3. Learners can always query the learner model about its data.
  4. All learner model data can be purged without negative consequences (except usability deterioration)
  5. Logging into ALEA is completely optional.
- **Observation:** Authentication for bonus quizzes are somewhat less optional, but you can always purge the learner model later.

So, now that you have an overview over what the ALEA system can do for you, let us see what you have to concretely do to be able to use it.

Concrete Todos for ALeA

- **Recall:** You will use ALeA for the prepquizzes (or lose bonus points)
  All other use is optional. (but AI-supported pre/postparation can be helpful)
- To use the ALeA system, you will have to log in via SSO: (do it now)
  - go to `https://courses.voll-ki.fau.de/course-home/iwgs-1`,
  - in the upper right hand corner you see 
,
  - log in via your FAU IDM credentials. (you should have them by now)
  - You get access to your personal ALeA profile via 

    (plus feature notifications, manual, and language chooser)
- **Problem:** Most ALeA services depend on the learner model. (to adapt to you)
- **Solution:** Initialize your learner model with your educational history!
  - **Concretely**: enter taken CS courses (FAU equivalents) and grades.
  - ALeA uses that to estimate your CS/AI competencies. (for your benefit)
  - then ALeA knows about you; I don't! (ALeA trust zone)

Even if you did not understand some of the AI jargon or the underlying methods (yet), you should be good to go for using the ALeA system in your day-to-day work.

Chapter 2

Introduction to Programming

2.1 What is 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 computers are universal machines. Unlike a conventional tool e.g a spade – which has a limited number of purposes/behaviors – digging holes in case of a spade, maybe hitting someone over the head, a computer can be given arbitrary[1] 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 image, audio, and video,

iii) transporting information for communication,

iv) representing knowledge and reasoning,

v) transforming, optimizing, and verifying other programs,

vi) learning patterns in data and predicting the future from the past.

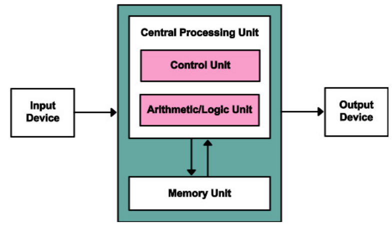
Computer Hardware/Software & Programming

▷ **Definition 2.1.1.** Computers consist of hardware and software.

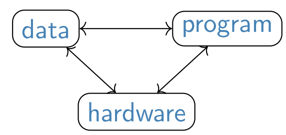
▷ **Definition 2.1.2.** Hardware consists of

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

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



- **Definition 2.1.3.** Software consists of
  - data that represents objects and their relationships in the world
  - programs that inputs, manipulates, outputs data



- **Remark:** Hardware stores data and runs programs.

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

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

With these components we can build various kinds of universal machines; these range from thought experiments like Turing machines, to today's general purpose computers like your laptop with various embedded systems (wristwatches, Internet routers, airbag controllers, ...) in-between.
Note that – given enough fantasy – the human brain has the same components. Indeed the human mind is a universal machine – we can think whatever we want, react to the environment, and are not limited to particular behaviors. There is a sub-field of computer science that studies this: AI (artificial intelligence). 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.

Programming Languages

- Programming $\widehat{=}$ writing programs (Telling the computer what to do)
- *Remark 2.1.4.* The computer does exactly as told
  - extremely fast extremely reliable
  - completely stupid: will not do what you mean unless you tell it exactly

[2] Marvin Minsky; one of the founding fathers of the field of artificial intelligence

▷ Programming can be extremely fun/frustrating/addictive (try it)

▷ **Definition 2.1.5.** 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 programming paradigms: imperative, functional, logic, object oriented programming.

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

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

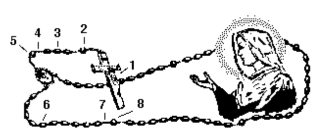
- the behavior of programs is – in principle – predictable
- all errors of program behavior are your own (the programmer's)

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

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

Program Execution

▷ **Definition 2.1.6.** Algorithm: informal description of what to do (good enough for humans)

▷ **Example 2.1.7.**  

▷ **Example 2.1.8.** Program: computer processable version, e.g. in Python.

```
for x in range(0, 3):
```

```
    print ("we tell you",x,"time(s)")
```

▷ **Definition 2.1.9.** Interpreter: reads a program and executes it directly

- ▹ special case: interactive interpretation (lets you experiment easily)

▷ **Definition 2.1.10.** Compiler: translates a program (the source) into another program (the binary) in a much simpler programming language for optimized execution on hardware directly.

▷ *Remark 2.1.11.* Compilers are efficient, but more cumbersome for development.

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

2.2 Programming in IWGS

After the general introduction to programming in ???, 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 ($\widehat{=}$ "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

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

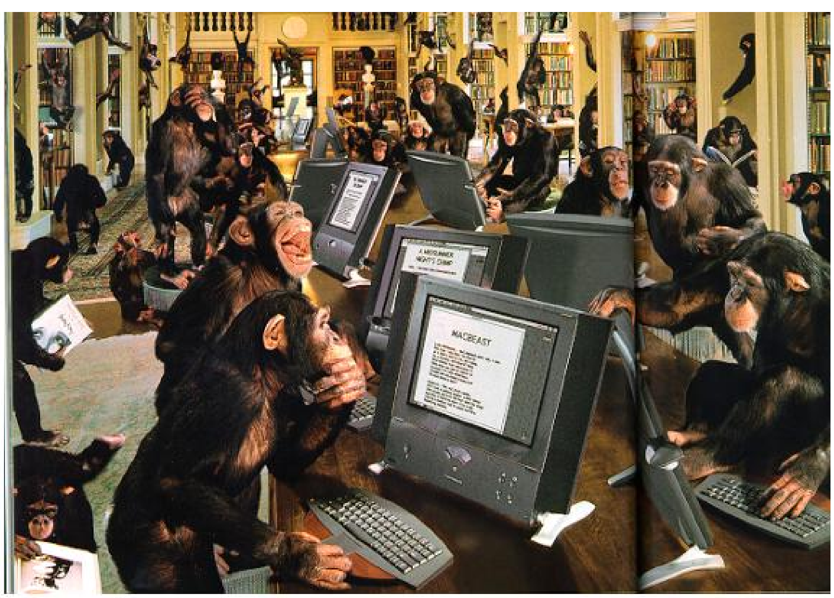
But Seriously. . . Learning programming in IWGS

- ▷ The IWGS course 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

FAU : 26 2025-06-05

However, the result would probably be the following:

⚠ 2am in the Kollegienhaus CIP Pool ⚠



FAU : 27 2025-06-05

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 course is for)
  - read Richard Stallman's quote carefully ⇝ problem understanding is a crucial prerequisite for hacking.
- "*The GIGO Principle: Garbage In, Garbage Out*" (– ca. 1967)
- "*Applets, Not Craplets*^{tm}" (– ca. 1997)

FAU : 28 2025-06-05

2.3 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.

2.3.1 Hello IWGS

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

Python in a Nutshell

- **Why Python?:**
  - general purpose programming language
  - imperative, interactive interpreter
  - syntax very easy to learn (spend more time on problem solving)
  - scales well:
    - easy for beginners to write simple programs,
    - but advanced software can be written with it as well.
- **Interactive mode:** The Python shell IDLE3
- **For the eager (optional):**

  Establish a Python interpreter (version 3.7) (not 2.?.?, that has different syntax)
  - install Python from `http://python.org` (for offline use)
  - make sure (tick box) that the python executable is added to the path. (makes shell interaction much easier)

FAU : 29 2025-06-05

Installing Python: Python can be installed from `http://python.org` ⇝ "Downloads", as a `MSWindows` installer or a `macOS` disk image. For `linux` it is best installed via the package manager, e.g. using

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

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

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

```
python ⟨⟨filename⟩⟩
```

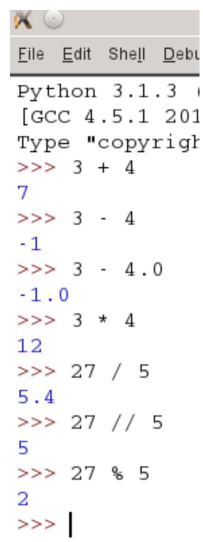
to run the python file ⟨⟨filename⟩⟩. This is better than using the windows-specific

```
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)
  - 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 _)
  - **Definition 2.3.1.** Expressions are created from values (and other expressions) via operators.
  - **Observation:** The Python interpreter simplifies expressions to values by computation.



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 well be one of the "others" yourself, (in a year's time)
  - writing comments first helps you organize your thoughts.

[1] EdNote: fully introduce the concept of a shell in the next round

- 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.

FAU : 31 2025-06-05

2.3.2 JupyterLab, a Python Web IDE for IWGS

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

jupyterLab A Cloud IDE for Python

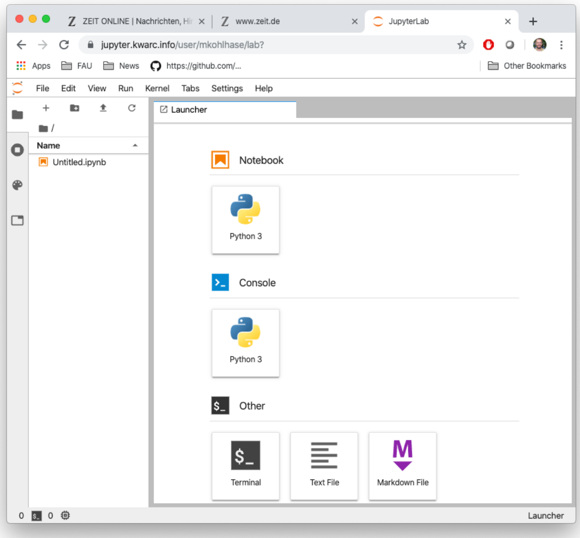
- **For helping you** it would be good if the TAs could access to your code
- **Idea:** Use a web IDE (a web based integrated development environment): jupyterLab, which you can use for interacting with the interpreter.
- We will use jupyterLab for IWGS. (but you can also use Python locally)
- **Homework:**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

FAU : 32 2025-06-05

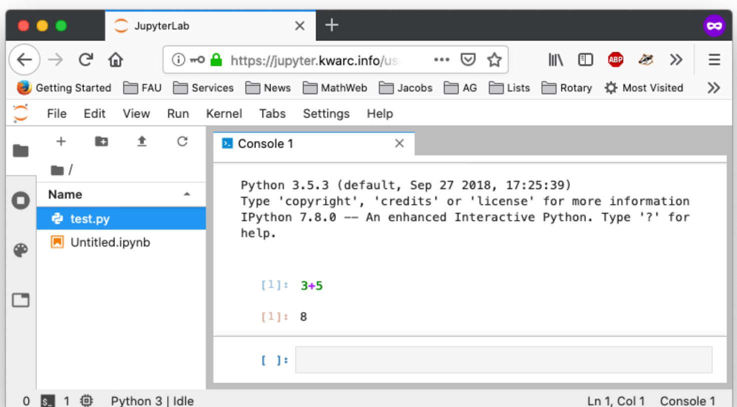
The advantage of a cloud IDE like jupyterLab for a course like IWGS is that you do not need any installation, cannot lose your files, and your teachers (the course instructor and the teaching assistants) can see (and even directly interact with) the your run time environment. This gives us a much more controlled setting and we can help you better.
Both IDLE3 as well as jupyterLab come with an integrated editor for writing Python programs. These editors gives you Python syntax highlighting, and interpreter and debugger integration. In short, IDLE3 and jupyterLab are integrated development environments for Python. Let us now go through the interface of the jupyterLab IDE.

jupyterLab Components

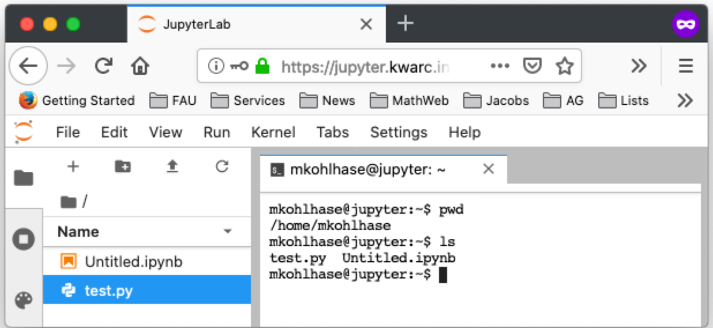
- **Definition 2.3.2.** The jupyterLab dashboard gives you access to all components.



▷ **Definition 2.3.3.** The jupyterLab python console, i.e. a Python interpreter in your browser. (use this for Python interaction and testing.)



▷ **Definition 2.3.4.** The jupyterLab terminal, i.e. a `UNIX` shell in your browser. (use this for managing files)



▷ **Definition 2.3.5.** A shell is a command line interface for accessing the services of a computer's operating system.

There are multiple shell implementations: sh, csh, bash, zsh; they differ in advanced features.

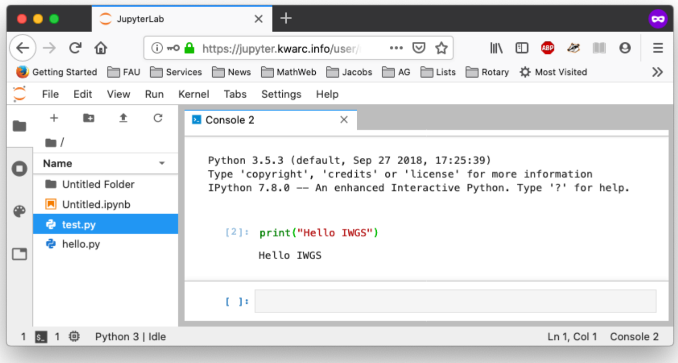
▷ **Useful shell commands:** See e.g. [All18] for a basic tutorial

- ▷ ls: "list" the files in this directory
- ▷ mkdir: "make" folder (called "directory")
- ▷ pwd: "print working directory" (where am I)
- ▷ cd ⟨⟨dirname⟩⟩: "change directory"
  - ▷ if ⟨⟨dirname⟩⟩ = ..: one up in the directory tree
  - ▷ empty ⟨⟨dirname⟩⟩: go to your home directory.
- ▷ rm ⟨⟨name⟩⟩: remove file/directory
- ▷ cp/mv ⟨⟨filename⟩⟩ ⟨⟨newname⟩⟩: copy to or rename
- ▷ cp/mv ⟨⟨filename⟩⟩ ⟨⟨dirname⟩⟩: copy or move to
- ▷ . . . see [All18] for more . . .

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

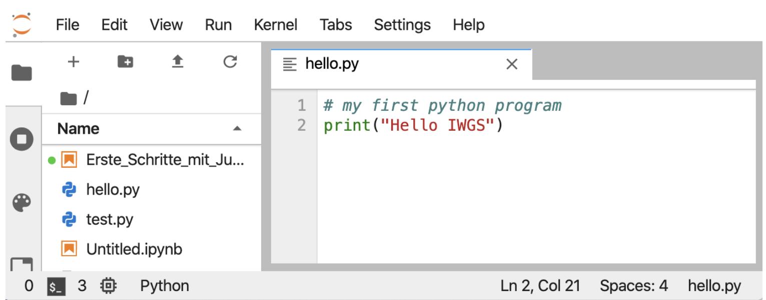
A first program in Python

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



▷ **Alternatively:**

1. got to the jupyterLab dashboard select "Text File",
2. Type your program,



3. Save the file as hello.py
4. Go to your terminal and type python3 hello.py

3' **Alternatively**: go to your python console and type (in the same directory)

```
import hello
```


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

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

   ```
   python ⟨⟨filename⟩⟩
   ```

   which the terminal interprets as "call the program python3 on me".

3. Finally, we make the file hello executable, i.e. tell the terminal the file should behave like a shell command by issuing

   ```
   chmod u+x booksapp
   ```

   in the directory where the file hello is stored.

4. We add the line

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

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

With this simple recipe we could in principle extend the repertoire of instructions of the terminal and automate repetitive tasks.
We now come to the signature component of jupyterLab: jupyter notebooks. They take the important practice of documenting code to a whole new level. Instead of just allowing comments in program files, they provide rich text cells, in which we can write elaborate text.

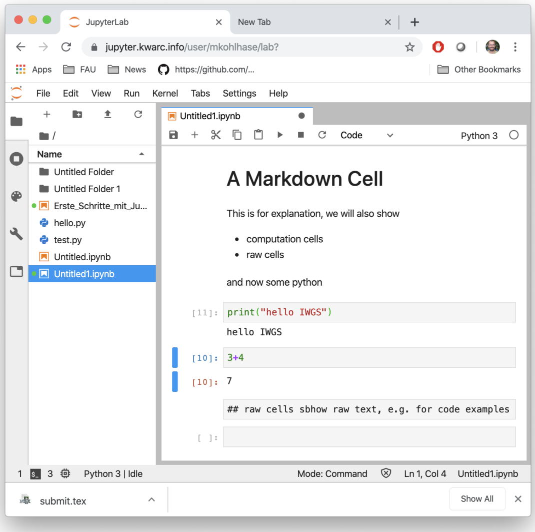
Jupyter Notebooks

- ▷ **Definition 2.3.6.** Jupyter notebooks are documents that combine live runnable code with rich, narrative text (for comments and explanations).
- ▷ **Definition 2.3.7.** Jupyter notebooks consist of cells which come in three forms:
  - ▷ a raw cell shows text as is,
  - ▷ a markdown cell interprets the contents as markdown text, (later more)
  - ▷ a code cell interprets the contents as (e.g. Python) code.
- ▷ Cells can be executed by pressing "shift enter". (Just "enter" gives a new line)
- ▷ **Idea:** Jupyter notebooks act as a REPL, just as IDLE3, but allows
  - ▷ documentation in raw and markdown cells and
  - ▷ changing and re-executing existing cells.

FAU : 35 2025-06-05

Jupyter Notebooks

- ▷ **Example 2.3.8 (Showing off Cells in a Notebook).**



Markdown a simple Markup Format Generating HTML

.

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

▷ **Definition 2.3.9.** Markdown is a family of markup formats whose control words are unobtrusive and easy to write in a text editor. It is intended to be converted to HTML and other formats for display.

▷ **Example 2.3.10.** Markdown is used in applications that want to make user input easy and efficient, e.g. wikis and issue tracking systems.

▷ **Workflow:** Users write markdown, which is formatted to HTML and then served for display.

▷ A good cheet-sheet for markdown control words can be found at `https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet`.

2.3.3 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 2.3.11.** A variable is an identifier (the variable name) that references a memory location which contains a value. Storing the value v into the memory of a variable x is called assigning v to x.

A variable must be initialized (assigned an initial value) before it can be (usefully) referenced.

▷ **Note:** In Python a variable name

- ▹ must start with letter or _,
- ▹ cannot be a Python keyword
- ▹ is case-sensitive (foobar, FooBar, and fooBar are different variables)

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

▷ **Definition 2.3.12 (in Python).** A variable assignment ⟨⟨var⟩⟩=⟨⟨val⟩⟩ assigns a new value to a variable.

▷ **Example 2.3.13 (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 intermeditate results.

Variables in Python: Extended Example

▷ **Example 2.3.14 (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 2.3.15 (Variables for Storing Intermediate Variables).**

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

```
>>> z
'OhGottOhGottOhGottOhGottOhGottOhGottOhGottOhGottOhGott'
```

FAU : 39 2025-06-05

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 2.3.15; 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 variable into expressions.
- ▷ Data types group data and tell the interpreter what to expect
  - ▷ 1, 2, 3, etc. are data of type "integer"
  - ▷ "hello" is data of type "string"
- ▷ Data types determine which operators can be applied
- ▷ In Python, every values has a type, variables can have any type, but can only be assigned values of their type.
- ▷ **Definition 2.3.16.** Python has the following five basic types

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

- ▷ We will ecounter more types later.

FAU : 40 2025-06-05

We will now see what we can – and cannot – do with data types, this becomes most noticable 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 2.3.17.** *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 2.3.18 (Type Errors and Type Coersion).**

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


2.3.4 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 2.3.13 is a perfect example. We will now change that by introducing control structures, i.e complex program instructions that change the control flow of the program.

Conditionals and Loops

▷ **Problem:** Up to now programs seem to execute all the instructions in sequence, from the first to the last. (a linear program)

▷ **Definition 2.3.19.** The control flow of a program is the sequence of execution of the program instructions. It is specified via special program instructions called control structures.

▷ **Definition 2.3.20.** Conditional execution (also called branching) allows to execute (or not to execute) certain parts of a program (the branches) depending on a condition. We call a code block that enables conditional execution a conditional statement or conditional.

▷ **Definition 2.3.21.** A condition is a Boolean expression in a control structure.

▷ **Definition 2.3.22.** A loop is a control structure that allows to execute certain parts of a program (the body) multiple times depending on the value of its conditions. A break instruction terminate the loop irrespective of the value of the condition.

▷ **Example 2.3.23.** In Python, conditions are constructed by applying a Boolean operator to arguments, e.g. 3>5, x==3, x!=3, ...

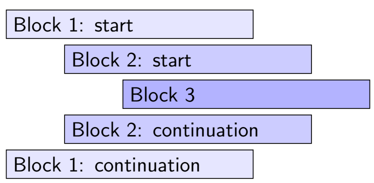
or by combining simpler conditions by Boolean connectives or, and, and not (using brakets if necessary), e.g. x>5 or x<3

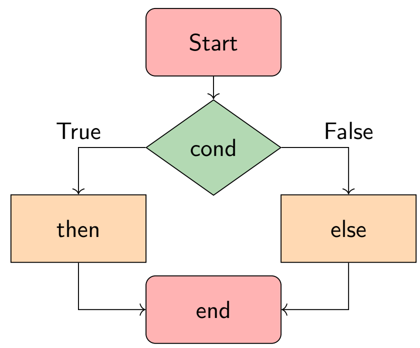
After this general introduction – conditional execution and loops are supported by all programming languages in some form – we will see how this is realized in Python

Conditionals in Python

▷ **Definition 2.3.24.** Conditional execution via **if/else** statements

```
if ⟪condition⟫ :
    ⟪then-part⟫
else :
    ⟪else-part⟫
⟪more code⟫
```





▷ ⟪then-part⟫ and ⟪else-part⟫ have to be indented equally. (e.g. 4 blanks)

▷ If control structures are nested they need to be further indented consistently.

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

Conditional Execution Example

▷ **Example 2.3.25 (Empathy in Python).**

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

Note the indenting of the body parts.

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

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

Variant: Multiple Branches

▷ Making multiple branches is similar

```
if ⟪condition⟫ :
    ⟪then-part⟫
elif ⟪condition⟫ :
    ⟪other then-part⟫
else :
    ⟪else-part⟫
```

- ▷ The there can be more than one **elif** clause.
- ▷ The ⟪condition⟫s are evaluated from top to bottom and the ⟪then-part⟫ of the first one that comes out true is executed. Then the whole control structure is exited.
- ▷ multiple branches could achieved by nested **if**/**else** structures.

▷ **Example 2.3.26 (Better Empathy in Python).** In Example 2.3.25 we print Good! even if the input is e.g. I feel terrible, so extend **if**/**else** by

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


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

```
if ⟪condition⟫ :
    ⟪then-part⟫
    if ⟪condition⟫ :
        ⟪other then-part⟫
    else :
        ⟪else-part⟫
```

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

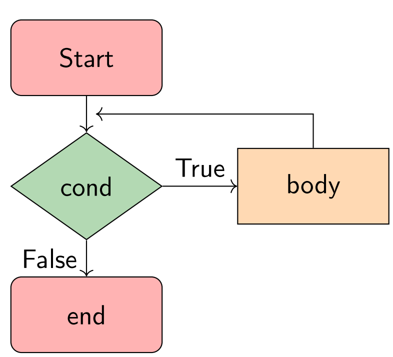
Loops in Python

▷ **Definition 2.3.27.** Python makes loops via **while** blocks

- ▷ syntax of the **while** loop

```
while ⟪condition⟫ :
    ⟪body⟫
⟪more code⟫
```

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



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

Examples of Loops

▷ **Example 2.3.28 (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 2.3.29 (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 2.3.28 and Example 2.3.29 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, sind 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 was we passed 0, then the whole product must be zero. A more efficient implementation is on the right which breaks after seeing the first 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
```

Examples of Loops

▷ **Example 2.3.30 (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)
```


2.4 Some Thoughts about Computers and Programs

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

Computers as Universal Machines (a taste of theoretical 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 2.4.1 (Deep Fundamental Result).* There are things no computer can compute.
- ▷ **Example 2.4.2.** There cannot be a program that decides whether another program will terminate in finite time.
- ▷ *Remark 2.4.3 (Church-Turing Hypothesis).* There are two classes of languages
  - ▷ Turing complete (or computationally universal) ones that can compute what is theoretically possible.
  - ▷ data languages that cannot. (but describe data sets)
- ▷ **Observation 2.4.4 (Turing Equivalence).** *All programming languages are (made to be) universal, so they can compute exactly the same. (compilers/interpreters exist)*
- ▷ **. . . in particular . . . :** Everybody who tells you that one programming languages is the best has no idea what they're talking about (though differences in efficiency, convenience, and beauty exist)

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".

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 2.4.5.** *Modern programming languages compose various primitives and give them a pleasing, concise, and uniform syntax.*

▷ **Question:** What does all of this even mean?

▷ **Definition 2.4.6.** In a programming language, a primitive is a "basic unit of processing", i.e. the simplest element that can be given a procedural meaning (its semantics) of its own.

▷ **Definition 2.4.7 (Compositionality).** All programming languages provide composition principles that allow to compose smaller program fragments into larger ones in such a way, that the semantics of the larger is determined by the semantics of the smaller ones and that of the composition principle employed.

▷ **Observation 2.4.8.** *The semantics of a programming language, is determined by the meaning of its primitives and composition principles.*

▷ **Definition 2.4.9.** Programming language syntax describes the surface form of the program: the admissible character sequences. It is also a composition of the syntax for the primitives.

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

Consequences of Compositionality

▷ **Observation 2.4.10.** *To understand a programming language, we (only) have to understand its primitives, composition principles, and their syntax.*

▷ **Definition 2.4.11.** The "art of programming" consists of composing the primitives of a programming language.

▷ **Observation 2.4.12.** *We only need very few – about half a dozen – primitives to obtain a Turing complete programming language.*

▷ **Observation 2.4.13.** *The space of program behaviors we can achieve by programming is infinites large nonetheless.*

▷ *Remark 2.4.14.* More primitives make programming more convenient.

▷ *Remark 2.4.15.* Primitives in one language can be composed in others.

A note on Programming: Little vs. Large Languages

▷ **Observation 2.4.16.** *Most such concepts can be studied in isolations, and some can be given a syntax on their own.* *(standardization)*

▷ **Consequence:** If we understand the concepts and syntax of the sublanguages, then learning another programming language is relatively easy.

2.5 More about Python

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

2.5.1 Sequences and Iteration

We now come to a commonly used class of objects in Python: sequences, such as lists, sets, tuples, and ranges as well as 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 2.5.1.** A list is a finite sequence of objects, its elements.

▷ In programming languages, lists are used for locally storing and passing around collections of objects.

▷ In Python lists can be written as a sequence of comma separated expressions between square brackets.

▷ **Definition 2.5.2.** We call [⟪seq⟫] the list constructor.

▷ **Example 2.5.3 (Three lists).** Elements can be of different types in Python

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"];
```

▷ **Example 2.5.4.** List elements can be accessed by specifying ranges

```
>>> list1[0]
'physics'
```

```
>>> list1[-2]
1997
```

```
>>> list2[1:4]
[2, 3, 4]
```

▷ **Definition 2.5.5.** Selecting sublists by specifying start and/or end is called list slicing.

As Example 2.5.4 shows, Python treats counting in list accessors somewhat peculiarly. It starts counting with zero when counting from the front and with one when counting from the back.

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

Sequences in Python

▷ **Definition 2.5.6.** Python has more types that behave just like lists, they are called sequence types.

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

▷ **Definition 2.5.7.** A range is a finite sequence of numbers it can conveniently be constructed by the range function: range(⟪start⟫,⟪stop⟫,⟪step⟫) constructs a range from ⟪start⟫ (inclusive) to ⟪stop⟫ (exclusive) with step size ⟪step⟫.

▷ **Example 2.5.8.** Lists can be constructed from ranges:

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

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

Ranges are useful, because they are easily and flexibly constructed for iteration (up next).
You may ask yourselves, why Python has a special data structure for ranges. The main reason is that we can treat them more efficiently than lists. Consider the range constructed by range(1,1000000000), i.e. the numbers between 1 and a billion. If we were to represent this as a list, then this would probably take most of the memory available on your laptop, even if we do not do anything with it. But in the range, Python only needs to actually create those parts of the range that it actually needs. Say we want to access the 1000th element, then the Python interpreter can just compute that as 1+1000 when it needs to (and free the memory when that is no longer needed); in particular, the interpreter does even have to create all the intermediate elements.

Iterating over Sequences in Python

▷ **Definition 2.5.9.** 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 2.5.10.** A range function makes an sequence over which we can iterate.

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

▷ **Example 2.5.11.** Lists and strings can also act as sequences. (try it)

```
print("Let me reverse something for you!")
x = input("please type somegthing!")
for i in reversed(list(x)):
    print(i)
```


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

Python Dictionaries

▷ **Definition 2.5.12.** A dictionary is an unordered collection of ordered pairs (k,v), where we call k the key and v the value.

▷ In Python dictionaries are written with curly brackets, pairs are separated by commata, and the value is separated from the key by a colon.

▷ **Example 2.5.13.** Dictionaries can be used for various purposes,

```
painting = {
    "artist": "Rembrandt",
    "title": "The Night Watch",
    "year": 1642
}
```

```
dict_de_en = {
  "Maus": "mouse",
  "Ast": "branch",
  "Klavier": "piano"
}
```

```
enum = {
  1: "copy",
  2: "paste",
  3: "adapt"
}
```

▷ Dictionaries and sequences can be nested, e.g. for a list of paintings.

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

Interacting with Dictionaries

▷ **Example 2.5.14 (Dictionary operations).**

- ▹ 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 2.5.15 (Printing Keys and Values).**

keys

```
for x in thisdict.keys():
    print(x)
```

values

```
for x in thisdict.values():
    print(x)
```

key/value pairs

```
for x, y in thisdict.items():
    print(x, y)
```

▷ More dictionary commands:

- ▹ **if** ⟪key⟫ **in** ⟪dict⟫ checks whether ⟪key⟫ is a key in ⟪dict⟫.
- ▹ painting.pop("title") removes the "title" item from painting.

Note that thisdict.keys has a short form: we can just iterate over the keys using **for** x **in** thisdict:.

2.5.2 Input and Output

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

Input/Output in Python

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

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

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

- **Definition 2.5.16.** A file is a resource for recording data in a storage device. File size is measured in bit.
- **Definition 2.5.17.** Files are identified by a file name which usually consists of a base name and an extension separated by a dot character.

  Files 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.
- Some file systems restrict the characters allowed in the file name and/or lengths of the base name or extension.
- **Definition 2.5.18.** Once a file has been opened, the CPU can write to it and read from it. After use a file should be closed to protect it from accidental reads and writes.

Many operating systems use files as a primary computational metaphor, also treating other resources like files. This leads to an abstraction of files called streams, which encompass files as well as e.g. keyboards, printers, and the screen, which are seen as objects that can be read from (keyboards) and written to (e.g. screens). This practice allows flexible use of programs, e.g. re-directing a the (screen) output of a program to a file by simply changing the output stream.

Now we can come to the Python bindings for the file input/output operations. They are rather straightforward.

Disk Input/Output in Python

- **Definition 2.5.19.** Python uses file objects to encapsulate all file input/output functionality.
- 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 $\widehat{=}$ write to the end), and r+ (read/write).
  - f.read() reads the file represented by file object 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 2.5.20 (Duplicating the contents of a file).**

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


The only interesting thing is that we have to declare our intentions when we opening a file. This allows the file system to protect the files against unintended actions and also optimize the data transfer to the storage devices involved.
Let us now look at some examples to fortify our intuition about what we can do with files in practice.

Disk Input/Output in Python (continued)

- **Example 2.5.21 (Reading a file linewise).**

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

```
>>> for line in f:
... print(line, end='')
...
This is the first line of the file.
Second line of the file
```

- 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 libraries (additional functionality)

▷ also useful for loading Python-encoded data (e.g. dictionaries)

The code snippet on the right of Example 2.5.21 show that files can be iterated over using a for loop: the file object is implicitly converted into a sequences of strings via the readline method.

2.5.3 Functions and Libraries in Python

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

Functions in Python (Introduction)

▷ **Observation:** Sometimes programming tasks are repetitive

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

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

▷ **Example 2.5.22.**We encapsultate the greeting functionality in a function:

```
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?"))
...
```

and use it repeatedly.

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

Functions in Python (Example)

▷ **Example 2.5.23 (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 2.5.24.** A Python function is defined by a code snippet of the form

```
def f (p_1,...,p_n):
    """docstring, what does this function do on parameters
        :param p_i: document arguments}
    """
    ⟨⟨body⟩⟩ # it can contain p_1, ..., p_n, and even f
    return ⟨⟨value⟩⟩ # value of the function call (e.g text or number)
⟨⟨more code⟩⟩
```

  - the indented part is called the body of f, (⚠ : whitespace matters in Python)
  - the p_i are called parameters, and n the arity of f.

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

We now come to a peculiarity of an object-oriented language like Python: it treats types as first-class entities, which can be defined by the user – they are called classes then. We will not go into that here, since we will not need classes in IWGS, but have have to briefly talk about methods, which are essentially functions, but have a special notation.
Python provides two kinds of function-like facilities: regular functions as discussed above and methods, which come with Python classes. We will not attempt a presentation of object oriented programming and its particular implementation in Python this would be beyond the mandate of the IWGS course – but give a brief introduction that is sufficient to use methods.

Functions vs. Methods in Python

- There is another mechanism that is similar to functions in Python. (we briefly introduce it here to delineate)
- **Background:** Actually, the types from ??? are classes, ...
- **Definition 2.5.25.** In Python all values belong to a class, which provide special functions we call methods. Values are also called objects, to emphasise class aspects. Method application is written with dot notation: ⟨⟨obj⟩⟩.⟨⟨meth⟩⟩(⟨⟨args⟩⟩) corresponds to ⟨⟨meth⟩⟩(⟨⟨obj⟩⟩,⟨⟨args⟩⟩).
- **Example 2.5.26.** Finding the position of a substring

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

Functions vs. Methods in Python

▷ **Example 2.5.27 (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 (of the objects they act on).

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 2.5.28.** A Python library is a Python file with a collection of functions, classes, and methods. It can be imported (i.e. loaded and interpreted as a Python program fragment) via the **import** command.

▷ There are $\geq$ 150.000 libraries for Python ($\widehat{=}$ 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.

2.5.4 A Final word on Programming in IWGS

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

For more information on Python

RTFM ($\widehat{=}$ "read the fine manuals")

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

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

Chapter 3

Numbers, Characters, and Strings

In our basic introduction to programming above we have convinced ourselves that we need some basic objects to compute with, e.g. Boolean values for conditionals, numbers to calculate with, and characters to form strings for input and output. In this chapter we will look at how these are represented in the computer, which in principle can only store binary digits voltage or no noltage on a wire – which we think of as 1 and 0.
In this chapter we look at the representation of the basic data structures of programming languages (numbers and characters) in the computer; Boolean values ("True" and "False") can directly be encoded as binary digits.

Documents as Digital Objects

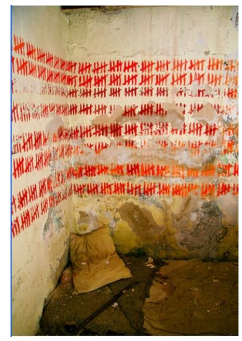
- ▷ **Question:** how do texts get onto the computer?(after all, computers can only do 0/1)
- ▷ **Hint:** At the most basic level, texts are just sequences of characters.
- ▷ **Answer:** We have to encode characters as sequences of bits.
- ▷ **We will go into how:**
  - ▷ documents are represented as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

3.1 Representing and Manipulating Numbers

We start with the representation of numbers. There are multiple number systems, as we are interested in the principles only, we restrict ourselves to the natural numbers – all other number systems can be built on top of these. But even there we have choices about representation, which influence the space we need and how we compute with natural numbers.
The first system for number representations is very simple; so simple in fact that it has been discovered and used a long time ago.

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 the number 11.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1

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

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 representation. 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 natural numbers, use position information and addition, multiplication, and exponentiation.
- **Definition 3.1.3.** A positional number system $\mathcal{N}$ is a pair $\langle D,\varphi\rangle$ with
  - D is a finite set of b digits; $b := \#(D)$ is the base or radix of $\mathcal{N}$.
  - $\varphi\colon D \to [0,b-1]$ is bijective.

  We extend φ to a bijection between sequences $d_k,\ldots,d_0$ of digits and natural numbers by setting

  $$\varphi(d_k,\ldots,d_0) := \sum_{i=0}^{k} \varphi(d_i)\cdot b^i$$

  We say that the digit sequence $n_b := d_k,\ldots,d_0$ is the positional notation of a natural number n, iff $\varphi(d_k,\ldots,d_0) = n$.
- **Example 3.1.4.** $\langle\{a,b,c\},\varphi\rangle$ with with $\varphi(a) := 0$, $\varphi(b):=1$, and $\varphi(c) := 2$ is a positional number system for base three. We have

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

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 digits in D_b, such that the position matters, and to give these encodings a meaning by mapping them into the unary natural numbers via a mapping φ_b.

Commonly Used Positional Number Systems

- **Definition 3.1.5.** The following positional number systems are in common use.

name	set	base	digits	example
unary	$\mathbb{N}_1$	1	0	00000_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	$FF3A12_{16}$

Binary digits are also called bits, and a sequence of eight bits an octet.

▷ **Notation:** 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)

▷ $110001101011100_2 = \underbrace{0110_2}_{6_{16}}\underbrace{0011_2}_{3_{16}}\underbrace{0101_2}_{5_{16}}\underbrace{1100_2}_{C_{16}} = 635C_{16}$

▷ $110001101011100_2 = \underbrace{110_2}_{6_8}\underbrace{001_2}_{1_8}\underbrace{101_2}_{5_8}\underbrace{011_2}_{3_8}\underbrace{100_2}_{4_8} = 61534_8$

▷ $F3A_{16} = \underbrace{F_{16}}_{1111_2}\underbrace{3_{16}}_{0011_2}\underbrace{A_{16}}_{1010_2} = 111100111010_2, \quad 4721_8 = \underbrace{4_8}_{100_2}\underbrace{7_8}_{111_2}\underbrace{2_8}_{010_2}\underbrace{1_8}_{001_2} = 100111010001_2$

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 (compound) digits for these groups. 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 electronic circuits, since binary digits can be represented physically by voltage/no voltage.

Arithmetics in Positional Number Systems

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

▷ Tom Lehrer's "New Math": `https://www.youtube.com/watch?v=DfCJgC2zezw`

▷ **Example 3.1.6.**

Addition base 4

$$\begin{array}{cccc} & 1 & 2 & 3 \\ + & 1_1 & 2_1 & 3 \\ \hline & 3 & 1 & 2 \end{array}$$

binary multiplication

$$\begin{array}{cccccc} & & 1 & 0 & 1 & 0 \\ & * & & 1 & 1 & 0 \\ \hline & & 0 & 0 & 0 & 0 \\ & 1 & 0 & 1 & 0 & \\ 1 & 0 & 1 & 0 & & \\ \hline 1 & 1 & 1 & 1 & 0 & 0 \end{array}$$

How to get back to Decimal (or any other system)

▷ **Observation:** ??? specifies how we can get from base b representations to decimal. We can always go back to the base b numbers.

▷ **Observation 3.1.7.** *To convert a decimal number n to base b, use successive integer division (division with remainder) by b:*

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

▷ **Example 3.1.8 (Convert 456 to base 8).** Result: 710_8

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

3.2 Characters and their Encodings: ASCII and UniCode

IT systems need to encode characters from our alphabets as bit strings (sequences of binary digits (bits) 0 and 1) for representation in computers. To understand the current state – the unicode standard – we will take a historical perspective. It is important to understand that encoding and decoding of characters is an activity that requires standardization in multi-device settings – be it sending a file to the printer or sending an e-mail to a friend on another continent. Concretely, the recipient wants to use the same character mapping for decoding the sequence of bits as the sender used for encoding them – otherwise the message is garbled.

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

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

The ASCII Character Code

▷ **Definition 3.2.1.** The American Standard Code for Information Interchange (ASCII) is a character encoding that assigns characters to numbers 0 to 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	[	\	]	ˆ	_
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.

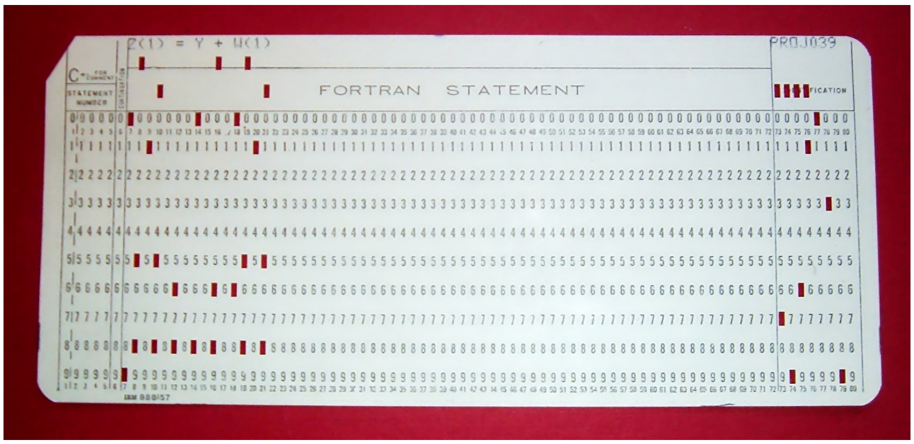
▷ **Motivated by punch cards:** The character 0 (0000000_2 in binary) carries no information NUL, (used as dividers)
Character 127 ($\widehat{=}\ 1111111_2$) can be used for deleting (overwriting) last value (cannot delete holes)

▷ 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

- ▷ **Definition 3.2.2.** A punch card is a piece of stiff paper that contains digital information represented by the presence or absence of holes in predefined positions.
- ▷ **Example 3.2.3.** This punch card encodes the `FORTRAN` statement Z(1) = Y + W(1)



Up to the 1970s, computers were batch machines, where the programmer delivered the program to the operator (a person behind a counter who fed the programs to the computer) and collected the printouts the next morning. Essentially, each punch card represented a single line (80 characters) of program code. Direct interaction with a computer is a relatively young mode of operation.
The ASCII code as above has a variety of problems, for instance that the control characters are mostly no longer in use, the code is lacking many characters of languages other than the English language it was developed for, and finally, it only uses seven bits, where an octet (eight bits) is the preferred unit in information technology. Therefore a whole zoo of extensions were introduced, which — due to the fact that there were so many of them — never quite solved the encoding problem.

Problems with ASCII encoding

- ▷ **Problem:** Many of the control characters are obsolete by now/ (e.g. `NUL`,`BEL`, or `DEL`)
- ▷ **Problem:** Many European characters are not represented. (e.g. è,ñ,ü,ß,. . .)
- ▷ **European ASCII Variants:** Exchange less-used characters for national ones.
- ▷ **Example 3.2.4 (German ASCII).** Remap e.g. [↦ Ä,] ↦ Ü in German ASCII (“Apple][” comes out as “`Apple ÜÄ`”)
- ▷ **Definition 3.2.5 (ISO-Latin (ISO/IEC 8859)).** 16 Extensions of ASCII to 8-bit (256 characters) ISO Latin 1 $\widehat{=}$ “Western European”, ISO Latin 6 $\widehat{=}$ “Arabic”, ISO Latin 7 $\widehat{=}$ “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, and (character set)
- a mapping from bit strings to characters. (character encoding)

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

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

Unicode and the Universal Character Set

▷ **Definition 3.2.6 (Twin Standards).** A scalable architecture for representing all the worlds writing systems:

- The universal character set (UCS) defined by the ISO/IEC 10646 International Standard, is a standard set of characters upon which many character encodings are based.
- The unicode standard defines a set of standard character encodings, rules for normalization, decomposition, collation, rendering and bidirectional display order.

▷ **Definition 3.2.7.** Each UCS character is identified by an unambiguous name and an natural number called its code point.

▷ The UCS has 1.1 million code points and nearly 100 000 characters.

▷ **Definition 3.2.8.** Most (non-Chinese) characters have code points in $[1,65536]$: the basic multilingual plane (BMP).

▷ **Definition 3.2.9 (Notation).** For code points in the (BMP), four hexadecimal digits are used, e.g. $\mathrm{U}+0058$ for the character LATINCAPITALLETTERX;

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

Character Encodings in Unicode

▷ **Definition 3.2.10.** A character encoding is a mapping from bit strings to UCS code points.

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

▷ **Definition 3.2.11 (Unicode Transformation Format).**

- ▷ UTF-8, 8-bit, variable width character encoding, which maximizes compatibility with ASCII.
- ▷ UTF-16, 16-bit, variable width character encoding (popular in Asia)
- ▷ UTF-32, a 32-bit, fixed width character encoding (as a fallback)

▷ **Definition 3.2.12.** The UTF-8 encoding follows the following schema:

Unicode	octet 1	octet 2	octet 3	octet 4
$\mathrm{U}+000000-\mathrm{U}+00007\mathrm{F}$	0xxxxxxx			
$\mathrm{U}+000080-\mathrm{U}+0007\mathrm{FF}$	110xxxxx	10xxxxxx		
$\mathrm{U}+000800-\mathrm{U}+00\mathrm{FFFF}$	1110xxxx	10xxxxxx	10xxxxxx	
$\mathrm{U}+010000-\mathrm{U}+10\mathrm{FFFF}$	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

▷ **Example 3.2.13.** $\$ = \mathrm{U}+0024$ is encoded as 00100100 (1 byte)
¢ $= \mathrm{U}+00\mathrm{A}2$ is encoded as 11000010,10100010 (two bytes)
€ $= \mathrm{U}+20\mathrm{AC}$ is encoded as 11100010,10000010,10101100 (three bytes)

FAU : 82 2025-06-05

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

XKCD's Take on Recent Unicode Extensions

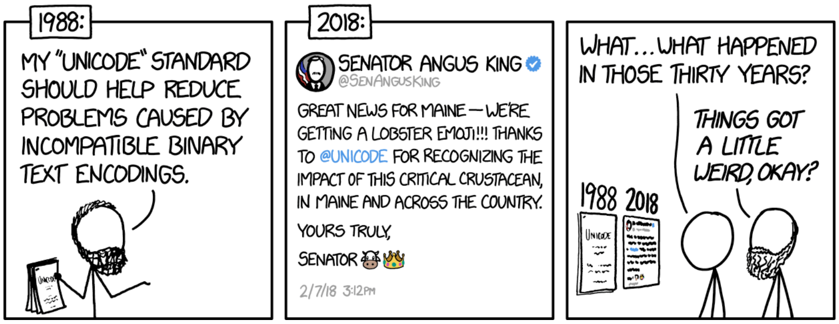
▷ UniCode 6.0 adopted hundreds of emoji characters in 2010 (2666 in July 2017)

▷ Modifying characters (https://xkcd.com/1813/)

FAU : 83 2025-06-05

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

▷ Recent UniCode extensions (https://xkcd.com/1953/)



3.3 More on Computing with Strings

We now extend our repertoire on handling and formatting strings in Python: we will introduce string literals, which allow writing complex strings.

Playing with Strings and Characters in Python

▷ **Definition 3.3.1.** Python strings are sequences of UniCode characters.

▷ ⚠ In Python, characters are just strings of length 1.

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

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

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

▷ Strings can be accessed by ranges $[i{:}j]$ ($[i] \mathrel{\widehat{=}} [i{:}i]$)

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

```
def cap(s) :
    if s == "":
        return "" # base case
    else:
        return uc(s[0]) + cap(s[1:len(s)])

>>> cap('iwgs')
'IWGS'
```

Example 3.3.3 may be difficult to understand at first. It is a programming technique called recursion, i.e. functions that call themselves from within their body to solve problems by utilizing solutions to smaller instances of the same problem. Recursion can lead to very concise code, but requires some getting-used-to.

In Example 3.3.3 we define a function cap that given a string s returns a string that is constructed by combining the first character uppercased by the uc function with the result of calling the cap function on the rest string – s without the first character. The base case for the recursion is the empty string, where uc also returns the empty string. So let us see what happens in our test cap('iwgs'):
cap('iwgs') ⇝ uc('i')+cap('wgs') ⇝ 'I'+uc('w')+cap('gs') ⇝ 'I'+'W'+uc('g')+cap('s') ⇝ 'IW'+'G'+cap('s') ⇝ 'IWG'+uc('s')+cap('') ⇝ 'IWG'+'S'+cap('') ⇝ 'IWGS'+'' ⇝ 'IWGS'

Example 3.3.2 and Example 3.3.3 (or any other examples in this lecture) are not production code, but didactically 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 UCS code: the answer is \x83, which is the character with the hexadecimal code 83 (decimal 131), i.e. the character No Break Here.

In production code (as used e.g. in the Python lower method), we would have some range checks, etc.

String Literals in Python

- **Problem:** How to write strings including special characters?
- **Definition 3.3.4.** A literal is a notation for representing a fixed value for a data structure in source code.
- **Definition 3.3.5.** 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.

Raw String Literals in Python

- **Definition 3.3.6.** 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 3.3.7.** 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.

Now that we understand the "theory" of encodings, let us work out how to program with them in Python:
Programming with UniCode strings is particularly simple, strings in Python are UTF-8-encoded UniCode strings and all operations on them are UniCode-based[1]. This makes the introduction to UniCode in Python very short, we only have to know how to produce non-ASCII characters, i.e. the characters that are not on regular keyboards.

If we know the code point, this is very simple: we just use UniCode escape sequences.

Unicode in Python

- *Remark 3.3.8.* The Python string data type is UniCode, encoded as UTF-8.
- **How to write UniCode characters?:** there are five ways
  - write them in your editor (make sure that it uses UTF-8)
  - otherwise use Python escape sequences (try it!)

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


Note that the discussion about entry methods for unicode characters applies to the bare Python interpreter, not Python-specific text editor modes or user interfaces, which are often helpful by automatically replacing the input by the respective glyphs themselves.
String literals are convenient for creating simple string objects. For more complex ones, we usually want to build them from pieces, usually using the values of variables or the results of functions. This is what f strings are for in Python; we will cover that now.

Formatted String Literals (aka. f-strings)

- **Problem:** In a program we often want to build strings from pieces that we already have lying around interspersed by other strings.
- **Solution:** Use string concatenation:

```
>>> course="IWGS"
>>> students=6*11
>>> "The " + course + " course has " + str(students) + " students"
'The IWGS course has 66 students'
```

- We can do better! (mixing blanks and quotes is error-prone)

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

▷ **Definition 3.3.9.** Formatted string literals (aka. f strings) are string literals can contain Python expressions that will be evaluated – i.e. replaced with their values at runtime.

F strings are prefixed by f or F, the expressions are delimited by curly braces, and the characters { and } themselves are represented by {{ and }}.

▷ **Example 3.3.10 (An f-String for IWGS).**

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


F-String Example with a Dictionary

▷ **Example 3.3.11 (An F-String with a 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
```


3.4 More on Functions in Python

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

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

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

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

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

Anonymous Functions (lambda)

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

▷ **Definition 3.4.2.** Python also allows to make anonymous functions via the function literal lambda for function objects:

lambda p_1,. . .,p_n: ⟪expr⟫

▷ **Example 3.4.3.** The following two Python fragments are equivalent:

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

```
cube = lambda x: x*x*x
```

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

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

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

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

Higher-Order Functions in Python

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

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

- ▷ map(f,L) returns the list of f-values of the elements of L.
- ▷ filter(p,L) returns the sub-list L' of those l in L, such that $p(l)$=True.

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

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


Admittedly, in our example, we could also have defined a named function twice and then mapped that over li:

```
def twice (x):
    x*x
map twice li
```

But the code from Example 3.4.6 is more compact. Once we get used to the programming idiom and understand it, it becomes quite readable.

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

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

Argument Passing in Python: Keyword Arguments

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

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

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

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

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

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


BTW: The opener argument of open is a function, and often an anonymous function is used if it is specified.

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

Argument Passing in Python: Flexible Arity

▷ **Definition 3.4.9.** Python functions can take a variable number of arguments: def f $(p_1,\ldots,p_k,*r)$ allows $n \geq k$ arguments, e. g. $f(a_1,\ldots,a_k,a_{k+1},\ldots,a_n)$ and binds the parameter r the rest argument to the list $[a_{k+1},\ldots,a_n]$.

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

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

▷ **Definition 3.4.11.** The star operator unpacks a list into an argument sequence.

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

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


Actually the star operator can be used in other situations as well, consider for instance

```
>>> numbers = [2, 1, 3, 4, 7]
>>> more_numbers = [*numbers, 11, 18]
```

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

Here we have used the star operator twice: First to pass the list numbers as arguments to the list constructor and a second time to pass the extended list more_numbers to the print function.
Finally, we can combine the ideas from the last two to make keyword arguments flexary.

Argument Passing in Python: Flexible Keyword Arguments

▷ **Definition 3.4.13.** Python functions can take keyword arguments:
if k is a sequence of key/value pairs then def $f(p_1,\ldots,p_n,{*}{*}k)$ binds the keys to values in the body of f.

▷ **Example 3.4.14.**

```
def kw_args(farg, **kwargs):
    print (f"formal arg: {farg}")
    for key in kwargs :
        print (f"another keyword arg: {key}: {kwargs[key]}")
>>> kw_args(1, myarg2="two", myarg3=3)
formal arg: 1
another keyword arg: myarg2 : two
another keyword arg: myarg3 : 3
```


Just as for the flexible arity case above, we have an operator that unpacks argument structures, here a dictionary.

Argument Passing in Python: Flexible Keyword Arguments (cont.)

▷ **Definition 3.4.15.**3 The double star operator unpacks a dictionary into a sequence of keyword arguments.

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

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

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

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


Disclaimer: The last couple of features of Python functions are a bit more advanced than would usually be expected from a Python programming introduction in a course such as IWGS. But one of the goals of IWGS is to empower students to be able to read Python code of more experienced authors. And that kind of code may very well contain these features, so we need to cover them in IWGS.

So the last couple of slides should be considered as an "early exposure for understanding" rather than "essential to know for IWGS" content.

3.5 Regular Expressions: Patterns in Strings

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

Problem: Text/Data File Manipulation

- **Problem 1 (Information Extraction):** We often want to extract information from large document collections, e.g.
  - e-mail addresses or dates from collected correspondencesrtts
  - dates and places from newsfeeds
  - links from web pages
- **Problem 2 (Data Cleaning):** The representation in data files is often too noisy and inconsistent for directly importing into an application; e.g.
  - standardizing different spellings of e.g. city names, (Nuremberg vs. Nürnberg)
  - eliminating higher UniCode characters, when the application only accepts ASCII,
  - separating structured texts into data blocks. (e.g. in x-separated lists)
- **Enabling Technology:** Specifying text/data fragments $\rightsquigarrow$ regular expressions.

FAU : 97 2025-06-05

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 regular expression tester).

Regular Expressions, see [Pyt]

- **Definition 3.5.1.** A regular expression (also called regular expression) is a formal expression that specifies a set of strings.
- **Definition 3.5.2 (Meta-Characters for Regexps).**

char	denotes
.	any single character (except a newline)
^	beginning of a string
$	end of a string
[...]/[^...]	any single character in/not in the brackets
[x−y]/[^x−y]	any single character in/not in range x to y
[...]	marks a capture group
\\n	the n^{th} captured group
|	disjunction
∗	matches preceding element zero or more times
+	matches preceding element one or more times
?	matches preceding element zero or one times
{n,m}	matches the preceding element between n and m times
\S/\s	non-/whitespace character
\W/\w	non-/word character
\D/\d	non-/digit (not only 0-9, but also e.g. arabic digits)

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

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

Regular Expression Examples

▷ **Example 3.5.3 (Regular Expressions and their Values).**

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

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

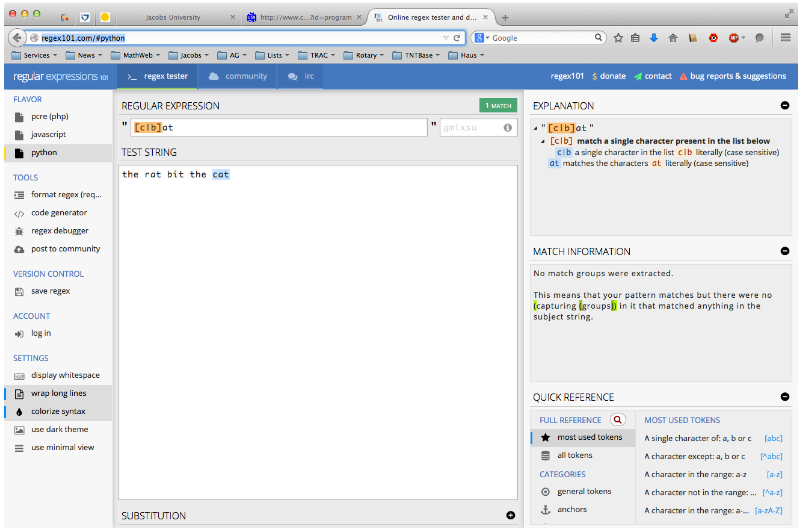
▷ **Example 3.5.4 (A more complex example).** The following regular expression matches times in a variety of formats, such as 10:22am, 21:10, 08h55, and 7.15 pm.

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


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



After covering regular expressions in the abstract, we will see how they are integrated into programming languages to solve problems. Of course we take Python as an example.

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 ⟪pat⟫.

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


As regular expressions form a special language for describing sets of strings, it is not surprising that they are used in all kinds of searching, splitting, and substring replacement operations. As the language of regular expressions is well standardized, these more or less work the same in all programming languages, so what you learn for Python, you can re-use in other languages.

We will now see what we can do with regular expressions in a practical example. You should consider it as a "code reading/understanding" exercise, not think of it as something you should (easily) be able to do yourself. But ??? could serve as a quarry of ideas for things you can do to texts with regular expressions.

Example: Correcting and Anonymizing Documents

▷ **Example 3.5.5 (Document Cleanup).**

We write a function that makes simple corrections on documents and also crosses out all names to anonymize.

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

Here is the function

- we import the regular expressions library and start the function

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

- we first add blanks after commata

```
    s = re.sub(r",(\S)", r", \1", 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)
```


This program is just a series of stepwise regular expression computations that are assigned to the variable s. For the last one, we use the **lambda** operator that constructs a function as an argument (the second) to re.sub. We use the anonymous functions because this function is only used once. This worked well, so we just continue along these lines.

Example: Correcting and Anonymizing Documents (cont.)

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

- next we make abbreviations for regular expressions to save space

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

- remove capital letters in the middle of words

```
    s = re.sub(f"({l})({c}+)({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

```
s
```

"*The worst president of the US,arguably was George W. Bush, right?*"
becomes
"*The worst president of the US, arguably was XXXXXX XX XXXX, right?*"

We show the whole program again, to see that it is relatively small (thanks to the very compact – if cryptic – regular expressions), when we leave out all the comments.

Example: Correcting and Anonymizing Documents (all)

▷ **Example 3.5.7 (Document Cleanup (overview)).**

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

Chapter 4

Documents as Digital Objects

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

4.1 Representing & Manipulating Documents on a Computer

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

Electronic Documents

- ▷ **Definition 4.1.1.** An electronic document is any media content that is intended to be used via a document renderer, i.e. a program or computing device that transforms it into a form that can be directy perceived by the end user.
- ▷ **Example 4.1.2.** PDFs, digital images, videos, audio recordings, web pages, ...
- ▷ **Definition 4.1.3.** An electronic document that contains a digital encoding of textual material that can be read by the end user by simply presenting the encoded characters is called digital text.
- ▷ **Definition 4.1.4.** Digital text is subdivided into plain text, where all characters carry the textual information and formatted text, which also contains instructions to the document renderer.
- ▷ **Example 4.1.5.** Python programs are plain text, PDFs are formatted.

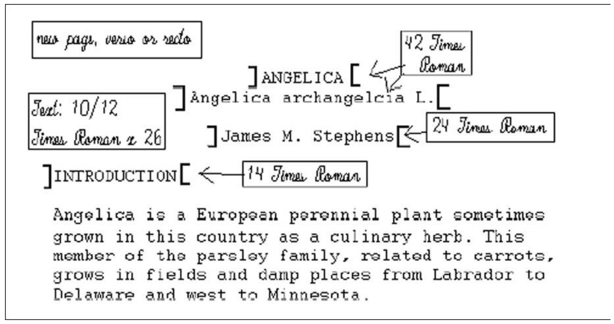
FAU : 105 2025-06-05

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

Document Markup

- ▷ **Definition 4.1.6.** Document markup (or just markup) is the process of adding control words (special character sequences also called markup codes) to a plain text to control the structure, formatting, or the relationship among its parts, making it a formatted text. All characters of a formatted text that are not control words constitute its textual content.

▷ **Example 4.1.7.** A text with markup codes (for printing)



▷ **Definition 4.1.8.** The control words and composition rules for a particular kind of markup system determine a markup format (also called a markup language). The markup format used in an electronic document is called its document type.

▷ *Remark 4.1.9.* Markup turns plain text into formatted text.

There are many systems for document markup, ranging from informal ones as in Example 4.1.7 that specify the intended document appearance to humans – in this case the printer – to technical ones which can be understood by machines but serving the same purpose.
Markup is by no means limited to visual markup for documents intended for printing as Example 4.1.7 may suggest. There are aural markup formats that instruct document renderers that transform documents to audio streams of e.g. reading speeds, intonation, and stress.
We now come to another aspect of electronic documents: We mostly interact with them in the form of files. Again, we fix our nomenclature.

File Types

▷ **Observation 4.1.10.** *We mostly encounter electronic documents in the form of files on some storage medium.*

▷ **Definition 4.1.11.** A text file is a file that contains text data, a binary file one that contains binary data

Definition 4.1.12. Text files are often processed as a sequence of text lines (or just lines), i.e. sub string separated by the line feed character $\mathrm{U}+000\mathrm{A}$; $\mathrm{LINEFEED}(\mathrm{LF})$. The line number is just the position in the sequence.

▷ *Remark 4.1.13.* Text files are usually encoded with ASCII, ISO Latin, or increasingly UniCode encodings like UTF-8.

▷ **Example 4.1.14.** Python programs are stored in text files.

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

Remark 4.1.15. Plain text is different from formatted text, which includes markup code, and binary files in which some portions must be interpreted as binary data (encoded integers, real numbers, digital images, etc.)

As we have seen above, it does not take much to render a text file: we only need to guess the right encoding scheme so we can decode the file and show the character sequence to the user. Indeed the `UNIX` cat just prints the contents of a text file to a shell. But we need much more, we need tools with which we can compose and edit text files; we do this with text editors, which we will discuss now.

Text Editors

▷ **Definition 4.1.16.** A text editor is a program used for rendering and manipulating text files.

▷ **Example 4.1.17.** Popular text editors include

- ▷ `Notepad` is a simple editor distributed with `MSWindows`.
- ▷ `emacs` and `vi` are powerful editors originating from `UNIX` and optimized for programming.
- ▷ `sublime` is a sophisticated programming editor for multiple operating systems.
- ▷ `EtherPad` is a browser-based real-time collaborative editor.

▷ **Example 4.1.18.** Even though it can save documents as text files, `MSWord` is not usually considered a text editor, since it is optimized towards formatted text; such "editors" are called word processors.

What text editors do for text files, word processors do for other electronic documents.

Word Processors and Formatted Text

▷ **Definition 4.1.19.** A word processor is a software application, that – apart from being a document renderer – also supports the tasks of composition, editing, formatting, printing of electronic documents.

▷ **Example 4.1.20.** Popular word processors include

- ▷ `MSWord`, an elaborated word processor for `MSWindows`, whose native format is Office Open XML (OOXML; file extension .docx).
- ▷ `OpenOffice` and `LibreOffice` are similar word processors using the ODF format (Open Office Format; file extension .odf) natively, but can also import other formats..
- ▷ `Pages`, a word processors for `macOS` it uses a proprietary format.
- ▷ `OfficeOnline` and `GoogleDocs` are browser-based real-time collaborative word processors.

▷ **Example 4.1.21.** Text editor are usually not considered to be word processors, even though they can sometimes be used to edit markup based formatted text.

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

4.2 Measuring Sizes of Documents/Units of Information

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

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

Units for Information

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

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 cell by binary number, therefore the derived information units are prefixed by special prefixes that are based on powers of 2.
- ▷ **Definition 4.2.3 (Binary Prefixes).** The following binary unit 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 4.2.4.** You can buy hard-disks that say that their capacity is "one terabyte", 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)	*binary digit 0/1*
Byte (B)	*8 bit*
2 Bytes	A UniCode character in UTF.
10 Bytes	your name.
Kilobyte (kB)	*1,000 bytes OR* 10^3 *bytes*
2 Kilobytes	A Typewritten page.
100 Kilobytes	A low-resolution photograph.
Megabyte (MB)	*1,000,000 bytes OR* 10^6 *bytes*
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 (GB)	*1,000,000,000 bytes or* 10^9 *bytes*
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 (TB)	*1,000,000,000,000 bytes or* 10^{12} *bytes*
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 (PB)	*1,000,000,000,000,000 bytes or* 10^{15} *bytes*
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 (EB)	*1,000,000,000,000,000,000 bytes or* 10^{18} *bytes*
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 (ZB)	*1,000,000,000,000,000,000,000 bytes or* 10^{21} *bytes*
2 Zettabytes	Total volume digital data transmitted in 2011
100 Zettabytes	Data equivalent to the human Genome in one body.

FAU : 113 2025-06-05

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

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

4.3 Hypertext Markup Language

WWW documents have a specialized document type that mixes markup for document structure with layout markup, hyper-references, and interaction. The HTML markup elements always concern text fragments, they can be nested but may not otherwise overlap. This essentially turns a text into a document tree. In IWGS, we discuss HTML mostly as a way to build interfaces of web applications. Therefore we will prioritize those aspects of HTML that have to do with "programming documents" over the creation of nice-looking web pages. Therefore we will pick up the notion of nested text fragments marked up by well-bracketed tags and elements in section 4.4 and generalize these ideas to XML as a general representation paradigm for semi-structured data in ???.

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

4.3.1 Introduction

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

HTML: Hypertext Markup Language

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

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

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

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

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

FAU : 114 2025-06-05

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

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

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

A very first HTML Example (Source)

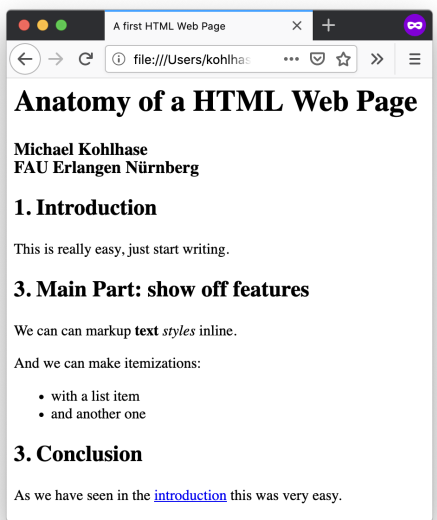
```
<html xmlns="http:www.w3.org/1999/xhtml">
  <head>
    <title>A first HTML Web Page</title>
  </head>
  <body>
    <h1>Anatomy of a HTML Web Page</h1>
    <h3>Michael Kohlhase<br/>FAU Erlangen Nuernberg</h3>
    <h2 id="intro">1. Introduction</h2>
    <p>This is really easy, just start writing.</p>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

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

FAU : 115 2025-06-05

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 [Hic+14] 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)



FAU : 116 2025-06-05

4.3.2 Interacting with HTML in Web Broswers

In the last slide, we have seen `FireFox` as a document renderer for HTML. We will now introduce this class of programs in general and point out a few others.

Web Browsers

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

Definition 4.3.5. A web browser usually supplies user tools like

▷ history that gives the user access to web pages visited earlier and

- bookmark to remember web pages.

Definition 4.3.6. A web browser usually supplies developer tools like

- the console that logs system-level events in the browser and
- an inspector that gives access to the structure and content of the DOM.

▷ **Practical Browser Tools:**

- Status Bar: security info, page load progress
- Favorites (bookmarks)
- View Source: view the code of a web page
- Tools/Internet Options, history, temporary Internet files, home page, auto complete, security settings, programs, etc.

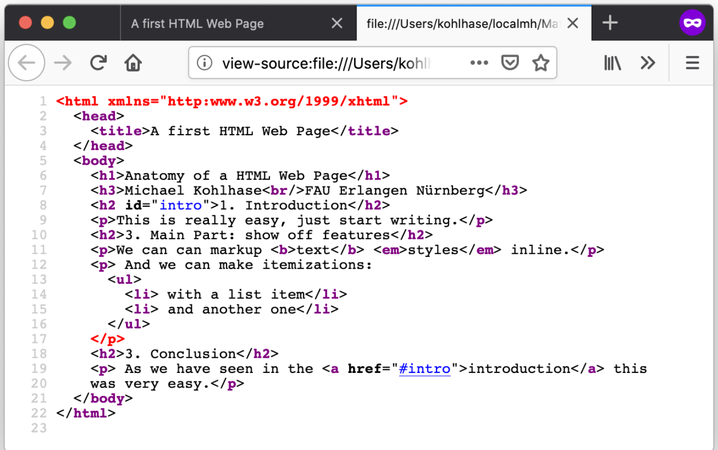
▷ **Example 4.3.7 (Common Browsers).**

- `MSInternetExplorer` is an once dominant, now obsolete browser for `MSWindows`.
- `Edge` is provided by Microsoft for `MSWindows`.(replaces `MSInternetExplorer`)
- `FireFox` is an open source browser for all platforms, it is known for its standards compliance.
- `Safari` is provided by Apple for `macOS` and `MSWindows`.
- `Chrome` is a lean and mean browser provided by Google Inc. (very common)
- `WebKit` is a library that forms the open source basis for `Safari` and `Chrome`.

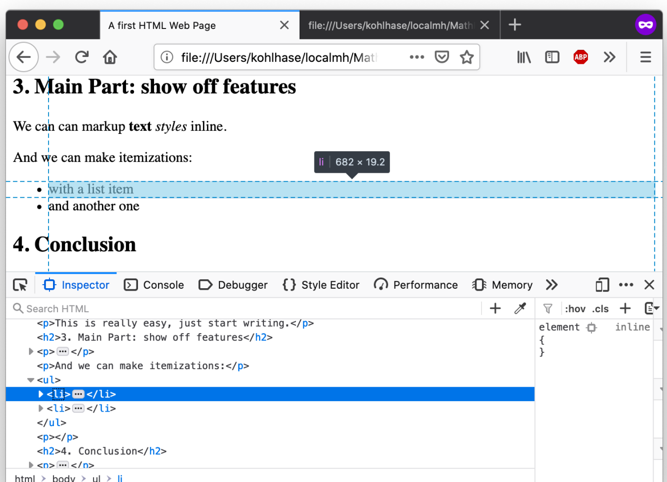
Let us now look at a couple of more advanced tools available in most web browsers for dealing with the underlying HTML document.

Browser Tools for dealing with HTML, e.g. in `FireFox`

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



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



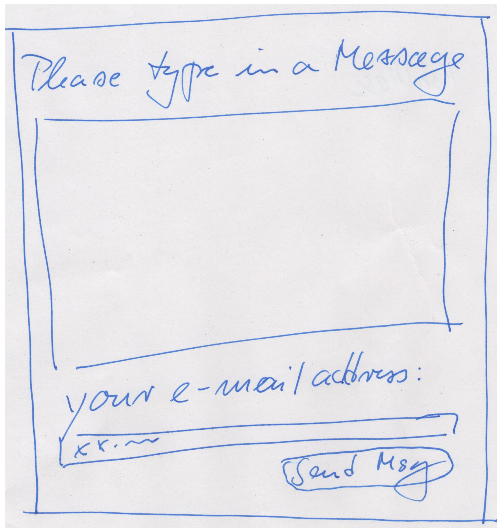
We have used FireFox as an example here, but these tools are available in some form in all major browsers the browser vendors want to make their offerings attractive to web developers, so that web pages and web applications get tested and debugged in them and therefore work as expected.

4.3.3 A Worked Example: The Contact Form

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

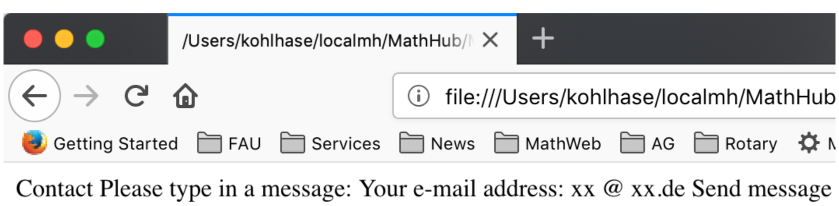
▷ Make a design and “paper prototype” of the page:



▷ Put the intended text into a file: contact.html:

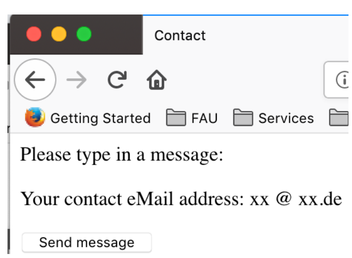
```
Contact
Please enter a message:
Your e—mail address: xx @ xx.de
Send message
```

▷ Load into your browser to check the state:



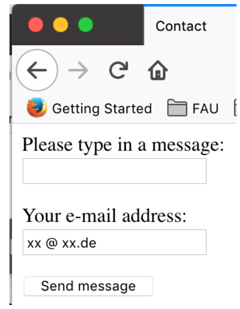
▷ Add title, paragraph and button markup:

```
<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:

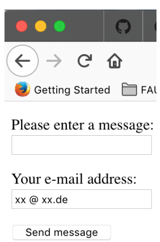
```
<title>Contact</title>
<h2>Please enter a message:</h2>
<input name="msg" type="text"/>
<h3> Your e-mail address:</h3>
<input name="addr" type="text"
       value="xx @ xx.de"/>
<br/>
<button>Send message</button>
```

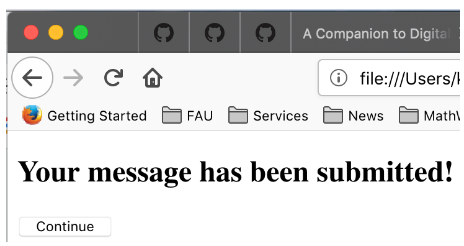


▷ Convert into a HTML form with action (message receipt):

```
<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 4.3.8.** A HTML form is realized by the HTML form tags, which groups the layout, form action specification and input fields:

- ▹ <**form action**="⟨⟨URI⟩⟩"...> specifies the form action (as a web page address).
- ▹ the input field <**input type**="submit".../> triggers the form action: it sends the form data to web page specified there.

▷ **Example 4.3.9 (In the Contact Form).** We send the request

```
GET contact-after.html?
    msg=Hi;addr=foo@bar.de
```

We current ignore the form data (the part after the ?)

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

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 tags 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
```

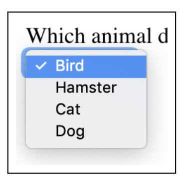
My major is ☐ Computer Science ☐ Digital Humanities ☐ Other

▷ File selector dialogs (interaction is system specific here for MacOS Mojave)

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

▷ 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>
```



4.4 Documents as Trees

We have concentrated on HTML as a document type for interactive multimedia documents. Before we progress, we want to discuss an important feature: all practical document types that employ control words are in some sense well-bracketed. Well-bracketed structures are well-understood in CS and mathematics: they are called trees and come with a rich and useful collection of descriptive concepts and tools. We will present the concepts in this section and the tools they enable in ???.

Well-Bracketed Structures in Computer Science

- **Observation 4.4.1.** *We often deal with well-bracketed structures in CS, e.g.*
  - *Expressions: e.g.* $\frac{3\cdot(a+5)}{2x+7}$ *(numerator an denominator in fractions implicitly bracketed)*
  - *Markup languages like HTML:*

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

  - *Programming languages like python:*

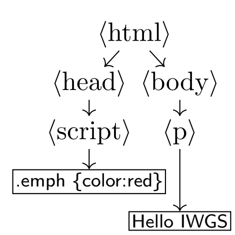
```
answer = input("Are you happy? ")
if answer == 'No' or answer == 'no':
    print("Have a chocolate!")
else:
    print("Good!")
print("Can I help you with something else?")
```

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

A Common Data Structure for Well Bracketed Structures

- **Observation 4.4.2.** *In well-bracketed strutures, brackets contain two kinds of objects*
  - *bracket-less objects*
  - *well-bracketed structures themselves*
- **Idea:** Write bracket pairs and bracket-less objects as nodes, connect with an arrow when contained. (let arrows point downwards)
- **Example 4.4.3.** Let's try this for HTML creating nodes top to bottom

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



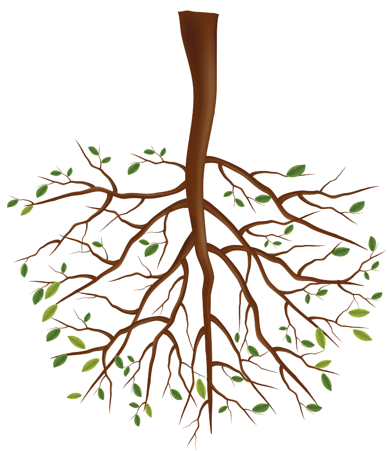
- **Definition 4.4.4.** We call such structures tree. (more on trees next)

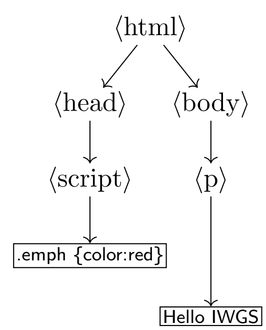
Trees are well understood mathematical objects and tree data structures are very commonly used in computer science and programming. As such they have a well-developed nomenclature, which we will introduce now.

Well-Bracketed Structures: Tree Nomenclature

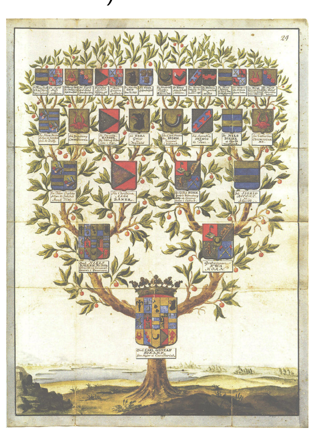
▷ **Definition 4.4.5.** In mathematics and CS, such well-bracketed structures are called trees (with root, branches, leaves, and height). (but written upside down)

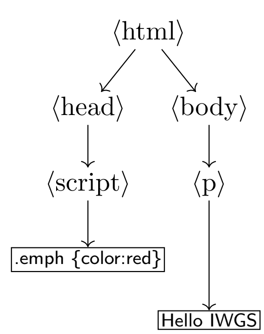
▷ **Example 4.4.6.** In a tree, there is only one path from the root to the leaves





▷ **Definition 4.4.7.** We speak of parent, child, ancestor, and descendant nodes (genealogy nomenclature).

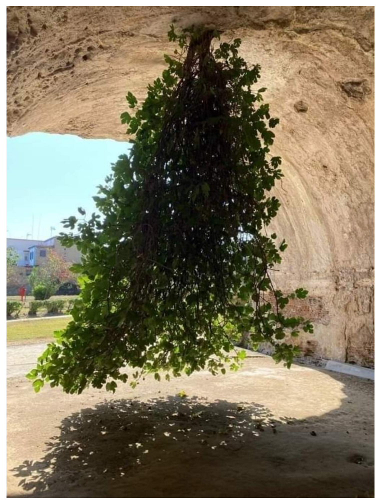




Why are trees written upside-down?: The main answer is that we want to draw tree diagrams in text. And we naturally start drawing a tree at the root. So, if a tree grows from the root and we do not exactly know the tree height, then we do not know how much space to leave. When we write trees upside down, we can directly start from the root and grow the tree downward as long as we need. We will keep to this tradition in the IWGS course.

Upside Down Trees in Nature

- Actually, upside down trees exist in nature (though rarely):

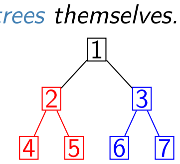


This is a fig tree in Bacoli, Italy; see `https://www.atlasobscura.com/places/upside-down-fig-tree`

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

Computing with Trees in Python

- **Observation 4.4.8.** *All connected substructures of trees are trees themselves.*
  - **Idea:** Operate on the tree by "Divide and Conquer"
    - operate on the two subtrees
    - combine results, taking root into account



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

- **Idea:** Represent trees as lists of tree labels and lists (of subtrees).
- **Example 4.4.9 (The tree above).** Represented as [1,[2,[[4],[5]]],[3,[[6],[7]]]]
  compute the tree height by the following Python functions:

```
def height (tree):
    return maxh(tree[1:]) + 1

height([1,[2,[[4],[5]]],[3,[[6],[7]]]])
>>> 3
```

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


Let use have a closer look at Example 4.4.9. The algorithm consists of two functions:

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

Note that maxh and height each call the other. We call such functions mutually recursive. Here this behavior poses no problem, since the arguments in the recursive calls are smaller than the inputs: for maxh it is the rest list, and for height the "list of children" of the input tree.
Example 4.4.9 was complex for two reasons: mutual recursion and the somewhat cryptic encoding of trees as lists of lists of integers. We claim that recursive programming is "not a bug, but a feature", as it allows to succinctly capture the "divide-and-conquer" approach afforded by trees. For the cryptic encoding of trees we can do better.

Computing with Trees in Python (Dictionaries)

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

▷ **Idea:** why not use dictionaries? (they are more explicit)

▷ **Example 4.4.10.** Compute the tree weight (the sum of all labels) by

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

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

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

weight(t);
>>> 28
```


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

The Document Object Model

▷ **Definition 4.4.11.** The document object model (DOM) is a data structure for storing marked up electronic documents as trees together with a standardized set of access methods for manipulating them.

▷ **Idea:** When a web browser loads a HTML page, it directly parses it into a DOM and then works exclusively on that. In particular, the HTML document is immediately discarded; documents are rendered from the DOM.

4.5 An Overview over XML Technologies

We have seen that many of the technologies that deal with marked-up documents utilize the tree-like structure of (the DOM) of HTML documents. Indeed, it is possible to abstract from the concrete vocabulary of HTML that implements 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.

4.5.1 Introduction to XML

XML (EXtensible Markup Language)

▷ **Definition 4.5.1.** XML (short for Extensible Markup Language) is a framework for markup formats for documents and structured data.

- ▷ Tree representation language (begin/end brackets)
- ▷ Restrict instances by *Doc. Type Def. (DTD)* or *Schema* (Grammar)
- ▷ Presentation markup by *style files* (XSL: XML Style Language)

▷ **Intuition:** XML is extensible HTML

▷ logic annotation (*markup*) instead of presentation!

▷ many tools available: parsers, compression, data bases, ...

▷ **conceptually**: transfer of trees instead of strings.

▷ details at `http://w3c.org` (XML is standardize by the WWW Consortium)

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

XML is Everywhere (E.g. Web Pages)

▷ **Example 4.5.2.** Open web page file in `FireFox`, then click on $View \searrow PageSource$, you get the following text: (showing only a small part and reformatting)

```
<html xmlns="http://www.w3.org/1999/xhtml">
 <head>
  <title>Michael Kohlhase</title>
  <meta name="generator"
```

```
        content="Page generated from XML sources with the WSML package"/>
  </head>
  <body>...
    <p>
      <i>Professor of Computer Science</i><br/>
      Jacobs University<br/><br/>
      <strong>Mailing address - Jacobs (except Thursdays)</strong><br/>
      <a href="http://www.jacobs-university.de/schools/ses">
        School of Engineering amp; Science</a><br/>...</p>...</body></html>
```

▷ **Definition 4.5.3.** XHTML is the XML version of HTML.(just make it valid XML)

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

XML is Everywhere (E.g. Catalogs)

▷ **Example 4.5.4 (The NYC Galleries Catalog).** A public XML file at `https://data.cityofnewyork.us/download/kcrmj9hh/application/xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<museums>
  <museum>
    <name>American Folk Art Museum</name>
    <phone>212-265-1040</phone>
    <address>45 W. 53rd St. (at Fifth Ave.)</address>
    <closing>Closed: Monday</closing>
    <rates>admission: $9; seniors/students, $7; under 12, free</rates>
    <specials>
      Pay-what-you-wish: Friday after 5:30pm;
      refreshments and music available
    </specials>
  </museum>
  <museum>
    <name>American Museum of Natural History</name>
    <phone>212-769-5200</phone>
    <address>Central Park West (at W. 79th St.)</address>
    <closing>Closed: Thanksgiving Day and Christmas Day</closing>
```


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

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

XML is Everywhere (E.g. Office Suites)

▷ **Example 4.5.5 (MS Office uses XML).** The `MSOffice` suite and `LibreOffice` use compressed XML as an electronic document format.

1. Save a `MSOffice` file test.docx, add the extension .zip to obtain test.docx.zip.
2. Uncompress with unzip (`UNIX`) or open File Explorer, right-click ⇝ "Extract All" (`MSWindows`)
3. You obtain a folder with 15+ files, the content is in word/contents.xml

4. Other files have packaging information, images, and other objects.

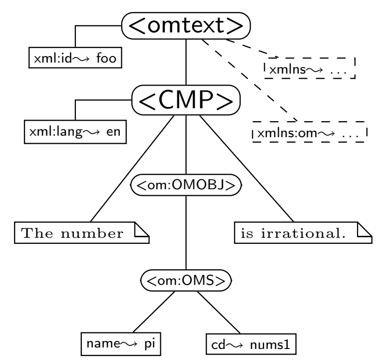
⚠ This is huge and offensively ugly.

- But you have everything you wanted and more
- In particular, you can process the contents via a program now.

XML Documents as Trees

- **Idea:** An XML Document is a Tree

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



- **Definition 4.5.6.** The XML document tree is made up of XML elements, attribute nodes, text nodes (and namespace declarations, comments,...)

XML Documents as Trees (continued)

- **Definition 4.5.7.** For communication this tree is serialized into a balanced bracketing structure, where
  - an inner XML element nodes is represented by the brackets <el> (called the opening tag) and </el> (called the closing tag),
  - the leaves of the XML tree are represented by empty element tags (serialized as <el></el>, which can be abbreviated as <el/>,
  - and text node (serialized as a sequence of UniCode characters).
  - An XML 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.

4.5.2 Computing with XML in Python

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

We use the external lxml library [LXMLa] in IWGS, even though the Python distribution includes the standard library ElementTree library [ET] for dealing with XML. lxml subsumes ElementTree and extends it by functionality for XPath and can parse a large set of HTML documents even though they are not valid XML. This makes lxml a better basis for practical applications in the Digital Humanities.

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

Computing with XML in Python (Elements)

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

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

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

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

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

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

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

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

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


Computing with XML in Python (Result)

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

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

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

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

The lxml API documentation [LXMLb] has the details.

This method of "manually" producing XML trees in memory by applying etree methods may seem very clumsy and tedious. But the power of lxml lies in the fact that these can be embedded in Python programs. And as always, programming gives us the power to do things very efficiently.

Computing with XML in Python (Automation)

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

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

produces a tree with 1000 children without much effort.

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

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

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

Computing with XML in Python (Attributes)

▷ Attributes can directly be added in the Element function

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

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

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

We can set them with the .set method:

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

This results in a changed element:

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


Recall that we could use Python dictionaries for iterating over in a for loop. We can do the same for attributes:

Computing with XML in Python (Attributes; continued)

- We can access attributes by the keys, values, and items methods, known from dictionaries:

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

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

- ⚠ To get a 'real' dictionary, use the attrib method (e.g. to pass around)

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

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

- ⚠ To get an independent snapshot of the attributes that does not depend on the XML tree, copy it into a dict:

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


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

This is also the reason why we spoke of "dictionary-like interface" to XML trees in lxml: dictionaries are immutable, while XML trees are not.

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

Computing with XML in Python (Text nodes)

- XML elements can contain text: we use the .text property to access and set it.

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


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

Case Study: Creating an HTML document

▷ We create nested html and body elements

```
>>> html = etree.Element("html")
>>> body = etree.SubElement(html, "body")
```

▷ Then we inject a text node into the latter using the .text property.

```
>>> body.text = "TEXT"
```

▷ Let's check the result

```
>>> etree.tostring(html)
b'<html><body>TEXT</body></html>'
```

▷ We add another element: a line break and check the result

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

▷ Finally, we can add trailing text via the .tail property

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


Note the use of the .tail property here? While the .text property can be used to set "all" the text in an XML element, we have to use the .tail property to add trailing text (e.g. after the
 element).
Notwithstanding the "Python power" argument from above, there are situations, where we just want to write down XML fragments and insert them into (programmatically created) XML trees. lxml as functionality for this: XML literals, which we introduce now.

Computing with XML in Python (XML Literals)

▷ **Definition 4.5.8.** We call any string that is well-formed XML an XML literal.

▷ We can use the XML function to read XML literals.

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

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

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

BTW, the fromstring function does the same.

▷ There is a variant html that also supplies the necessary HTML decoration.

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

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

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


4.5.3 XML Namespaces

We now come to a topic that is considered very difficult, confusing, and un-necessary by many people: XML namespaces. But it really is not, if you approach it with an open mind. Indeed it is probably what you would have come up with if you had been presented with the problem of mixing vocabularies, which is in turn a consequence of the fact that XML is used pervasively in the computing world and especially in Digital Humanities, where we often need to aggregate semi-structured data from multiple sources (and this multiple XML vocabularies).

XML is Everywhere (E.g. document metadata)

▷ **Example 4.5.9.** Open a PDF file in AcrobatReader, then click on

$$File \searrow DocumentProperties \searrow DocumentMetadata \searrow ViewSource$$

you get the following text: (showing only a small part)

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

▷ **Example 4.5.10.** Example 4.5.9 mixes elements from three different vocabularies:

- ▷ RDF: xmlns:rdf for the "Resource Descritpion Format",
- ▷ PDF: xmlns:pdf for the "Portable Document Format", and
- ▷ DC: xmlns:dc for the "Dublin Core" vocabulary

This is an excerpt from the document metadata which AcrobatDistiller 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 MSWord saves with each Word document should be in the same format as the metadata that Amazon saves with each of its book records, and again the same that the British library uses, etc.
We will now reflect what we have seen in Example 4.5.9 and fully define the namespacing mechanisms involved. Note that these definitions are technically involved, but conceptually quite natural. As a consequence they should be read more with an eye towards "what are we trying to achieve" than the technical details.

Mixing Vocabularies via XML Namespaces

- **Problem:** We would like to reuse elements from different XML vocabularies
  What happens if element names coincide, but have different meanings?
- **Idea:** Disambiguate them by vocabulary name. (prefix)
- **Problem:** What if vocabulary names are not unique? (e.g. different versions)
- **Idea:** Use a long string for identification and a short prefix for referencing
- **Definition 4.5.11.** An XML namespace is a string that identifies an XML vocabulary. Every element and attribute name in XML consists of a local name and a namespace.
- **Definition 4.5.12.** A namespace declaration is an attribute xmlns:⟪prefix⟫= whose value is an XML namespace n on an XML element e. The first associates the namespace prefix ⟪prefix⟫ with the namespace n in e: Then, any XML element in e with a prefixed name ⟪prefix⟫:⟪name⟫ has namespace n and local name ⟪name⟫.

  A default namespace declaration xmlns=d on an element e gives all elements in e whose name is not prefixed, the namepsace d.

  Namespace declarations on subtrees shadow the ones on supertrees.

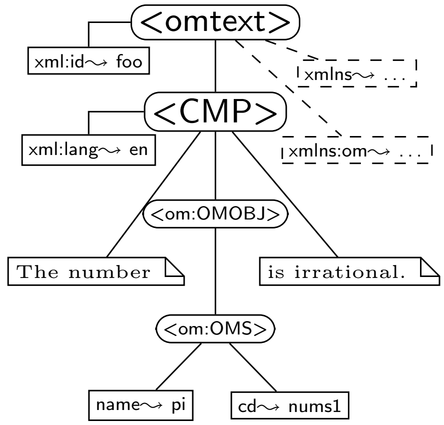
FAU : 144 2025-06-05

4.5.4 XPath: Specifying XML Subtrees

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

XPath, A Language for talking about XML Tree Fragments

- **Definition 4.5.13.** The XML path language (XPath) is a framework for specifying (sets of) fragments of XML trees by specifying paths from the root.
- **Intuition:** XPath is for trees what regular expressions are for strings.
- **Example 4.5.14.**



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

FAU : 145 2025-06-05

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

Computing with XML in Python (XPath)

▷ Say we have an XML tree:

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

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

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

▷ And we can do it again, ...

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

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

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

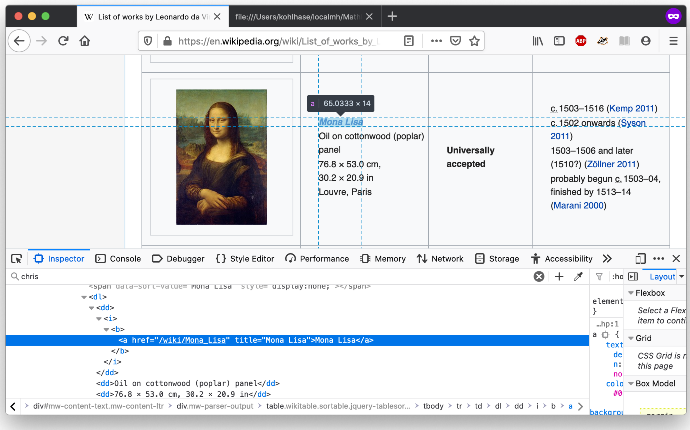
FAU : 146 2025-06-05

To see that XPath is not just a plaything, we will now look at at a typical example where we can identify useful subtrees in a large HTML document: the Wikipedia page on paintings by Leonardo da Vinci.

XPath Example: Scraping Wikipedia

▷ **Example 4.5.15 (Extracting Information from HTML).**

- ▷ We want a list of all titles of paintings by Leonardo da Vinci.
- ▷ open `https://en.wikipedia.org/wiki/List_of_works_by_Leonardo_da_Vinci` in FireFox. (save it into a file leo.html)
- ▷ call DOM inspector to get an idea of the XPath of titles. (bottom line)



The path is table > tbody > tr > td > dl > dd > i > b > a
Alternatively: right-click on highlighted line, ⇝ "copy" ⇝ "XPath", gives /html/body/div[3]/div[3]/div[4]/div/table[4]/tbody/tr[3]/td[2]/dl/dd/i/b/a.

- ▷ **Idea**: We want to use the second table cells td[2].
- ▷ Program it in Python using the lxml library: titles is list of title strings.

```python
from lxml import html

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


If the task of writing an XPath for extracting the 50+ titles from this page does not convince you as worth learning XPath for, consider that Wikipedia has ca. 30 such lists, which apparently have exactly the same tree structure, so the XPath developed once for da Vinci, probably works for all the others as well.

Chapter 5

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.

For that we discuss the basics of the World Wide Web as the client server architecture that enables such applications. Then we take up the contact form example to get an understanding how information is passed between client and server in interactive web pages. This motivates a discussion of server-side computation of web pages that can react to such information. A discussion of CSS styling shows how to make the web pages that are generated can be made visually appealing. We conclude the chapter by a discussion of client-side computation that allows making web pages interactive without recurring to the server. **Excursion:** The World Wide Web as we introduce it here is based on the Internet infrastructure and protocols. In some places it may be useful to read up on this insection A.1.

5.1 Web Applications: The Idea

Web Applications: Using Applications without Installing

▷ **Definition 5.1.1.** A web application (also called webapp) 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 5.1.2.** 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.

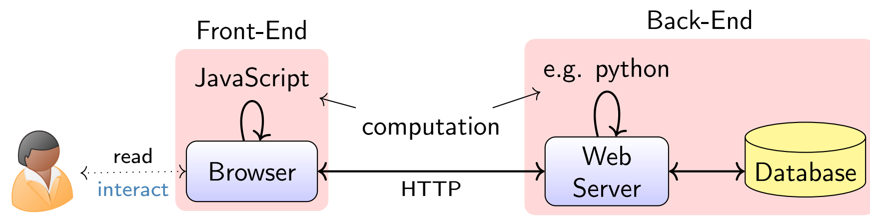
▷ **Common Traits:** Pages generated from databases and external feeds, content submission via HTML forms, file upload, dynamic HTML.

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

Anatomy of a Web Application

▷ **Definition 5.1.3.** A web application consists of two parts:

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



Both parts rely on (separate) computational facilities.
A database as a persistence layer is optional.

▷ **Note:** The web browser, web server, and database can

- ▷ be deployed on different computers, (high throughput)
- ▷ all run on your laptop (e.g. for development)

FAU : 149 2025-06-05

To understand web applications, we will first need to understand

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

5.2 Basic Concepts of the World Wide Web

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

5.2.1 Preliminaries

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

The Internet and the Web

▷ **Definition 5.2.1.** The Internet is a global computer network that connects hundreds of thousands of smaller networks.

▷ **Definition 5.2.2.** The World Wide Web (WWW) is an open source information space where electronic documents and other web resources are identified by URLs, interlinked by hypertext links, and can be accessed via the Internet.

▷ **Intuition:** The WWW is the multimedia part of the internet, they form critical infrastructure for modern society and commerce.

▷ The internet/WWW is huge:

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

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

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

Concepts of the World Wide Web

▷ **Definition 5.2.3.** A web page is a document on the WWW that can include multimedia data and hyperlinks.

▷ **Note:** Web pages are usually marked up in in HTML.

▷ **Definition 5.2.4.** A web site is a collection of related web pages usually designed or controlled by the same individual or organization.

▷ A web site generally shares a common domain name.

▷ **Definition 5.2.5.** A hyperlink is a reference to data that can immediately be followed by the user or that is followed automatically by a user agent.

▷ **Definition 5.2.6.** A collection text documents with hyperlinks that point to text fragments within the collection is called a hypertext. The action of following hyperlinks in a hypertext is called browsing or navigating the hypertext.

▷ In this sense, the WWW is a multimedia hypertext.

5.2.2 Addressing on the World Wide Web

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

Uniform Resource Identifier (URI), Plumbing of the Web

▷ **Definition 5.2.7.** 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 components:

- ▷ 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, and
- ▷ a fragment identifier in the resource.

▷ **Example 5.2.8.** 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 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 5.2.9.** URIs can be abbreviated to relative URIs; missing parts are filled in from the context.

▷ **Example 5.2.10.** Relative URIs are more convenient to write

relative URI	abbreviates	in context
#foo	⟪current − file⟫#foo	curent file
bar.txt	file:///home/kohlhase/foo/bar.txt	file system
../bar/bar.html	http://example.org/bar/bar.html	on the web

▷ **Definition 5.2.11.** 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/kohlhase/foo/foo.html – and for the web example via the URI //example.org/foo/foo.html. Note that in the last example, the relative URI ../bar/ goes up one segment of the path component (that is the meaning of ../), and specifies the file bar.html in the directory bar.

But relative URIs have another advantage over absolute URIs: they make a web page or web site easier to move. If a web site only has links using relative URIs internally, then those do not mention e.g. authority (this is recovered from context and therefore variable), so we can freely move the web-site e.g. between domains.
Note that some forms of URIs can be used for actually locating (or accessing) the identified resources, e.g. for retrieval, if the resource is a document or sending to, if the resource is a mailbox. Such URIs are called "uniform resource *locators*", all others "uniform resource *locators*".

Uniform Resource Names and Locators

- ▷ **Definition 5.2.12.** 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 name (URN).
- ▷ **Idea:** A URN defines the identity of a resource, a URL provides a method for finding it.
- ▷ **Example 5.2.13.**
  The following URI is a URL (try it in your browser)
  http://kwarc.info/kohlhase/index.html
- ▷ **Example 5.2.14.** urn:isbn:978—3—540—37897—6 only identifies [Koh06] (it is in the library)
- ▷ 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 5.2.15.* URIs are ASCII strings.
- ▷ **Problem:** This is awkward e.g. for "*France Télécom*", worse in Asia.
- ▷ **Solution?:** Use unicode! (no, too young/unsafe)
- ▷ **Definition 5.2.16.** Internationalized resource identifiers (IRIs) extend the ASCII-based URIs to the universal character set.
- ▷ **Definition 5.2.17.** URI-encoding maps non-ASCII characters to ASCII strings:

1. Map each character to its UTF-8 representation.
2. Represent each byte of the UTF-8 representation by three characters.
3. The first character is the percent sign (%),
4. and the other two characters are the hexadecimal representation of the byte.

URI-decoding is the dual operation.

▷ **Example 5.2.18.** The letter "ł" ($\mathrm{U}+142$) would be represented as `%C5%82`.

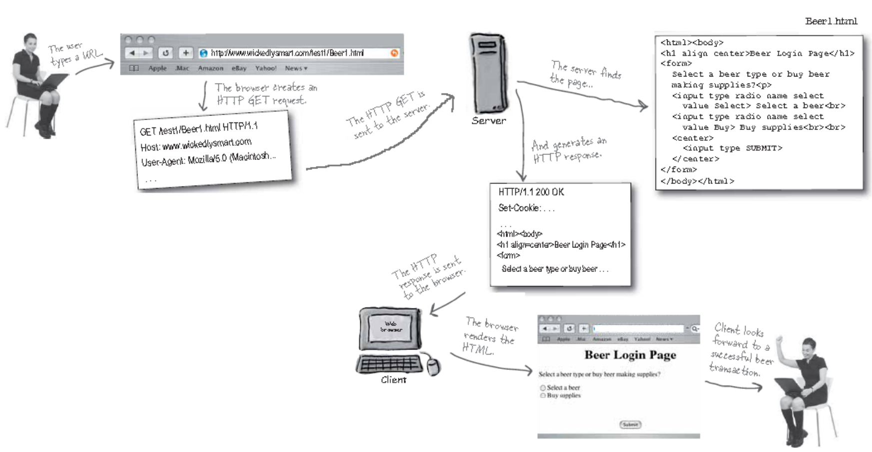
▷ **Example 5.2.19.** `http://www.Übergrößen.de` becomes `http://www.%C3%9Cbergr%C3%B6%C3%9Fen.de`

▷ *Remark 5.2.20.* Your browser can still show the URI-decoded version (so you can read it)

5.2.3 Running the World Wide Web

The infrastructure of the WWW 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 HTTPs and HTTPSs protocols. We give an overview via a concrete example before we go into details.

The World Wide Web as a Client/Server System



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

HTTP: Hypertext Transfer Protocol

▷ **Definition 5.2.21.** 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].

▷ **Preview/Recap:** HTTP is used by a client (called user agent) to access web web resources (addressed by uniform resource locators (URLs)) via a HTTP request. The web server answers by supplying the web resource (and metadata).

▷ **Definition 5.2.22.** Most important HTTP request methods. (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 5.2.23.** We call a HTTP request safe, iff it does not change the state in the web server. (except for server logs, counters,. . . ; no side effects)

▷ **Definition 5.2.24.** We call a HTTP request idempotent, iff executing it twice has the same effect as executing it once.

▷ HTTP is a stateless protocol. (very memory efficient for the server.)

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

Web Servers

▷ **Definition 5.2.25.** A web server is a network program (a server in a client server architecture of the WWW) that delivers web resources to and receives content from clients via the Hypertext Transfer Protocol (HTTP).

▷ **Example 5.2.26 (Common Web Servers).**

- ▹ `apache` is an open source web server that serves about 50% of the WWW.
- ▹ `nginx` is a lightweight open source web server. (ca. 35%)
- ▹ `IIS` is a proprietary web server provided by Microsoft Inc.

▷ **Definition 5.2.27.** A web server can host – i.e serve web resources for multiple domains (via configurable hostnames) 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 `MSWindows` [Xam].

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

Example: An HTTP request in real life

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

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

▷ The response from the server

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

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

▷ **Note:** As you can seen, these are clear-text messages that go over an unprotected network. A consequence is that everyone on this network can intercept this communication and see what you are doing/reading/watching.

5.3 Recap: HTML Forms Data Transmission

EdN:2 The first two requirement for web applications above are already met by HTML in terms of HTML forms (see slide 120 ff.). Let us recap and extend[2]

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 5.3.1.** Forms contain input fields and explanations.

```
<form name="input" action="login.html" method="get">
  Username: <input type="text" name="user"/>
  Password: <input type="password" name="pass"/>
  <input type="submit" value="Submit"/>
</form>
```

yields the following in a web browser:

[2]EdNote: continue

Pressing the submit button activates a HTTP GET request to the URL login.html?user=⟨⟨name⟩⟩&pass=⟨⟨passwd⟩⟩

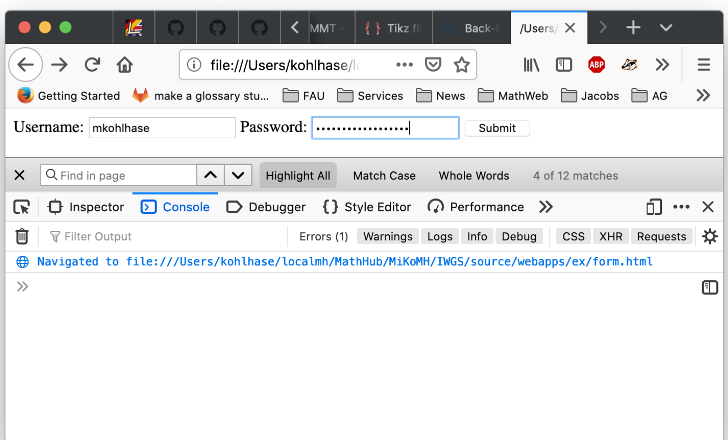
▷ ⚠ Never use the GET method for submitting passwords (see below)

We can now use the tools any modern browser supplies to check up on this claim. In fact, using the browser tools is essential for advanced web development. Here we use the web console, that monitors any activity, to check upon what really happens when we interact with the web page.

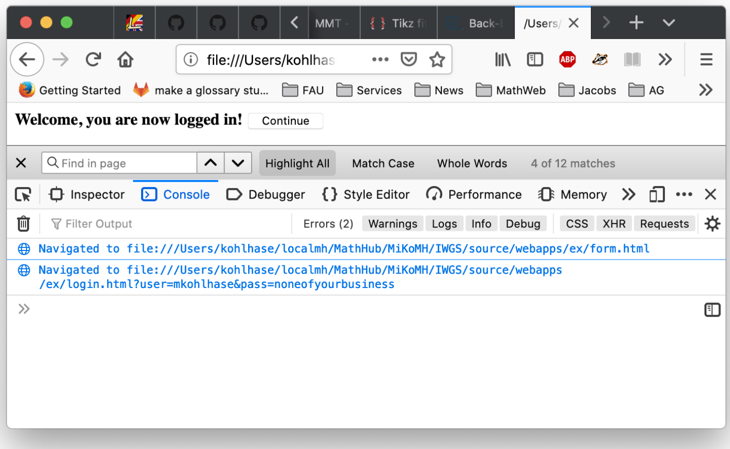
Checking up on the Transmission

▷ Let's verify the claims above using browser tools (here the web console)

▷ Loading the file and filling in the form: (console logs file URI)



▷ After submitting the form: (console logs the HTTP request)



FAU : 161 2025-06-05

A side effect of re-playing our development in the browser is that we see another type of input field: A password field, which hides user input from un-authorized eyes. We also see that the GET request incorporates the form data which contains the password into the URI of the request, which is visible to everyone on the web. We will come back to this problem later.

Let us now look at the data transmission mechanism in more detail to see what is actually transmitted and how.

HTML Forms and Form Data Transmission

- ▷ We specify the HTTP communication of HTML forms in detail.
- ▷ **Definition 5.3.2.** The HTML form element groups the layout and input elements:
  - ▷ <form action="⟪URI⟫" method="⟪req⟫"> specifies the form action in terms of a HTTP request ⟪req⟫ to the URI ⟪URI⟫.
  - ▷ The form data consists of a string ⟪data⟫ of the form $n_1{=}v_1\&\cdots\&n_k{=}v_k$, where
    - ▷ n_i are the values of the name attributes of the input fields
    - ▷ and v_i are their values at the time of submission.
  - ▷ <input type="submit" .../> triggers the form action: it composes a HTTP request
    - ▷ If ⟪req⟫ is get (the default), then the browser issues a GET request ⟪URI⟫?⟪data⟫.
    - ▷ If ⟪req⟫ is post, then the browser issues a POST request to ⟪URI⟫ with document content ⟪data⟫.
- ▷ We now also understand the form action, but should we use GET or POST.

FAU : 162 2025-06-05

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

▷ **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.

5.4 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 5.4.1.** A server-side scripting framework is a web server extension that generates web pages upon HTTP requests.

▷ **Example 5.4.2.** `perl` is a scripting language with good string manipulation facilities. PERL CGI is an early server-side scripting framework based on this.

▷ **Example 5.4.3.** Python is a scripting language with good string manipulation facilities. And bottle WSGI is a simple but powerful server-side scripting framework based on this.

▷ **Observation:** 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:** A server-side scripting framework solves two problems:

1. making the development of functionality that generates HTML pages convenient and efficient, usually via a template engine, and
2. binding such functionality to URLs the routes, we call this routing.

We will look at the second problem: routing first. There is a dedicated Python library for that.

5.4.1 Routing and Argument Passing in Bottle

We wil now introduce the bottle library, which supplies a lightweight web server and server-side scripting framework implemented in Python. It is already installed on the JuptyerLab cloud IDE at `http://jupyter.kwarc.info`. To install it on your laptop, just type pip install bottle in a shell.

The Web Server and Routing in Bottle WSGI

- ▷ **Definition 5.4.4.** Serverside routing (or simply routing) is the process by which a web server connects a HTTP request to a function (called the route function) that provides a web resource. A single URI path/route function pair is called a route.
- ▷ The bottle WSGI library supplies a simple Python web server and routing.
  - ▷ The run(⟪keys⟫) function starts the web server with the configuration in ⟪keys⟫.
  - ▷ The @route decorator connects path components to Python function that return strings. Decorators change functions. A decorator @route(⟪path⟫) augments the following function f to answer to HTTP requests to the ⟪path⟫ and return f's return value.
- ▷ **Example 5.4.5 (A Hello World route).** . . . for localhost on port 8080

```
from bottle import route, run

@route('/hello')
def hello():
    return "Hello IWGS!"

run(host='localhost', port=8080, debug=True)
```

This web server answers to HTTP GET requests for the URL `http://localhost:8080/hello`

Let us understand Example 5.4.5 line by line: The first line imports the library. The second establishes a route with the name hello and binds it to the Python function hello in line 3 and 4. The last line configures the bottle web server: it serves content via the HTTP protocol for localhost on port 8080.

So, if we run the program from Example 5.4.5, then we obtain a web server that will answer HTTP GET requests to the URL `http://localhost:8080/hello` with a HTTP answer with the content Hello IWGS!.

To keep the example simple, we have only returned a text string; A realistic application would have generated a full HTML page (see below).

In the last line of Example 5.4.5, we have also configured the bottle web server to use "debug mode", which is very helpful during early development.

In this mode, the bottle web server is much more verbose and provides helpful debugging information whenever an error occurs. It also disables some optimisations that might get in your way and adds some checks that warn you about possible misconfiguration.

Note that debug mode should be disabled in a production server for efficiency.

But we can do more with routes!

Dynamic Routes in Bottle

▷ **Definition 5.4.6.** A dynamic route is a route annotation that contains named wildcards, which can be picked up in the route function.

▷ **Example 5.4.7.** Multiple @route annotations per route function f are allowed ⇝ the web application uses f to answer multiple URLs.

```
@route('/')
@route('/hello/<name>')
def greet(name='Stranger'):
    return (f'Hello {name}, how are you?')
```

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

Concretely: A HTTP GET request to

- ▹ `http://localhost` is answered with Hello Stranger, how are you?.
- ▹ `http://localhost/hello/MiKo` is answered with Hello MiKo, how are you?.

Requests to e.g `http://localhost/hello` or `http://localhost/hello/prof/kohlhase` lead to errors. (404: not found)

Often we want to have more control over the routes. We can get that by filters, which can involve data types and/or regular expressions.

Restricting Dynamic Routes

▷ **Definition 5.4.8.** A dynamic route can be restricted by a route filter to make it more selective.

▷ **Example 5.4.9 (Concrete Filters).** We use :int for integers and :re:⟪regex⟫ for regular expressions

```
@route('/tel/<id:int>') # local number
@route('/tel/<num:re:^\+[1−9]{1}[0−9]{3,14}$>') # international
```

Different route filters allow to classify paths and treat them differently.

▷ **Note:** Multiple named wildcards are also possible, in a dynamic route; with and without filters

▷ **Example 5.4.10 (A route with two wildcards).**

```
@route('/<action>/<user:re:[a−z]+>') # matches /follow/miko
def user_api(action, user):
    ...
```


We have already seen above that we want to use HTTP GET and POST request for different facets of transmitting HTML form data to the web server. This is supported by bottle WSGI in two ways: we can specify the HTTP method of a route and we have access to the form data (and other aspects of the request).

Method-Specific Routes: HTTP GET and POST

- **Definition 5.4.11.** The @route decorator takes a method keyword to specify the HTTP request method to be answered. (HTTP GET is the default)
  - @get(⟨⟨path⟩⟩) abbreviates @route(⟨⟨path⟩⟩,method="GET")
  - @post(⟨⟨path⟩⟩) abbreviates @route(⟨⟨path⟩⟩,method="POST")
- **Example 5.4.12 (Login 1).** Managing logins with HTTP GET and POST.

```
from bottle import get, post, request # or route

@get('/login') # or @route('/login')
def login():
    return '''
        <form action="/login" method="post">
            Username: <input name="username" type="text" />
            Password: <input name="password" type="password" />
            <input value="Login" type="submit" />
        </form>
    '''
```

- **Note:** We can also have a POST request to the same path; we use that for handling the form data transmitted by the POST action on submit. (up next)

Recall that we have already seen most of this in slide 160. The only new thing is that we return the HTML as a string in the route function as a request to a HTTP GET request. Now comes the interesting part: the form uses the POST method in the form action and we have to specify a route for that. Recall from ??? that this allows for encrypted transmission, so we are less naive than our solution from slide 160.

Bottle Request: Dealing with POST Data

- **Recall:** from a HTML form we get a GET or POST request with form data $n_1{=}v_1\&\cdots\&n_k{=}v_k$ (here user=mkohlhase&login=noneofyourbusiness)
- Bottle WSGI provides the request object for dealing with HTTP request data.
- **Example 5.4.13 (Login 2).** Continuing from Example 5.4.12: we parse the request transmitted request and check password information:

```
@post('/login') # or @route('/login', method='POST')
def do_login():
    username = request.forms.get('username')
    password = request.forms.get('password')
    if check_login(username, password):
        return "<p>Your login information was correct.</p>"
    else:
        return "<p>Login failed.</p>"
```

We assume a Python function check_login that checks authentication credential and authenticator, and keeps a list of logged in users.

The main new thing in Example 5.4.13 is that we use the request.forms.get method to query the request object that comes with the HTTP request triggering the route for the form data.

5.4.2 Templating in Python via STPL

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

What would we do in Python

- **Example 5.4.14 (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 5.4.15 (Python Solution).** . . . use Python functions:

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

- **Problem 2:** If HTML markup dominates, want to use a HTML editor (mode),
  - e.g. for HTML syntax highlighting/indentation/completion/checking
- **Idea:** Embed program snippets into HTML. (only execute these, copy rest)

We will now formalize and toolify the idea of "embedding code into HTML". What comes out of this idea is called "templating". It exists in many forms, and in most programming languages.

Template Processing for HTML

- **Definition 5.4.16.** A template engine (or template processor) for a document format F is a program that transforms templates, i.e. strings or files (a template file) ith a mixture of program constructs and F markup, into a F strings or F documents by executing the program constructs in the template (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:** We can turn a template engine into a server-side scripting framework by employing the URIs of template files on a server as routes and extending the web server by template processing.
- **Example 5.4.17.** PHP (originally "Programmable Home Page Tools") is a very successful server-side scripting framework following this model.

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 5.4.18.** Bottle WSGI supplies the template engine stpl (Simple Template Engine) that processes the STPL (Simple Template Language) format. (documentation at [STPL])
- **Definition 5.4.19.** A template engine for a document format F is a program that transforms templates, i.e. strings or files through a mixture of program constructs and F markup, into F-strings or F-documents by executing the program constructs in the template (template processing).
- stpl uses the template function for template processing and {{...}} to embed program objects into a template; it returns a formatted unicode string.

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

>>> my_dict={'number': '123', 'street': 'Fake St.', 'city': 'Fakeville'}
>>> template('I live at {{number}} {{street}}, {{city}}', **my_dict)
u'I live at 123 Fake St., Fakeville'
```


The stpl template function is a powerful enabling basic functionality in Python, but it does not satisfy our goal of writing "HTML with embedded Python". Fortunately, that can easily be built on top of the template functionality:

stpl Syntax and Template Files

- **But what about...:** HTML files with embedded Python?
- stpl uses template files (extension .tpl) for that.
- **Definition 5.4.20.** 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 5.4.21.** 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 to create, the more fragments of the template files we want to reuse in multiple places – with and without adaptions to the particular use case.

Template Functions

▷ **Definition 5.4.22.** 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⟫): return the value of ⟪var⟫, or ⟪default⟫.
4. setdefault(⟪name⟫,⟪val⟫)

▷ **Example 5.4.23 (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 5.4.24 (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
```


5.4.3 Completing the Contact Form

We are now equipped to finish the contact form example

We now come back to our worked HTML example: the contact form from above. Here is the current state:

Back to our Contact Form (Current State)

▷ A contact form and message receipt (communicate via HTTPs request)

contact4.html

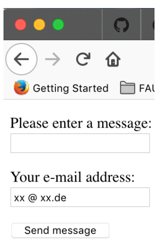
```
<title>Contact</title>
<form action="contact-after.html">
  <h2>Please enter a message:</h2>
    <input name="msg" type="text"/>
  <h3>Your e-mail address:</h3>
  <input name="addr" type="text"
         value="xx @ xx.de"/>
  <br/>
  <input type="submit"
         value="Send message"/>
</form>
```

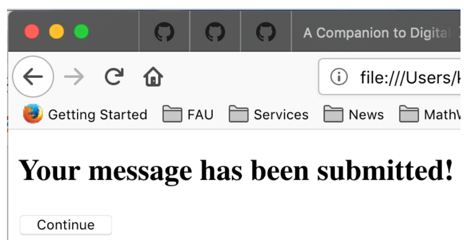
```
GET contact-after.html?
    msg=Hi;addr=foo@bar.de
```

contact-after.html

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

```
GET contact.html
```





▷ **Problem:** The answer is a static HTML document independent of form data.

▷ **Solution:** Generate the answer programmatically using the form data. (up next)

There are two great flaws in the current state of the contact form:

1. The "receipt page" contact-after.html is static and does not take the data it receives from the contact form into account. It would be polite to give some record on what happened. We can fix this using bottle WSGI using the methods we just learned.

2. Nothing actually happens with the message. It should be either entered into an internal message queue in a database0 or ticketing system, or fed into an e-mail to a sales person. As we do not have access to the first, we will just use a Python library to send an e-mail programmatically.

Completing the Contact Form

▷ bottle WSGI has functionality (request.GET and request.POST) to decode the form data from a HTTP request. (so we do not have to worry about the details)

▷ **Example 5.4.25 (Submitting a Contact Form).** We use a new route for contact-form-after.html with a corresponding template file:

contact.py

```
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)

run(host="localhost", port=8080)
```

contact−after.tpl

```
<p>Message submitted!</p>
<table>
  <tr>
    <td>Return Address:</td>
    <td>{{addr}}</td>
  </tr>
  <tr>
    <td>Message Sent:</td>
    <td>{{msg}}</td>
  </tr>
</table>
```


Fortunately, the only remaining part: actually sending off an e-mail to the specified mailbox is very easy: using the smtplib library we just create an e-mail message object, and then specify all the components.

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 $\widehat{=}$ Simple Mail Transfer Protocol”)

▷ **Example 5.4.26 (Continuing).**

```
import smtplib
from email.message import EmailMessage

def send−contact−email (addr, text)
    msg = EmailMessage()
    msg.set_content(text)
    msg['Subject'] = 'Contact Form Result'
    msg['From'] = info@example.org
    msg['To'] = addr
    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”)

Once we have the e-mail message object msg, we open a “SMTP connection” s send the message via its send_message method and close the connection by s.quit()). Again, the Python library hides all the gory details of the SMTP protocol.

Chapter 6

Front-end Technologies

We introduce three important concepts for building modern web front-ends for web applications:

1. Client-side computation: manipulating the browser DOM via JavaScript.
2. Cascading Stylesheets (CSS) for styling the layout of HTML (and XML).
3. The jQuery library: a symbiosis of JavaScript and CSS ideas to make JavaScript coding easier and more efficient.

6.1 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 document.

We will sketch the WWW solution to this in the following.

To understand client-side computation, we first need to understand the way browsers render HTML pages.

Background: Rendering Pipeline in browsers

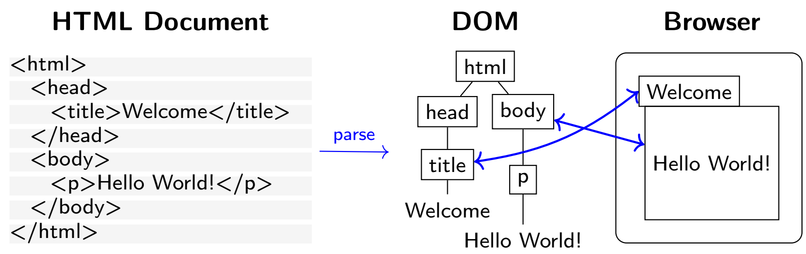
▷ **Observation:** The nested markup codes turn HTML documents into trees.

▷ **Definition 6.1.1.** 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 is the purpose of the client side scripting language, which we will go into next.

6.1.1 JavaScript in HTML

Dynamic HTML

- ▷ **Definition 6.1.2.** We call a web page dynamic, if its presentation can change without the web browser loading new content.
- ▷ **Idea:** Generate parts of the web page dynamically by manipulating the DOM.
- ▷ **Definition 6.1.3.** JavaScript is an object-oriented scripting language mostly used to enable programmatic access to the DOM in a web browser.
- ▷ JavaScript is standardized by ECMA in [Ecm].
- ▷ **Example 6.1.4.** We write the some text into HTML DOM.

```
<html>
 <head>
  <script type="text/javascript">document.write("Dynamic HTML!");</script>
 </head>
 <body><!-- nothing here; will be added by the script later --></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.

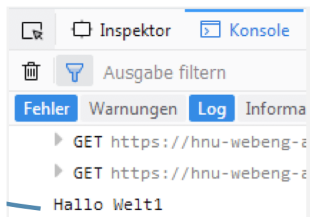
The example above already shows a JavaScript command: document.write, which replaces the content of the <body> element with its argument – this is only useful for testing and debugging purposes.

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).
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: 1

▷ **Example 6.1.5 (Logging to the browser console).**

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



: 180 2025-06-05

The function console.log writes its argument into the console of the web browser.
It is primarily used for debugging the source code of a web page.

Example 6.1.6. If we want to know whether a function square has been executed we add calls to console.log like this:

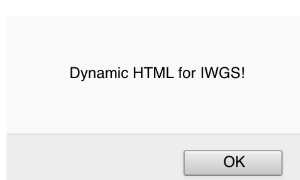
```
function square (n) {
  console.log ("entered function square with argument " + n);
  return (n * n);
  console.log ("exited function square with result " + n * n);
  }
```

In the console we can check whether the content contains e.g. entered function square and moreover whether argument and value are as expected.

Browser-level JavaScript functions: 2

▷ **Example 6.1.7 (Raising a Popup).**

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

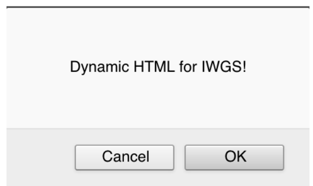


The function alert creates a popup that contains the argument.

Browser-level JavaScript functions: 3

▷ **Example 6.1.8 (Asking for Confirmation).**

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



The function confirm creates a popup that contains the argument and a confirmation/cancel button pair and returns the corresponding Boolean value.
If the user clicks on the confirmation button, the returned value will be false and true for the cancel button.

Example 6.1.9. You can play with this in the following frizzle:

```
<html>
<head>
    <title>confirm</title>
    <script src="./client-js/jquery-3.6.4.min.js" type="application/javascript"></script>
    <style>
        .emph{
            color: blue;
        }
        .code{
            font-size: 110%;
        }
    </style>
</head>
<body>
    <h2>Live Demo of the JavaScript <span class="code emph">confirm</span> Function</h2>

    <textarea id="output" style="width:400px">
    </textarea>
    <textarea id="code" style="width:400px; height:400px">
    </textarea>
    <p>
      Click <button onclick="openPopup()">here</button> to execute
      the <span class="code">confirm</span> function again!
    </p>
    <p>
        Show <button onclick="showCode()">source code</button>
    </p>

    <script type="application/javascript">
        function openPopup(){
            console.log("executed openPopup function");
            var output="";
```

```
            var returnValue=confirm ("Hello World!");
            if(returnValue==true){
                output="You clicked the OK button!" + "(return value: " + returnValue + ")";

            } else {
                output="You clicked the Cancel button!" + "(return value: " + returnValue
+ ")";
            }
            console.log(output);
            $("#output").html(output);
            $("p").show();
        }
        openPopup();

        function showCode(){
            console.log("executed showCode function");
            var func=openPopup.toString();
            //alert(func);
            $("#code").html(func);
        }
    </script>
</body>
</html>
```

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 attribute:

```
<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')"/>
```

FAU : 183 2025-06-05

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:..."/>
```


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 <a href="⟪URI⟫"> 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 <img> and stylesheets may be not yet loaded. load: the browser loaded all resources (images, styles etc); beforeunload / unload: when the user is leaving the page.

5. resource loading events; onload: successful load, onerror: an error occurred.

Let us now use all we have learned in an example to fortify our intuition about using JavaScript to change the DOM.

Example: Changing Web Pages Programmatically

▷ **Example 6.1.10 (Stupid but Fun).**

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

<script type="text/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 6.1.10 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 6.1.10. 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.

6.2 Cascading Stylesheets

In this section we introduce a technology of digital documents which naturally belongs into chapter 4: the specification of presentation (layout, colors, and fonts) for marked-up documents.

6.2.1 Separating Content from Layout

As the WWW evolved from a hypertext system purely aimed at human readers to a Web of multimedia documents, where machines perform added-value services like searching or aggregating, it became more important that machines could understand critical aspects web pages. One way to facilitate this is to separate markup that specifies the content and functionality from markup that specifies human-oriented layout and presentation (together called "styling"). This is what "cascading style sheets" set out to do.

Another motivation for CSS is that we often want the styling of a web page to be customizable (e.g. for vision impaired readers).

CSS: Cascading Style Sheets

- ▷ **Idea:** Separate structure/function from appearance.
- ▷ **Definition 6.2.1.** Cascading Style Sheets (CSS) is a style language that allows authors and users to attach style (e.g., fonts, colors, and spacing) to HTML and XML documents.
- ▷ **Example 6.2.2.** Our text file from ??? 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 6.2.3.** A CSS style sheet consists of a sequence of rules that in turn consist of a set of selectors that determine which XML elements the rule applies to and a declaration block that specifies intended presentation.
- ▷ **Definition 6.2.4.** A CSS declaration block consists of a semicolon separated list of declarations in curly braces. Each declaration itself consists of a property, a colon, and a value.
- ▷ **Example 6.2.5.** In Example 6.2.2 we have three rules, they address color and font properties:

```
       body {background-color:#d0e4fe;}
       h1 {color:orange;
             text-align:center;}
       p {font-family:"Verdana";
```

- ▷ **Observation:** In modern web sites, CSS contributes as much – if not more – to the appearance as the choice of HTML elements.

In Example 6.2.5 the selectors are just element names, they specify that the respecive declaration blocks apply to all elements of this name.

We explore this new technology by way of an example. We rework the title box from the HTML example above – after all treating author/affiliation information as headers is not very semantic. Here we use div and span elements, which are generic block-level (i.e. paragraph-like) and inline containers, which can be styled via CSS classes. The class titlebox is represented by the CSS selector .titlebox.

A Styled HTML Title Box (Source)

- ▷ **Example 6.2.6 (A style Title Box).** The HTML source:

```
  <head>
    <title>A Styled HTML Title</title>
    <link rel="stylesheet" type="text/css" href="style.css"/>
```

```
  </head>
  <body>
    <div class="titlebox">
      <div class="title">Anatomy of a HTML Web Page</div>
      <div class="author">
        <span class="name">Michael Kohlhase</span>
        <span class="affil">FAU Erlangen−Nuernberg</span>
      </div>
    </div>
    ...
```

And the CSS file referenced in the **<link>** element in line 3:

```
.titlebox {border: 1px solid black;padding: 10px;
           text−align: center
           font−family: verdana;}
.title {font−size: 300%;font−weight: bold}
.author {font−size: 160%;font−style: italic;}
.affil {font−variant: small−caps;}
```


And here is the result in the browser:

A Styled HTML Title Box (Result)

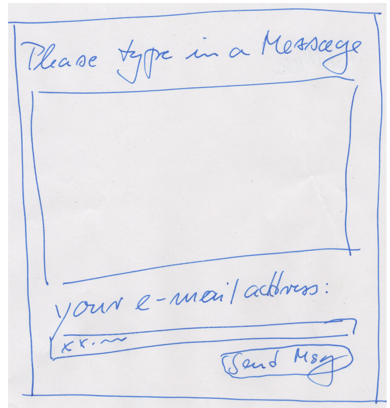
6.2.2 Worked Example: The Contact Form

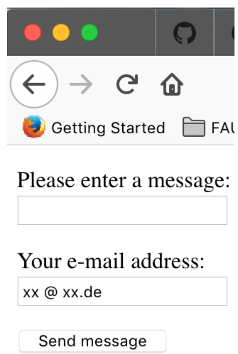
To fortify our intuition on CSS, we take up the "contact form" example from above and improve the layout in a step-by-step process concentrating on one aspect at a time.

CSS in Practice: The Contact Form Example (Continued)

▷ Recap: The unstyled contact form – Dream vs. Reality

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

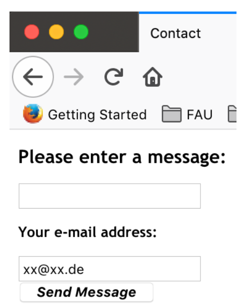




▷ Add a CSS file with font information

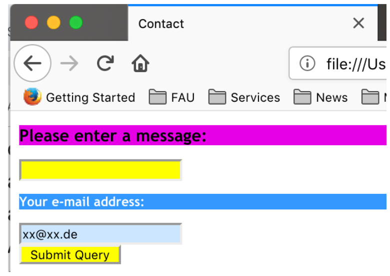
```
<link rel="stylesheet" type="text/css"
      href="csscontact1.css" />
  <input class="important" type="submit"
         value="Send Message"/>
```

```
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 (ooops, what about the size)

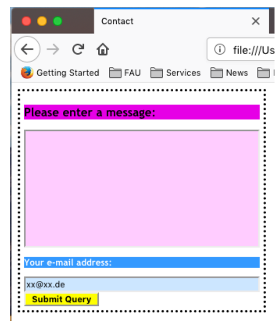
```
<h2>Please enter a message:</h2>
<h3>Your e−mail address:</h3>
<input class="important" name="addr"
      style="background−color:#cce6ff"
      type="text" value="xx@xx.de"/>
```




```
h2 {background−color: #e600e6;}
h3 {background−color: #3399ff;
    color: white;}
input{background−color:yellow}
```

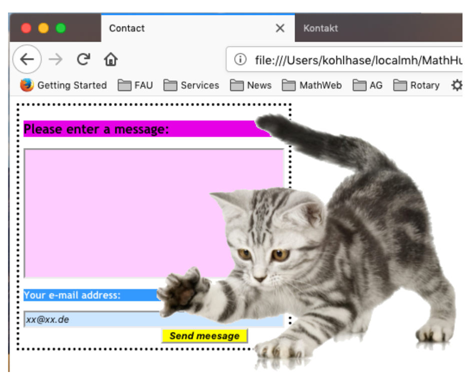
▷ Add size information and a dotted frame

```
<form action="contact−after.html"
      style="width:8cm;border:dotted;padding:5px">
  <h2>Please enter a message:</h2>
  <input name="msg" type="text"
         style="height:4cm;width:8cm;
                background−color:#ffccff"/>
  <br/>
  <h3>Your e−mail address:</h3>
  <input class="important" name="addr"
         type="text"
         value="xx@xx.de" style="width:8cm;
                background−color:#cce6ff"/>
```



▷ Add a cat that plays with the submit button (because we can)

```
<img id="cat" src="cat.png"
     style="position:absolute;
            left:170px;top: 15px;
            width=300px"/>
```



FAU : 190 2025-06-05

This worked example should be enough to cover most layout needs in practice. Note that in most use cases, these generally layout primitives will have to be combined in different and may be even new ways.

Actually, the last “improvement” may have gone a bit overboard; but we used it to show how absolute positioning of images (or actually any CSS boxes for that matter) works in practice.

6.2.3 A small but useful Fragment of CSS

CSS is a huge ecosystem of technologies, which is spread out over about 100 particular specifications – see [CSSa] for an overview.

We will now go over a small fragment of CSS that is already very useful for web applications in more detail and introduce it by example. For a more complete introduction, see e.g. [CSSc].

Recall that selectors are the part of CSS rules that determine what elements a rule affects. We now give the most important cases for our applications.

CSS Selectors

- **Question:** Which elements are affected by a CSS rule?
- Elements of a given name (optionally with given attributes)
  - Selectors: name $\widehat{=}$ ⟨⟨elname⟩⟩, attributes $\widehat{=}$ [⟨⟨attname⟩⟩=⟨⟨attval⟩⟩]
- **Example 6.2.7.** p[xml:lang='de'] applies to <**p** xml:**lang**="de">...</**p**>
- Any element with a given class attributes
  - Selector: .⟨⟨classname⟩⟩
- **Example 6.2.8.** .important applies to <⟨⟨el⟩⟩ class='important'>...</⟨⟨el⟩⟩>
- The element with a given id attribute
  - Selector: #⟨⟨id⟩⟩
- **Example 6.2.9.** #myRoot applies to <⟨⟨el⟩⟩ id='myRoot'>...</⟨⟨el⟩⟩>
- **Note:** Multiple selectors can be combined in a comma separated list.
- For a full list see `https://www.w3schools.com/cssref/css_selectors.asp`.

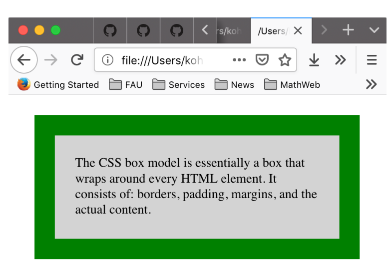
We now come to one of the most important conceptual parts of CSS: the box model. Understanding it is essential for dealing with CSS based layouts.

The CSS Box Model

- **Definition 6.2.10.** 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;
}
```

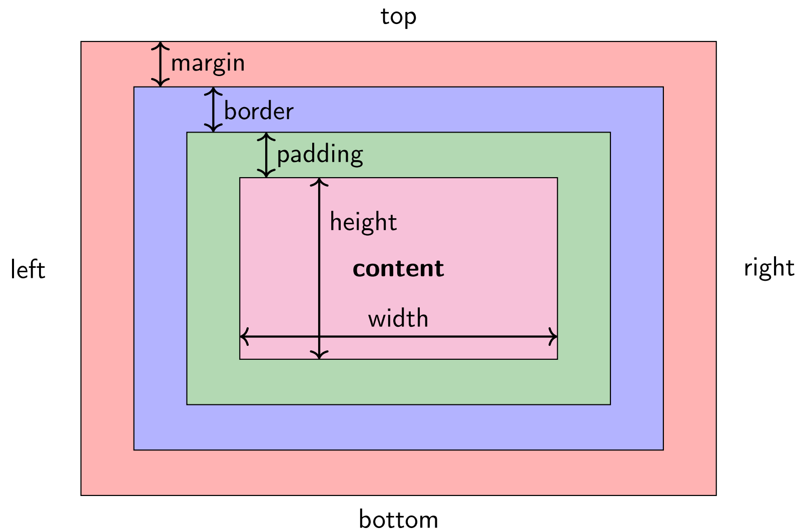


Note that the overall width of the CSS box is $300 + 2 \cdot 3 \cdot 25 = 450$ pixels.

As a summary of the above, we can visualize the CSS box model in a diagram:

The CSS Box Model: Diagram

▷ **Definition 6.2.11.** 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 stylesheets. 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 priorization: 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, priorization takes care of them in an integrated fashion.

Cascading of selectors in CSS: Priorization

▷ Multiple CSS selectors apply with the following priorizations:

1. important (i.e. marked with !important) before unimportant
2. inline (specified via the style attribute)
3. media-specific rules before general ones
4. user-defined CSS stylesheet (e.g. in the `FireFox` profile)
5. specialized before general selectors (complicated; see e.g. [CSSb])
6. rule order: later before earlier selectors
7. parent inheritance: unspecified properties are inherited from the parent.

8. Style sheet included or referenced in the HTML document.
9. browser default

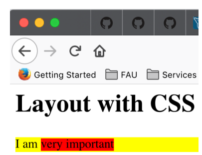
But do not despair with this technical specification, you do not have to remember it to be effective with CSS practically, because the rules just encode very natural "behavior". And if you need to understand what the browser – which implements these rules – really sees, use the integrated page inspector tool (see slide 199 for details).
We now look at an example to fortify our intuition.

Cascading of selectors in CSS: Priorization Example

▷ **Example 6.2.12.** 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 agains the color yellow inherited from the parent <div> element (rule 7 above).
Let us now look at CSS inheritance in a little more detail.

Cascading in CSS: Inheritance

▷ **Definition 6.2.13.** Child elements can inherits some properties (called inheritable) from their parents. 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 priorization. (recall case 7. above)

▷ Inheritance makes for consistent text properties and smaller CSS stylesheets.

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 Float to their Place

▷ **Definition 6.2.14.** CSS Flow describes how different elements are distributed in the visible area. (how they flow; hence the name)
The float property allows to influence that.

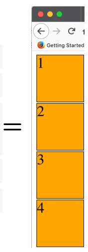
▷ **Example 6.2.15.** Block-level boxes (here div node) float to the left:

```html
<div class="square">1</div>
<div class="square">2</div>
<div class="square">3</div>
<div class="square">4</div>
```

+

```css
.square {font-size:200%;
         height:100px;
         width:100px;
         border:1px solid black;
         margin:2px;
         background-color:orange;}
```

=



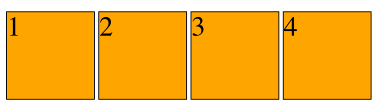
▷ **Example 6.2.16.** float:left floats boxes as far as they will go: (without overlap)

```html
<div class="square">1</div>
<div class="square">2</div>
<div class="square">3</div>
<div class="square">4</div>
```

+

```css
.square {font-size:200%;
         height:100px;
         width:100px;
         border:1px solid black;
         margin:2px;
         background-color:orange;
         float:left}
```

=



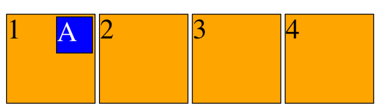
▷ **Example 6.2.17.** float:right in a div will float inside the corresponding box.

```html
<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>
```

+

```css
.smallsq {color:white;
  height: 40px;width: 40px;
  border: 1px solid black;
  margin: 2px;
  background-color: blue;
  float: right}
```

=



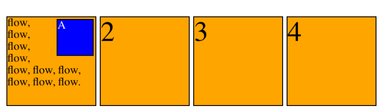
▷ **Example 6.2.18.** float:left will let contents flow around an obstacle

```html
<div class="square"
  style="font-size:small">
  <div class="smallsq">A</div>
  flow, flow, flow, flow, flow,
  flow, flow, flow, flow, flow.
</div>
```

+

```css
.smallsq {color:white;
  height: 40px;width: 40px;
  border: 1px solid black;
  margin: 2px;
  background-color: blue;
  float: right}
```

=



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

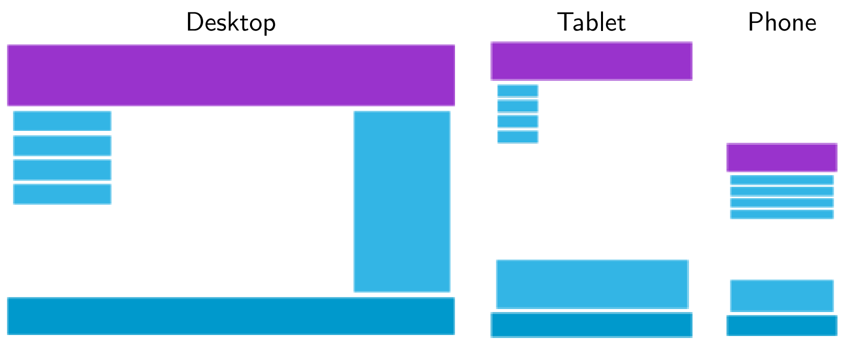
One of the important applications of the content/form separation made possible by CSS is to tailor web page layout to the screen size and resolution of the device it is viewed on. Of course, it would be possible to maintain multiple layouts for a web page one per screensize/resolution class, but a better way is to have one layout that changes according to the device context. This is what we will briefly look at now.

CSS Application: Responsive Design

▷ **Problem:** What is the screen size/resolution of my device?

▷ **Definition 6.2.19.** 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 monitors to mobile phones).

▷ **Example 6.2.20.** A web page with content blocks



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

6.2.4 CSS Tools

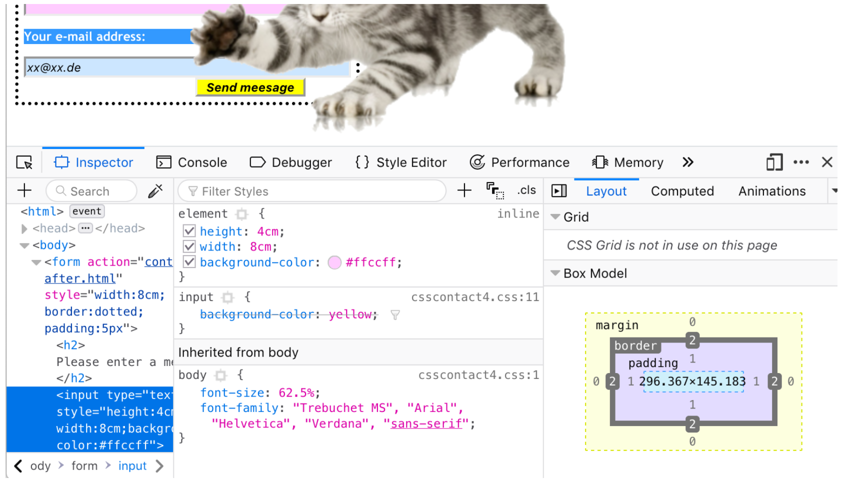
In this subsection we introduce a technology of digital documents which naturally As CSS has grown to be very complex and moreover, the browser DOM of which CSS is part can even be modified after loading the HTML (see ???), we need tools to help us develop effective and maintainable CSS.

But how to find out what the web browser really sees?

▷ CSS has many interesting inheritance rules.

▷ **Definition 6.2.21.** The page inspector tool gives you an overview over the internal state of the web browser and its DOM.

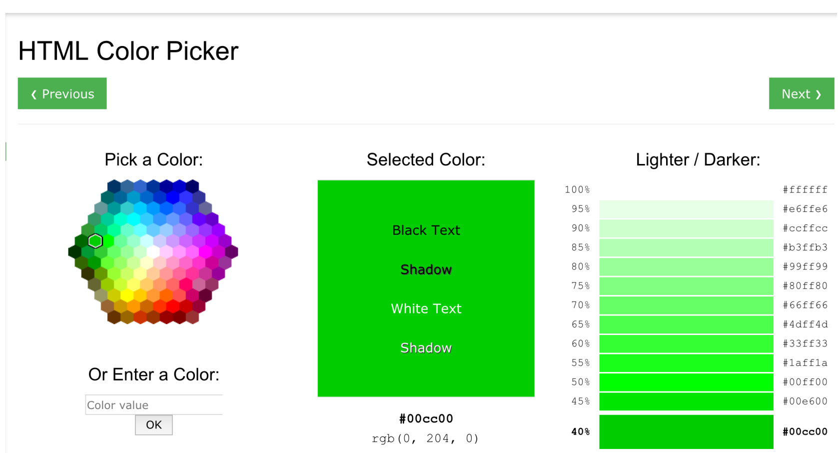
▷ **Example 6.2.22.**



In CSS we can specify colors by various names, but the full range of possible colors can only specified by numeric (usually hexadecimal) numbers. For instance in ???, 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`



6.3 jQuery: Write Less, Do More

While JavaScript is fully sufficient to manipulate the HTML DOM, it is quite verbose and tedious to write. To remedy this, the web developer community has developed libraries that extend the

JavaScript language by new functionalities that more concise programs and are often used Instead of pure JavaScript.

jQuery: Write Less, Do More

- **Definition 6.3.1.** JQuery is a feature-rich JavaScript library that simplifies tasks like HTML document traversal and manipulation, event handling, animation, and Ajax.
- **Using:**
  - Download from `https://jquery.com/download/`, save on your system (remember where)
  - integrate into your HTML (usually in the <head>)

```
<script type="text/javascript" src="client−js/jquery−3.2.1.min.js"/>
```

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

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


The key feature of jQuery is that it borrows the notion of "selectors" to describe HTML node sets from CSS actually, jQuery uses the CSS selectors directly and then uses JavaScript-like methods to act on them. In fact, the name jQuery comes from the fact that selectors "query" for nodes in the DOM.

jQuery Philosophy and Layers

- **jQuery Philosophy:** Select a subtree from the DOM, and operate on it.
- **Syntax Convention:** jQuery instructions start with a $ to distinguish it from JavaScript.
- **Example 6.3.2.** The following jQuery command achieves a lot in four steps:

```
$("#myId").show().css("color", "green").slideDown();
```

1. Find elements in the DOM by CSS selectors, e.g. $("#myId")
2. do something to them, here show() (chaining of methods)
3. change their layout by changing CSS attributes, e.g. css("color","green")
4. change their behavior, e.g. slideDown()

- **Good News:** jQuery selectors $\widehat{=}$ CSS selector.

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 'in the back'; });
```

▷ **Inserting before/after an element:**

```
$('#price').before('Price:');
$('#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

▷ **Observation:** jQuery is not limited to adding material to the DOM.

▷ **Idea:** Use jQuery to change CSS properties in the DOM as well.

▷ **Example 6.3.3 (Visibility).** 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>
```


Fun with Buttons (Three easy Interactions)

▷ **Example 6.3.4 (A Button that Changes Color on Hover).**

```
  <div id="hoverPoint">
    <button id="hover">hover</button>
    <script type="text/javascript">
      $("#hover").hover(function () {$(this).css("background-color", "red");},
                        function () {$(this).css("background-color", "blue");});
    </script>
  </div>
```

- The HTML has a button with text “hover”.
- The jQuery code selects it via its id and catches its hover event via the hover() method.
- This takes two functions as arguments:
  - The first is called when the mouse moves into the button, the second when it leaves.
  - The first changes changes the button color to red, the second reverts this.

Fun with Buttons (Three easy Interactions)

▷ **Example 6.3.5 (A Button that Uncovers Text).**

```
<div id="readPoint">
  <button class="read" style="display:block">Read More</button>
  <button class="read" style="display:none">Read Less</button>
  <div id="rText" style="display:none; width:200px; clear:left">
    A read−more button is not only a call−to−action, but it also organizes
    the screen area management in a non−wasteful way. If and only if users are interested,
    they will use the button.<br/>
  </div>
  <script type="text/javascript">
    $(".read").click(function() {$("#rText").toggle("slow",function(){$(".read").toggle()});});
  </script>
</div>
```

- The HTML has two buttons (one of them visible) and a text.
- The jQuery code selects both buttons via their read class.
- A click event activates the .click() method taking an event handler function:
  - This selects the text via its id attribute rTeX and
  - uses the toggle() method which changes the display between none and block.
  - The first parameter of toggle() is a duration for the animation.
  - The second is a completion function to be run after animation finishes.
  - Here complection function makes the respective other button visible (read more/less).

Fun with Buttons (Three easy Interactions)

▷ **Example 6.3.6 (A Button that Plays a Sound).**

```
<div id="soundPoint">
  <button id="sound" onclick="playSound('laugh.mp3')">Sound</button>
  <script type="text/javascript">
    function playSound(url) {
      console.log("Call playSound with " + url);
      const a = new Audio(url);
      a.play();
      }
  </script>
```

```
</div>
```

- The HTML has a button with text "sound" and an onclick attribute.
- That activates the playSound function on a URL:
- The playSound function is defined in the script element: it
  - logs the action and URL in the browser console,
  - makes a new audio object a, which
  - plays it via the play() method.

For reference, here is the full code of the examples in one file:

```
<html>
<head>
  <title>Buttons</title>
  <script src="https://code.jquery.com/jquery−3.4.1.min.js" type="text/javascript"></script>
  <style type="text/css">
    button {color: white; font−size: large; background−color: blue;
                 width: 110px; height: 40px; border−radius: 20px;}
    div[id$="Point"] {display: inline−block;}
  </style>
</head>

<body>
  <h1 id="top">Look how easy interaction is ... </h1>

  <div id="hoverPoint">
    <button id="hover">hover</button>
    <script type="text/javascript">
      $("#hover").hover(function () {$(this).css("background−color", "red");},
                          function () {$(this).css("background−color", "blue");});
    </script>
  </div>

  <div id="readPoint">
    <button class="read" style="display:block">Read More</button>
    <button class="read" style="display:none">Read Less</button>
    <div id="rText" style="display:none; width:200px; clear:left">
      A read−more button is not only a call−to−action, but it also organizes
      the screen area management in a non−wasteful way. If and only if users are interested,
      they will use the button.<br/>
    </div>
    <script type="text/javascript">
      $(".read").click(function() {$("#rText").toggle("slow",function(){$(".read").toggle()});});
    </script>
  </div>

  <div id="soundPoint">
    <button id="sound" onclick="playSound('laugh.mp3')">Sound</button>
    <script type="text/javascript">
      function playSound(url) {
        console.log("Call playSound with " + url);
        const a = new Audio(url);
        a.play();
        }
    </script>
  </div>
</body>
</html>
```

It has a bit more general CSS and includes jQuery in the beginning.

Chapter 7

Practical Aspects of Web Applications

7.1 Web Applications: Recap

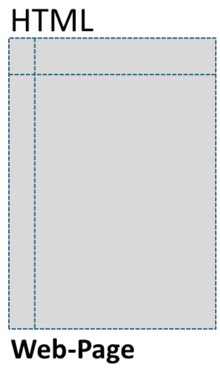
What Tools have we seen so far?

- ▷ HTML (Hypertext Markup Language)
  - ▷ Text-based markup language for the web.
  - ▷ Tree structure (realized as the DOM in the browser)
    - ▷ easy search&find ⇜ selection
    - ▷ DOM changes easy by clear dependencies.
- ▷ CSS (Cascading Stylesheets)
  - ▷ Language for specifying layout of HTML/DOM
  - ▷ CSS selection ties layout specifications into HTML/DOM
- ▷ Bottle WSGI (server-side web page generation via Python)
  - ▷ full programming language for comprehensive functionality
  - ▷ routes for complex but coherent web sites
  - ▷ template engine for HTML-centered web page design
- ▷ JavaScript (client-side scripting)
  - ▷ full programming language (Turing complete)
  - ▷ programmatic changes to the DOM ⇝ dynamic HTML
    - ▷ navigating the DOM via JS-selection (relatively clumsy, but sufficient)
    - ▷ jQuery navigates the DOM via CSS selectors (reuses successful concepts)

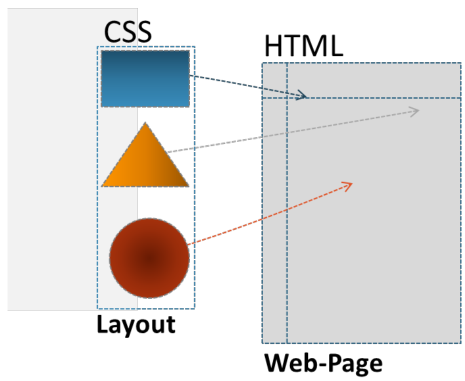
Recap: Web Application Front-end

▷ **Recap: Web Application Front-end:**

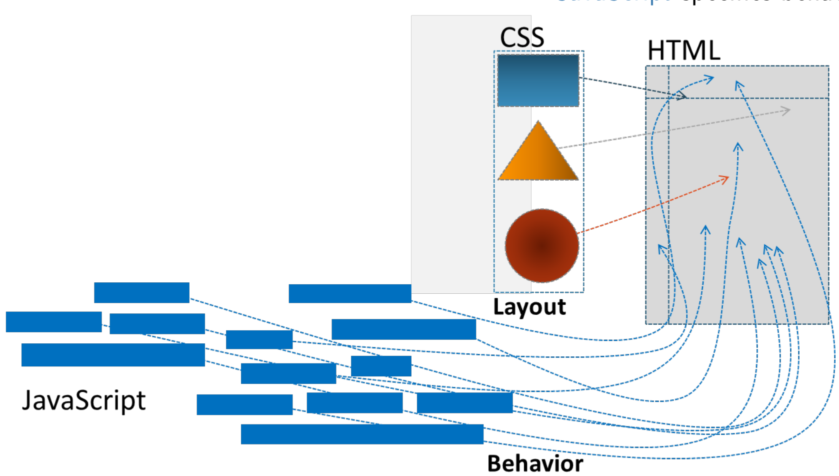
Web pages are just HTML files.



Layout is specified by CSS instructions and selectors

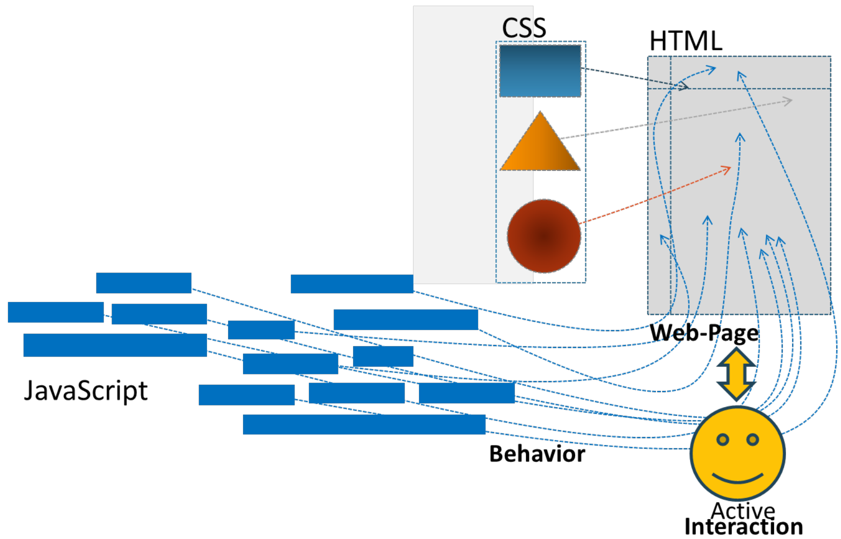


JavaScript specifies behavior



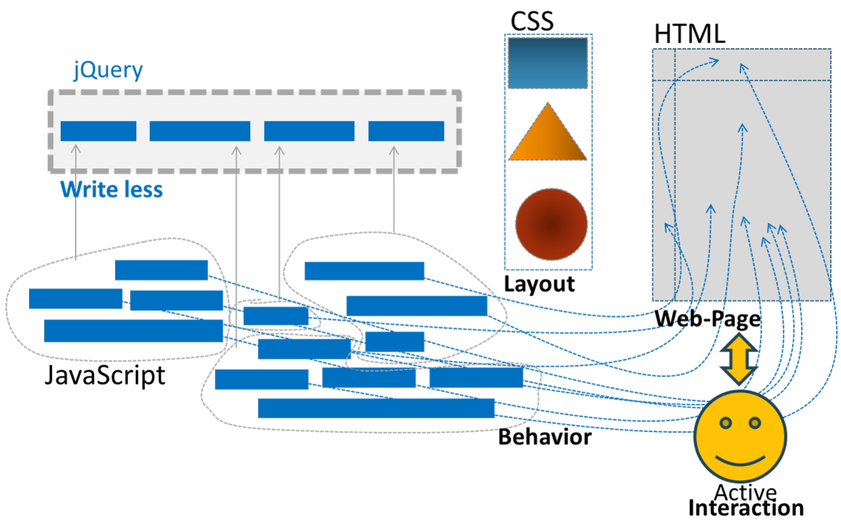
for

interacting with the user



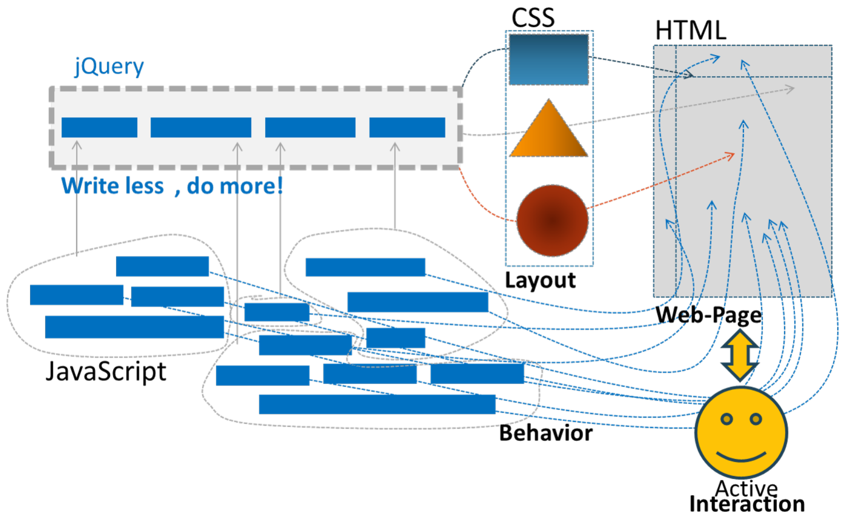
jQuery

≙ more succinct JavaScript



jQuery

attaches behaviors to DOM elements via CSS selectors



209 2025-06-05

7.2 Maintaining State in Web Sites

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 we only want to log in 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 7.2.1 (HTTP GET for Single Login).** Since we are generating pages we can generated 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 7.2.2.** 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 7.2.3.** Third-party cookies are used by advertising companies to track users across multiple sites. (but you can turn off, and even delete cookies)

Note that both solutions to the state problem are not ideal, for usernames and passwords the URL-based solution is particularly problematic, since HTTP transmits URLs in GET requests without encryption, and in our example passwords would be visible to anybody with a packet sniffer. Here cookies are little better, since they can be requested by any web site you visit.

7.3 Access Control and Management

Now that we have a basic web application running, we can start adding features. The most important one is access control to restrict who can access more critical functionalities of the web application, such as deleting or updating database record.

There are many technologies for access control, many use advanced features like browser cookies. Here we want to introduce the simplest one: HTTP basic authentication is built into the fabric of the world wide web: it is part of the HTTP protocol that drives it.

As HTTP basic authentication is unsafe (it sends usernames and passwords over the network only lightly encrypted), we also add a discussion on how to upgrade the web application to HTTPS.

The full source is available at `https://gl.mathhub.info/courses/FAU/IWGS/blob/master/source/databases/code/books-app-https.py`. The respective template files are siblings.

Access Control and Management

▷ **Problem:** Anyone can write, edit, and delete records from the books database.

▷ **Solution:** Implement a password-based log in procedure and restrict write/edit/delete access to logged-in agents.

▷ Let's fix some terminology before we continue

▷ **Definition 7.3.1.** Access control (AC) is the selective restriction of access to a resource or place, access management describes the corresponding process.

▷ Access management usually comprises both authentication and authorization.

▷ **Definition 7.3.2.** Authorization refers to a set of rules that determine who is allowed to do what with a collection of resources.

▷ **For our books application** we need four things

1. a browser interaction to query the user for username and password
2. a way to transport them to the web application program
3. a method for checking the username/password (authentication)
4. a way the specify who can do what. (authorization)

Realization: 1./2. via HTTP, 4. via bottle basic auth, implement 3. directly.

HTTP basic authentication is a simple mechanism in the HTTP protocol that standardizes the transmission of username/password information the “handshake” that leads to its acquisition.

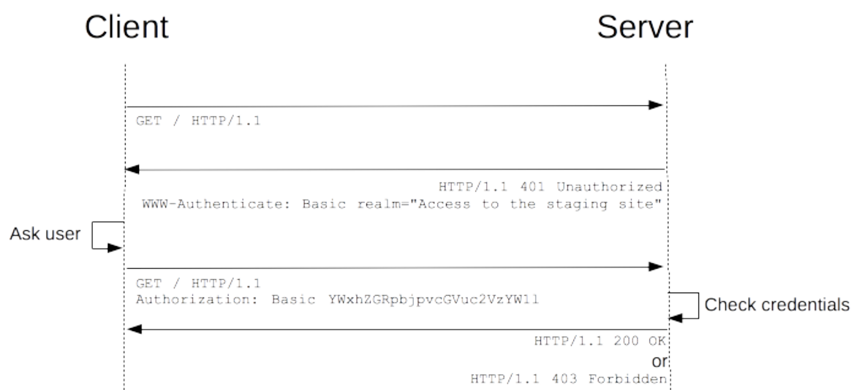
HTTP Basic Authentication

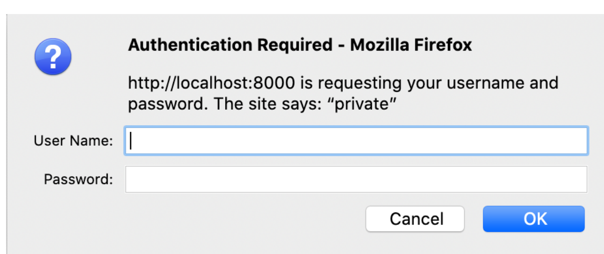
▷ **Recall** that HTTP is a plain text protocol that passes around headers like this

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

▷ **Idea:** For authentication extend the HTTP headers with support for username/password pairs.

▷ **Definition 7.3.3.** HTTP basic authentication introduces a HTTP header Authorization for base64 encoded pairs ⟪username⟫:⟪password⟫ and a couple of challenge/response messages.





▷ **Problem:** Base64 is very easy to decode, so usernames and passwords are communicated in the clear. (very unsafe)

▷ Passwords are "binary data" (think special characters), encoding just keeps them unchanged over the network. (no encryption)

The message sequence diagram in Definition 7.3.3 shows the basic handshake process that establishes authentication and the delivery of restricted resources to an authenticated user.

The diagram shows the details of the communication between client and server (symbolized by the two vertical lines). The top arrow is a normal HTTP GET request (without a Authorization field).

But – as the resource that is requested is access-restricted – the server does not just answer with a HTTP "200 OK" and the resource, instead the server response with a HTTP "401 Unauthorized" code, which contains a description of the reason for the restriction.

When the browser receives the 401 response, it asks the user for a username and password e.g. with a popup form like the one shown in Definition 7.3.3, possibly displaying the reason string – here "private". This information is then send to the server in a second GET request, this time with the username/password information in the Authorization request.

The server checks the user/password data and – depending on the result of the check – sends a HTTP response "200 OK" together with the resource or a "403 Forbidden" (without the resource).

One thing that we have not discussed here is that most browsers store the username/password information and supply it to the server – often directly in any outgoing requests – which makes it hard to test authentication and unauthenticated behavior in web application development. A useful trick here is – if you are logged into `http://example.org` – to address a GET request to `http://abc@example.org`. Background: HTTP basic authentication allows you to set user/password information directly by prepending ⟪user⟫:⟪pass⟫ to the authority of the URI used in a HTTP request.

Of course, HTTP basic authentication is supported by the bottle WSGI framework.

Basic Auth in Bottle

- ▷ **Idea:** Support the server side of HTTP basic authentication in bottle web-apps.
- ▷ **Implementation:** New decorator @auth_basic(⟪function⟫) to mark a route as password-protected.
- ▷ **Usage:** Decorate every route we want to restrict access of with @auth_basic(⟪function⟫), where ⟪function⟫ is a function that takes two string arguments (user name and password) and returns a Boolean for the authorization decision.

FAU : 213 2025-06-05

What happens behind the scene here is clear from the authentication handshake explained in ???

Basic Auth in Bottle: Minimal Viable Example

▷ **Example 7.3.4.** A web application with restricted route.

```
from bottle import run, get, auth_basic

def check(user, password):
    return user == "miko" and password == "test"

@get("/")
@auth_basic(check)
def protected():
    return "Authorized access granted!"

run(host="localhost", port=8000)
```

- ▷ **Idea:** Mix restricted and open routes in a partially restricted application.
- ▷ **Extension:** Use different check functions for different levels of restriction (user roles)

This was easy enough. But one problem remains: in HTTP basic authentication, usernames and passwords are not confidential when they are transported over the network. The simplest way to ensure confidentiality is to layer encryption on top of HTTP, which is just what the HTTPS protocol does.

7.4 HTTPS: Secure/Encrypted HTTP

HTTPS: HTTP over TLS

- **Definition 7.4.1.** Hypertext Transfer Protocol Secure (HTTPS) is an extension of the Hypertext Transfer Protocol (HTTP) for confidential communication over a computer network. HTTPS achieves this by running HTTP over a TLS connection.
- **Consequences for Web Applications:** We can use HTTP as usual, except
  - we gain communication privacy and server authentication,
  - server and browser need to speak HTTPS, (most do)
  - the server needs a public key certificate and a private key.
- In bottle, we can just swap out the HTTP server to one that can do HTTPS:

```
run(host='localhost',port='8888',
    server='gunicorn',keyfile='key.pem',certfile='cert.pem')
```

  install it first with pip install gunicorn.
- **Problem:** Where to get the certificate file cert.pem and private key key.pem?

For publically deploying a HTTPS based web application we need real TLS certificates. Fortunately, there is a relatively simple way of obtaining them.

Getting a Real TLS Certificate via Let's-Encrypt

- **Intuition:** HTTPS is the new "regular HTTP" on the web!
- **Definition 7.4.2.** In a public key infrastructure, the TLS certificate is issued by a certificate authority, an organization chartered to verify identity and issue TLS certificates.
- Commercial certificate authorities sell trust. (for a lot of money)
  They certify e.g. that the `https://bmw.com` is under control of BMW AG.
- **Idea:** Finding out that you have control over a particular web site on the web can be automated, if you run a program on the server host.
- **Definition 7.4.3.** Let's Encrypt is a not for profit certificate authority that does this and issues free TLS certificates. (to encourage HTTPS adoption)
- **Concretely:** on a linux server you need two steps

  1. install certbot (usually via your package manager)

2. then sudo /usr/local/bin/certbot certonly ––standalone will generate certs.

Details at `https://letsencrypt.org`.

▷ **Success:** $\geq$ 1.000.000.000 TLS certificates, 200.000.000 sites since 2016

We have only covered the basic ideas behind certificate authorities and Let's Encrypt here, but this should enable you to figure out the rest from the Let's Encrypt web site.

Chapter 8

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 wordsstring, character encodings, unicode, and regular expressions
- ▷ Electronic documents and document processing:
  - ▷ text files
  - ▷ markup systems, HTML, and CSS
  - ▷ XML: Documents are trees.
- ▷ Web technologies for interactive documents and web applications
  - ▷ internet infrastructure: web browsers and server
  - ▷ server-side computation: bottle routing and
  - ▷ client-side interaction: dynamic HTML, JavaScript, HTML forms
- ▷ Web application project (fill in the blanks to obtain a working web app)

Outline of IWGS-II:

- ▷ Databases
  - ▷ CRUD operations, querying, and python embedding
  - ▷ XML and JSON for file based data storage
- ▷ BooksApp: a Books Application with persistent storage
- ▷ Image processing
  - ▷ Basics

  - ▷ Image transformations, Image Understanding
- ▷ Ontologies, semantic web, and WissKI
  - ▷ Ontologies (inference ⇝ get out more than you put in)
  - ▷ semantic web Technologies (standardize ontology formats and inference)
  - ▷ Using semantic web Tech for cultural heritage research data ⇝ the WissKI System
- ▷ Legal Foundations of Information Systems
  - ▷ Copyright & Licensing
  - ▷ Data Protection (GDPR)

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.

[CSSa] *All CSS Specifications*. URL: https://www.w3.org/Style/CSS/specs.en.html (visited on 01/12/2020).

[CSSb] *CSS Specificity*. URL: https://en.wikipedia.org/wiki/Cascading_Style_Sheets#Specificity (visited on 12/03/2018).

[CSSc] *CSS Tutorial*. URL: https://www.w3schools.com/css/default.asp (visited on 12/02/2018).

[DH98] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460. Internet Engineering Task Force (IETF), 1998. URL: http://www.ietf.org/rfc/rfc2460.txt.

[Ecm] *ECMAScript Language Specification*. ECMA Standard. 5th Edition. Dec. 2009.

[ET] *xml.etree.ElementTree – The ElementTree XML API*. URL: https://docs.python.org/3/library/xml.etree.elementtree.html (visited on 04/15/2021).

[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 Recommentation. 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. 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).

[JKI] Jonas Betzendahl. *juptyter.kwarc.info Documentation*. URL: https://kwarc.info/teaching/IWGS/jupyter-documentation.pdf (visited on 08/29/2020).

[Kar] Folgert 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 LaTeX 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`.

[LP] *Learn Python – Free Interactive Python Tutorial*. URL: `https://www.learnpython.org/` (visited on 10/24/2018).

[LXMLa] *lxml – XML and HTML with Python*. URL: `https://lxml.de` (visited on 12/09/2019).

[LXMLb] *lxml API*. URL: `https://lxml.de/api/` (visited on 12/09/2019).

[LXMLc] *The lxml.etree Tutorial*. URL: `https://lxml.de/tutorial.html` (visited on 12/09/2019).

[Nor+18a] Emily Nordmann et al. *Lecture capture: Practical recommendations for students and lecturers*. 2018. URL: `https://osf.io/huydx/download`.

[Nor+18b] Emily Nordmann et al. *Vorlesungsaufzeichnungen nutzen: Eine Anleitung für Studierende*. 2018. URL: `https://osf.io/e6r7a/download`.

[P3D] *Python 3 Documentation*. URL: `https://docs.python.org/3/` (visited on 09/02/2014).

[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`.

[Rfc] *DOD Standard Internet Protocol*. RFC. 1980. URL: `http://tools.ietf.org/rfc/rfc760.txt`.

[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`.

[She24] Esther Shein. 2024. URL: `https://cacm.acm.org/news/the-impact-of-ai-on-computer-science-education/`.

[sTeX] *sTeX: A semantic Extension of TeX/LaTeX*. URL: `https://github.com/sLaTeX/sTeX` (visited on 05/11/2020).

[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`.

Appendix A

Excursions

As this course is predominantly an overview over (some) CS tools useful in the humanities and social sciences and not about the theoretical underpinnings, we give the discussion about these as a "suggested readings" chapter here.

A.1 Internet Basics

We will show aspects of how the internet can cope with this enormous growth of numbers of computers, connections and services. The growth of the internet rests on three design decisions taken very early on. The internet

1. is a packet-switched network rather than a network, where computers communicate via dedicated physical communication lines.
2. is a network, where control and administration are decentralized as much as possible.
3. is an infrastructure that only concentrates on transporting packets/datagrams between computers. It does not provide special treatment to any packets, or try to control the content of the packets.

The first design decision is a purely technical one that allows the existing communication lines to be shared by multiple users, and thus save on hardware resources. The second decision allows the administrative aspects of the internet to scale up. Both of these are crucial for the scalability of the internet. The third decision (often called "net neutrality") is hotly debated. The defenders cite that net neutrality keeps the Internet an open market that fosters innovation, where as the attackers say that some uses of the network (illegal file sharing) disproportionately consume resources.

Package-Switched Networks

- ▷ **Definition A.1.1.** A packet switched network divides messages into small network packets that are transported separately and re assembled at the target.
- ▷ **Advantages:**
  - ▷ many users can share the same physical communication lines.
  - ▷ packets can be routed via different paths. (bandwidth utilization)
  - ▷ bad packets can be re-sent, while good ones are sent on. (network reliability)
  - ▷ packets can contain information about their sender, destination.
  - ▷ no central management instance necessary (scalability, resilience)

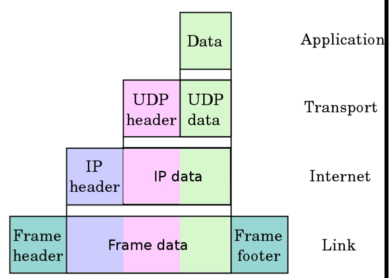
These ideas are implemented in the Internet Protocol Suite, which we will present in the rest of the section. A main idea of this set of protocols is its layered design that allows to separate concerns and implement functionality separately.

The Intenet Protocol Suite

▷ **Definition A.1.2.** The Internet Protocol Suite (commonly known as TCP/IP) is the set of communications protocols used for the internet and other similar networks. It structured into 4 layers.

Layer	e.g.
Application Layer	HTTP, SSH
Transport Layer	UDP, TCP
Internet Layer	IPv4, IPsec
Link Layer	Ethernet, DSL

▷ **Layers in TCP/IP:** TCP/IP uses encapsulation to provide abstraction of protocols and services.
An application (the highest level of the model) uses a set of protocols to send its data down the layers, being further encapsulated at each level.



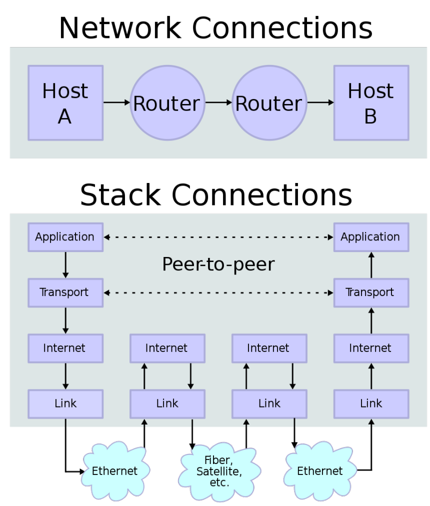
The Internet as a Network of Networks

▷ **Example A.1.3 (TCP/IP Scenario).** Consider a situation with two internet host computers communicate across local network boundaries.

▷ Network boundaries are constituted by internetworking gateways (routers).

▷ **Definition A.1.4.** A router is a purposely customized computer used to forward data among computer networks beyond directly connected devices.

▷ A router implements the link and internet layers only and has two network connections.



We will now take a closer look at each of the layers shown above, starting with the lowest one.
Instead of going into network topologies, protocols, and their implementation into physical signals that make up the link layer, we only discuss the devices that deal with them. Network Interface

controllers are specialized hardware that encapsulate all aspects of link-level communication, and we take them as black boxes for the purposes of this course.

Network Interfaces

- ▷ The nodes in the internet are computers, the edges communication channels
- ▷ **Definition A.1.5.** A network interface controller (NIC) is a hardware device that handles an interface to a computer network and thus allows a network-capable device to access that network.
- ▷ **Definition A.1.6.** Each NIC contains a unique number, the media access control address (MAC address), identifies the device uniquely on the network.
- ▷ MAC addresses are usually 48-bit numbers issued by the manufacturer, they are usually displayed to humans as six groups of two hexadecimal digits, separated by hyphens (-) or colons (:), in transmission order, e.g. 01-23-45-67-89-AB, 01:23:45:67:89:AB.
- ▷ **Definition A.1.7.** A network interface is a software component in the operating system that implements the higher levels of the network protocol (the NIC handles the lower ones).

Layer	e.g.
Application Layer	HTTP, SSH
Transport Layer	TCP
Internet Layer	IPv4, IPsec
Link Layer	Ethernet, DSL

- ▷ A computer can have more than one network interface. (e.g. a router)

The next layer ist he Internet Layer, it performs two parts: addressing and packing packets.

Internet Protocol and IP Addresses

- ▷ **Definition A.1.8.** The Internet Protocol (IP) is a protocol used for communicating data across a packet-switched internetwork. The Internet Protocol defines addressing methods and structures for datagram encapsulation. The Internet Protocol also routes data packets between networks
- ▷ **Definition A.1.9.** An IP address is a numerical label that is assigned to devices participating in a computer network, that uses the Internet Protocol for communication between its nodes.
- ▷ An IP address serves two principal functions: host or network interface identification and location addressing.
- ▷ **Definition A.1.10.** The global IP address space allocations are managed by the Internet Assigned Numbers Authority (IANA), delegating allocate IP address blocks to five Regional Internet Registries (RIRs) and further to Internet service providers (ISPs).

Internet Protocol and IP Addresses

▷ **Definition A.1.11.** The internet mainly uses Internet Protocol Version 4 (IPv4) [Rfc], which uses 32 bit numbers (IPv4 addresses) for identification of network interfaces of computers.

▷ IPv4 was standardized in 1980, it provides 4,294,967,296 (2^{32}) possible unique addresses. With the enormous growth of the internet, we are fast running out of IPv4 addresses.

▷ **Definition A.1.12.** Internet Protocol Version 6 [DH98] (IPv6), which uses 128 bit numbers (IPv6 addresses) for identification.

▷ Although IP addresses are stored as binary numbers, they are usually displayed in human-readable notations, such as $208.77.188.166$ (for IPv4), and $2001{:}db8{:}0{:}1234{:}0{:}567{:}1{:}1$ (for IPv6).

The internet infrastructure is currently undergoing a dramatic retooling, because we are moving from IPv4 to IPv6 to counter the depletion of IP addresses. Note that this means that all routers and switches in the internet have to be upgraded. At first glance, it would seem that this problem could have been avoided if we had only anticipated the need for more the 4 million computers. But remember that TCP/IP was developed at a time, where the internet did not exist yet, and it's precursor had about 100 computers. Also note that the IP addresses are part of every packet, and thus reserving more space for them would have wasted bandwidth in a time when it was scarce.

We will now go into the detailed structure of the IP packets as an example of how a low-level protocol is structured. Basically, an IP packet has two parts: the "header", whose sequence of bytes is strictly standardized, and the "payload", a segment of bytes about which we only know the length, which is specified in the header.

The Structure of IP Packets

▷ **Definition A.1.13.** IP packets are composed of a $160\mathrm{b}$ header and a payload. The IPv4 packet header consists of:

b	name	comment
4	version	IPv4 or IPv6 packet
4	Header Length	in multiples 4 bytes (e.g., 5 means 20 bytes)
8	QoS	Quality of Service, i.e. priority
16	length	of the packet in bytes
16	fragid	to help reconstruct the packet from fragments,
3	fragmented	DF $\widehat{=}$ "Don't fragment"/MF $\widehat{=}$ "More Fragments"
13	fragment offset	to identify fragment position within packet
8	TTL	Time to live (router hops until discarded)
8	protocol	TCP, UDP, ICMP, etc.
16	Header Checksum	used in error detection,
32	Source IP	
32	target IP	
...	optional flags	according to header length

▷ Note that delivery of IP packets is not guaranteed by the IP protocol.

As the internet protocol only supports addressing, routing, and packaging of packets, we need another layer to get services like the transporting of files between specific computers. Note that the IP protocol does not guarantee that packets arrive in the right order or indeed arrive at all, so the transport layer protocols have to take the necessary measures, like packet re-sending or handshakes, ...

The Transport Layer

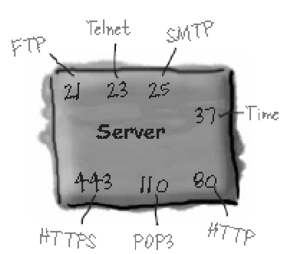
- ▷ **Definition A.1.14.** The transport layer is responsible for delivering data to the appropriate application process on the host computers by forming data packets, and adding source and destination port numbers in the header.
- ▷ **Definition A.1.15.** The internet protocol mainly uses suite the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) protocols at the transport layer.
- ▷ TCP is used for communication, UDP for multicasting and broadcasting.
- ▷ TCP supports virtual circuits, i.e. provide connection oriented communication over an underlying packet oriented datagram network. (hide/reorder packets)
- ▷ TCP provides end-to-end reliable communication (error detection & automatic repeat)

We will see that there are quite a lot of services at the network application level. And indeed, many web-connected computers run a significant subset of them at any given time, which could lead to problems of determining which packets should be handled by which service. The answer to this problem is a system of "ports" (think pigeon holes) that support finer-grained addressing to the various services.

Ports

- ▷ **Definition A.1.16.** To separate the services and protocols of the network application layer, network interfaces assign them specific port, referenced by a number.
- ▷ **Example A.1.17.** We have the following ports in common use on the internet

Port	use	comment
22	SSH	remote shell
53	DNS	Domain Name System
80	HTTP	World Wide Web
443	HTTPS	HTTP over SSL



On top of the transport-layer services, we can define even more specific services. From the perspective of the internet protocol suite this layer is unregulated, and application-specific. From a user perspective, many useful services are just "applications" and live at the application layer.

The Application Layer

- **Definition A.1.18.** The application layer of the internet protocol suite contains all protocols and methods that fall into the realm of process-to-process communications via an Internet Protocol (IP) network using the Transport Layer protocols to establish underlying host-to-host connections.
- **Example A.1.19 (Some Application Layer Protocols and Services).**

BitTorrent	Peer-to-peer	Atom	Syndication
DHCP	Dynamic Host Configuration	DNS	Domain Name System
FTP	File Transfer Protocol	HTTP	HyperText Transfer
IMAP	Internet Message Access	IRCP	Internet Relay Chat
NFS	Network File System	NNTP	Network News Transfer
NTP	Network Time Protocol	POP	Post Office Protocol
RPC	Remote Procedure Call	SMB	Server Message Block
SMTP	Simple Mail Transfer	SSH	Secure Shell
TELNET	Terminal Emulation	WebDAV	Write-enabled Web

The domain name system is a sort of telephone book of the internet that allows us to use symbolic names for hosts like `kwarc.info` instead of the IP number 212.201.49.189.

Domain Names

- **Definition A.1.20.** The DNS (Domain Name System) is a distributed set of servers that provides the mapping between (static) IP addresses and domain names.
- **Example A.1.21.** e.g. `www.kwarc.info` stands for the IP address 212.201.49.189.
- **Definition A.1.22.** Domain names are hierarchically organized, with the most significant part (the top level domain TLD) last.
- networked computers can have more than one DNS name. (virtual servers)
- Domain names must be registered to ensure uniqueness (registration fees vary, cybersquatting)
- **Definition A.1.23.** ICANN is a non profit organization was established to regulate human friendly domain names. It approves top-level domains, and corresponding domain name registrars and delegates the actual registration to them.

Let us have a look at a selection of the top-level domains in use today.

Domain Name Top-Level Domains

- .com (“commercial”) is a generic top-level domain. It was one of the original top-level domains, and has grown to be the largest in use.
- .org (“organization”) is a generic top-level domain, and is mostly associated with non-profit organizations. It is also used in the charitable field, and used by the open-

source movement. Government sites and Political parties in the US have domain names ending in .org

- ▷ .net (“network”) is a generic top-level domain and is one of the original top-level domains. Initially intended to be used only for network providers (such as Internet service providers). It is still popular with network operators, it is often treated as a second .com. It is currently the third most popular top-level domain.
- ▷ .edu (“education”) is the generic top-level domain for educational institutions, primarily those in the United States. One of the first top-level domains, .edu was originally intended for educational institutions anywhere in the world. Only post-secondary institutions that are accredited by an agency on the U.S. Department of Education’s list of nationally recognized accrediting agencies are eligible to apply for a .edu domain.

Domain Name Top-Level Domains

- ▷ .info (“information”) is a generic top-level domain intended for informative website’s, although its use is not restricted. It is an unrestricted domain, meaning that anyone can obtain a second-level domain under .info. The .info was one of many extension(s) that was meant to take the pressure off the overcrowded .com domain.
- ▷ .gov (“government”) a generic top-level domain used by government entities in the United States. Other countries typically use a second-level domain for this purpose, e.g., .gov.uk for the United Kingdom. Since the United States controls the .gov Top Level Domain, it would be impossible for another country to create a domain ending in .gov.
- ▷ .biz (“business”) the name is a phonetic spelling of the first syllable of “business”. A generic top-level domain to be used by businesses. It was created due to the demand for good domain names available in the .com top-level domain, and to provide an alternative to businesses whose preferred .com domain name which had already been registered by another.
- ▷ .xxx (“porn”) the name is a play on the verdict “X-rated” for movies. A generic top-level domain to be used for sexually explicit material. It was created in 2011 in the hope to move sexually explicit material from the “normal web”. But there is no mandate for porn to be restricted to the .xxx domain, this would be difficult due to problems of definition, different jurisdictions, and free speech issues.

Note: Anybody can register a domain name from a registrar against a small yearly fee. Domain names are given out on a first-come-first-serve basis by the domain name registrars, which usually also offer services like domain name parking, DNS management, URL forwarding, etc.

The telnet Protocol

- ▷ **Problem:** We need a way to remotely operate networked computers via a shell.
- ▷ **Idea:** Send shell instructions and responses as text messages between a terminal

client (a program on the local host) and a terminal server (a program on the remote host).

▷ **Definition A.1.24.** The telnet protocol uses TCP directly to send text based messages two networked computers. It customarily uses port 25.

▷ **Remark:**

telnet is one of the oldest protocols in the TCP/IP protocol suite. It is no longer used much by itself (it is superseded by rsh and ssh), but still serves as a basis for other protocols, e.g. HTTP.

The next application-level service is the SMTP protocol used for sending e-mail. It is based on the telnet protocol for remote terminal emulation which we do not discuss here.

A Protocol Example: SMTP over telnet

▷ **Definition A.1.25.** The Simple Mail Transfer Protocol (SMTP) is a communication protocol for electronic mail transmission based on telnet.

▷ **Example A.1.26.** The SMTP protocol starts out by establishing identity

- ▷ We call up the telnet service on the Jacobs mail server

```
telnet exchange.jacobs-university.de 25
```

- ▷ it identifies itself (have some patience, it is very busy)

```
Trying 10.70.0.128...
Connected to exchange.jacobs-university.de.
Escape character is '^]'.
220 SHUBCAS01.jacobs.jacobs-university.de
Microsoft ESMTP MAIL Service ready at Tue, 3 May 2011 13:51:23 +0200
```

- ▷ We introduce ourselves politely (but we lie about our identity)

```
helo mailhost.domain.tld
```

- ▷ It is really very polite.

```
250 SHUBCAS04.jacobs.jacobs-university.de Hello [10.222.1.5]
```


SMTP over telnet: The e mail itself

▷ **Example A.1.27 (Continued).** After identity is established, the e-mail is specified.

- ▷ We start addressing an e-mail (again, we lie about our identity)

```
mail from: user@domain.tld
```

- ▷ this is acknowledged

```
250 2.1.0 Sender OK
```

- ▷ We set the recipient (the real one, so that we really get the e-mail)

```
rcpt to: m.kohlhase@jacobs-university.de
```

- ▷ this is acknowledged

```
250 2.1.0 Recipient OK
```

- ▷ we tell the mail server that the mail data comes next

```
data
```

- ▷ this is acknowledged

```
354 Start mail input; end with <CRLF>.<CRLF>
```

- Now we can just type the a-mail, optionally with Subject, date,...

```
Subject: Test via SMTP

and now the mail body itself
.
```

- And a dot on a line by itself sends the e mail off

```
250 2.6.0 <ed73c3f3-f876-4d03-98f2-e5ad5bbb6255@SHUBCAS04.jacobs.jacobs-university.de>
[InternalId=965770] Queued mail for delivery
```


SMTP over telnet: Disconnecting

- **Example A.1.28 (Continued).**
  - That was almost all, but we close the connection (this is a telnet command)

```
quit
```

  - our terminal server (the telnet program) tells us

```
221 2.0.0 Service closing transmission channel
Connection closed by foreign host.
```


Essentially, the SMTP protocol mimics a conversation of polite computers that exchange messages by reading them out loud to each other (including the addressing information). We could go on for quite a while with understanding one Internet protocol after each other, but this is beyond the scope of this course (indeed there are specific courses that do just that). Here we only answer the question where these protocols come from, and where we can find out more about them.

Internet Standardization

- **Question:** Where do all the protocols come from?(someone has to manage that)
- **Definition A.1.29.** The Internet Engineering Task Force (IETF) is an open standards organization that develops and standardizes internet standards, in particular the TCP/IP and Internet protocol suite.
- All participants in the IETF are volunteers (usually paid by their employers)
- **Rough Consensus and Running Code:** Standards are determined by the "rough consensus method" (consensus preferred, but not all members need agree) IETF is interested in practical, working systems that can be quickly implemented.
- **Idea:** running code leads to rough consensus or vice versa.
- **Definition A.1.30.** The standards documents of the IETF are called Request for Comments (RFC). (more than 6300 so far; see `http://www.rfceditor.org/`)