Informatische Werkzeuge in den Geistes- und Sozialwissenschaften 1/2

Prof. Dr. Michael Kohlhase

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

2024-02-08

Informatische Werkzeuge in den Geistes- und Sozialwissenschaften 1/2

Prof. Dr. Michael Kohlhase

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

2024-02-08

Chapter 1
Preliminaries

1.1 Administrativa

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)

Assessment, Grades

- **Grading Background/Theory:** Only modules are graded! (by the law)
  - Module “DH-Einführung” (DHE) $\widehat{=}$ courses IWGS1/2, DH-Einführung.
  - DHE module grade $\leadsto$ pass/fail determined by “portfolio” $\widehat{=}$ collection of contributions/assessments.
- **Assessment Practice:** The IWGS assessments in the “portfolio” consist of
  - weekly homework assignments, (practice IWGS concepts and tools)
  - 60 minutes exam directly after lectures end: $\sim$ Feb. 10. 2024.
- **Retake Exam:** 60 min exam at the end of the exam break. ($\sim$ May 4. 2024)

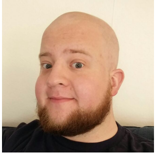
IWGS Homework Assignments

- **Homeworks:** will be small individual problem/programming/system assignments
  - but take time to solve (at least read them directly ⇝ questions)
  - group submission if and only if explicitly permitted.
- ⚠ Without trying the homework assignments you are unlikely to pass the exam.
- **Admin:** To keep things running smoothly
  - Homeworks will be posted on StudOn.
  - Sign up for IWGS under `https://www.studon.fau.de/crs5323051.html`.
  - Homeworks are handed in electronically there. (plain text, program files, PDF)
  - Go to the tutorials, discuss with your TA! (they are there for you!)
- **Homework Discipline:**
  - Start early! (many assignments need more than one evening's work)
  - Don't start by sitting at a blank screen (talking & study group help)
  - Humans will be trying to understand the text/code/math when grading it.

IWGS Tutorials

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

  Tutor: (Doctoral Student in CS)
  - - Jonas Betzendahl: `jonas.betzendahl@fau.de`
  They know what they are doing and really want to help you learn! (dedicated to DH)



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

Textbook, Handouts and Information, Forums, Videos

- **No Textbook:** but lots of online python tutorials on the web.
- Course notes will be posted at `http://kwarc.info/teaching/IWGS` (see references)
  - I mostly prepare/adapt/correct them as we go along.
  - please e-mail me any errors/shortcomings you notice. (improve for the group)
- The lecture videos of WS 2020/21 are at `https://www.fau.tv/course/id/1923` (not much changed)
- Matrix chat at `#iwgs:fau.de` (via IDM) (instructions)
- **StudOn Forum:** `https://www.studon.fau.de/crs5323051.html` for
  - announcements, homeworks (my view on the forum)
  - questions, discussion among your fellow students (your forum too, use it!)
- If you become an active discussion group, the forum turns into a valuable resource!

Experiment: Learning Support with KWARC Technologies

- **My research area:** Deep representation formats for (mathematical) knowledge
- **One Application:** Learning support systems (represent knowledge to transport it)
- **Experiment:** Start with this course (Drink my own medicine)
  1. Re-represent the slide materials in OMDoc (Open Mathematical Documents)
  2. Feed it into the ALeA system (`http://courses.voll-ki.fau.de`)
  3. Try it on you all (to get feedback from you)
- Research tasks
  - help me complete the material on the slides (what is missing/would help?)
  - I need to remember "what I say", examples on the board. (take notes)
- Benefits for you (so why should you help?)
  - you will be mentioned in the acknowledgements (for all that is worth)
  - you will help build better course materials (think of next-year's students)

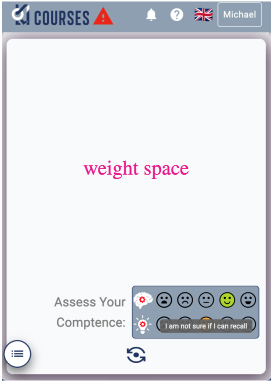
VoLL-KI Portal at `https://courses.voll-ki.fau.de`

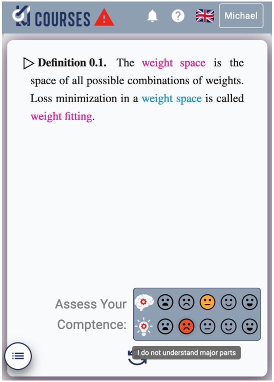
- **Portal for ALeA Courses:** `https://courses.voll-ki.fau.de`

- **AI-1 in ALeA:** `https://courses.voll-ki.fau.de/course-home/ai-1`
  - All details for the course.
  - recorded syllabus (keep track of material covered in course)
  - syllabus of the last semester (for over/preview)
- **ALeA Status:** The ALeA system is deployed at FAU for over 1000 students taking six courses
  - (some) students use the system actively (our logs tell us)
  - reviews are mostly positive/enthusiastic (error reports pour in)

New Feature: Drilling with Flashcards

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

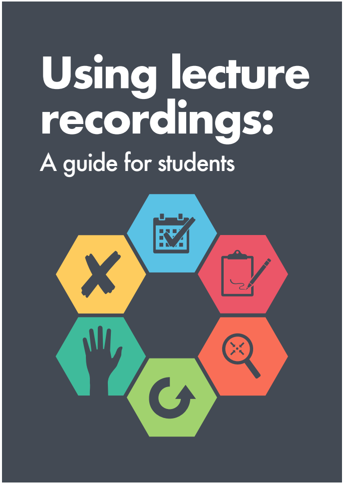


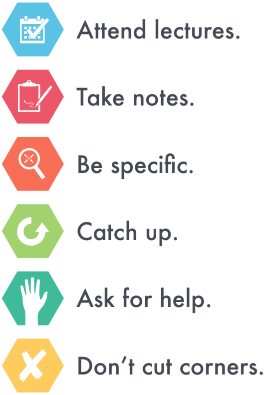


  ...and the definition/answer is on the back.
- Self-assessment updates the learner model (before/after)
- **Idea:** Challenge yourself to a card stack, keep drilling/assessing flashcards until the learner model eliminates all.
- **Bonus:** Flashcards can be generated from existing semantic markup (educational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

Practical recommendations on Lecture Videos

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





- Normally intended for "offline students" $\widehat{=}$ everyone during Corona times.

Software/Hardware tools

- You will need computer access for this course
- we recommend the use of standard software tools
  - find a text editor you are comfortable with (get good with it) A text editor is a program you can use to write text files. (not `MSWord`)
  - any operating system you like (I can only help with `UNIX`)
  - Any browser you like (I use `FireFox`: less spying)
- **Advice:** learn how to touch-type NOW (reap the benefits earlier, not later)
  - you will be typing multiple hours/week in the next decades
  - touch-typing is about twice as fast as "system eagle".
  - you can learn it in two weeks (good programs)

1.2 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 and tools that can become *useful in the short term*
  - generate immediate success and gratification,
  - alleviate the “programming shock” (the brain stops working when in contact with computer science tools or computer scientists) common in the humanities and social sciences.

Academic Culture in Computer Science

- **Definition 2.1.** The academic culture is the overall style of working, research, and discussion in an academic field.
- **Observation 2.2.** *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)

Outline of IWGS 1:

- Programming in Python: (main tool in IWGS)
  - Systematics and culture of programming
  - Program and control structures
  - Basic data strutures like numbers and strings, character encodings, unicode, and regular expressions
- Digital documents and document processing:
  - text files
  - markup systems, HTML, and CSS
  - XML: Documents are trees.
- Web technologies for interactive documents and web applications
  - internet infrastructure: web browsers and servers
  - serverside computing: bottle routing and
  - client-side interaction: dynamic HTML, JavaScript, HTML forms
- Web application project (fill in the blanks to obtain a working web app)

Do I need to attend the lectures

- Attendance is not mandatory for the IWGS lecture

Do I need to attend the lectures

- Attendance is not mandatory for the IWGS lecture
- There are two ways of learning IWGS: (both are OK, your mileage may vary)
  - Approach B: Read a Book
  - Approach I: come to the lectures, be involved, interrupt me whenever you have a question.

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

Do I need to attend the lectures

- Attendance is not mandatory for the IWGS lecture
- There are two ways of learning IWGS: (both are OK, your mileage may vary)
  - Approach B: Read a Book
  - Approach I: come to the lectures, be involved, interrupt me whenever you have a question.

  The only advantage of I over B is that books do not answer questions (yet! ⇝ we are working on this in AI research)
- Approach S: come to the lectures and sleep does not work!

Do I need to attend the lectures

- Attendance is not mandatory for the IWGS lecture
- There are two ways of learning IWGS: (both are OK, your mileage may vary)
  - Approach B: Read a Book
  - Approach I: come to the lectures, be involved, interrupt me whenever you have a question.

  The only advantage of I over B is that books do not answer questions (yet! ⇜ we are working on this in AI research)
- Approach S: come to the lectures and sleep does not work!
- **I really mean it:** If you come to class, be involved, ask questions, challenge me with comments, tell me about errors, ...

Do I need to attend the lectures

- Attendance is not mandatory for the IWGS lecture
- There are two ways of learning IWGS: (both are OK, your mileage may vary)
  - Approach B: Read a Book
  - Approach I: come to the lectures, be involved, interrupt me whenever you have a question.

  The only advantage of I over B is that books do not answer questions (yet! ⇜ we are working on this in AI research)
- Approach S: come to the lectures and sleep does not work!
- **I really mean it:** If you come to class, be involved, ask questions, challenge me with comments, tell me about errors, ...
  - I would much rather have a lively discussion than get through all the slides

Do I need to attend the lectures

- Attendance is not mandatory for the IWGS lecture
- There are two ways of learning IWGS: (both are OK, your mileage may vary)
  - Approach B: Read a Book
  - Approach I: come to the lectures, be involved, interrupt me whenever you have a question.

  The only advantage of I over B is that books do not answer questions (yet! ⇝ we are working on this in AI research)
- Approach S: come to the lectures and sleep does not work!
- **I really mean it:** If you come to class, be involved, ask questions, challenge me with comments, tell me about errors, ...
  - I would much rather have a lively discussion than get through all the slides
  - You learn more, I have more fun (Approach B serves as a backup)

Do I need to attend the lectures

- Attendance is not mandatory for the IWGS lecture
- There are two ways of learning IWGS: (both are OK, your mileage may vary)
  - Approach B: Read a Book
  - Approach I: come to the lectures, be involved, interrupt me whenever you have a question.

  The only advantage of I over B is that books do not answer questions (yet! ⇝ we are working on this in AI research)
- Approach S: come to the lectures and sleep does not work!
- **I really mean it:** If you come to class, be involved, ask questions, challenge me with comments, tell me about errors, ...
  - I would much rather have a lively discussion than get through all the slides
  - You learn more, I have more fun (Approach B serves as a backup)
  - You may have to change your habits, overcome shyness, ... (please do!)

Do I need to attend the lectures

- Attendance is not mandatory for the IWGS lecture
- There are two ways of learning IWGS: (both are OK, your mileage may vary)
  - Approach B: Read a Book
  - Approach I: come to the lectures, be involved, interrupt me whenever you have a question.

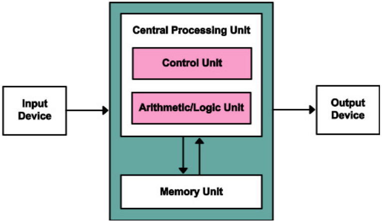
  The only advantage of I over B is that books do not answer questions (yet! ⇝ we are working on this in AI research)
- Approach S: come to the lectures and sleep does not work!
- **I really mean it:** If you come to class, be involved, ask questions, challenge me with comments, tell me about errors, ...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

Chapter 2
Introduction to Programming

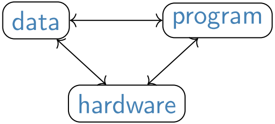
2.1 What is Programming?

Computer Hardware/Software & Programming

- **Definition 1.1.** Computers consist of hardware and software.
- **Definition 1.2.** Hardware consists of
  - 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 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.

Programming Languages

- Programming $\widehat{=}$ writing programs (Telling the computer what to do)

Programming Languages

- Programming $\widehat{=}$ writing programs (Telling the computer what to do)
- *Remark 1.6.* The computer does exactly as told
  - extremely fast extremely reliable
  - completely stupid: will not do what you mean unless you tell it exactly
- Programming can be extremely fun/frustrating/addictive (try it)

Programming Languages

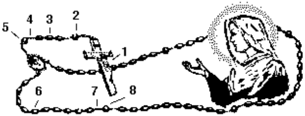
- Programming $\widehat{=}$ writing programs (Telling the computer what to do)
- *Remark 1.8.* The computer does exactly as told
  - extremely fast extremely reliable
  - completely stupid: will not do what you mean unless you tell it exactly
- Programming can be extremely fun/frustrating/addictive (try it)
- **Definition 1.9.** 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)

Programming Languages

- Programming $\widehat{=}$ writing programs (Telling the computer what to do)
- *Remark 1.10.* The computer does exactly as told
  - extremely fast extremely reliable
  - completely stupid: will not do what you mean unless you tell it exactly
- Programming can be extremely fun/frustrating/addictive (try it)
- **Definition 1.11.** 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.

Program Execution

- **Definition 1.12.** Algorithm: informal description of what to do (good enough for humans)
- **Example 1.13.** 



- **Example 1.14.** Program: computer processable version, e.g. in Python.

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

- **Definition 1.15.** Interpreter: reads a program and executes it directly
  - special case: interactive interpretation (lets you experiment easily)
- **Definition 1.16.** Compiler: translates a program (the source) into another program (the binary) in a much simpler programming language for optimized execution on hardware directly.
- *Remark 1.17.* Compilers are efficient, but more cumbersome for development.

2.2 Programming in IWGS

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)

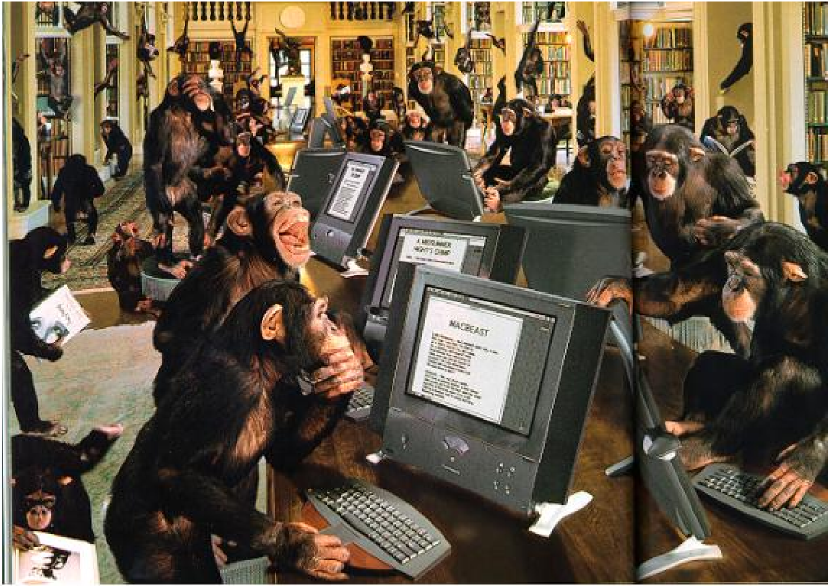
RTFM ($\widehat{=}$ "read those fine manuals")

- **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

But Seriously... Learning programming in IWGS

- The IWGS lecture teaches you
  - a general introduction to programming and Python (next)
  - various useful concepts and how they can be done in Python (in principle)
- The IWGS tutorials
  - teach the actual skill and joy of programming (hacking $\neq$ security breach)
  - supply you with problems so you can practice that.
- **Richard Stallman (MIT) on Hacking:** "What they had in common was mainly love of excellence and programming. They wanted to make their programs that they used be as good as they could. They also wanted to make them do neat things. They wanted to be able to do something in a more exciting way than anyone believed possible and show "Look how wonderful this is. I bet you didn't believe this could be done.""
- **So, ...** Let's hack

2am in the Kollegienhaus CIP Pool



no, let's think

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

2.3 Programming in Python

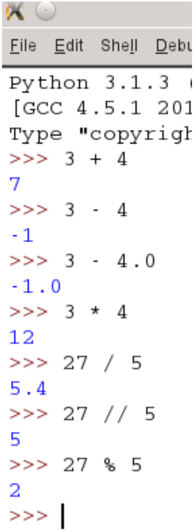
2.3.1 Hello IWGS

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)

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



Comments in Python

- **Generally:** It is highly advisable to insert comments into your programs,
  - especially, if others are going to read your code, (TAs/graders)
  - you may very well be one of the "others" yourself, (in a year's time)
  - writing comments first helps you organize your thoughts.
- Comments are ignored by the Python interpreter but are useful information for the programmer.

Comments in Python

- **Generally:** It is highly advisable to insert comments into your programs,
  - especially, if others are going to read your code, (TAs/graders)
  - you may very well be one of the "others" yourself, (in a year's time)
  - writing comments first helps you organize your thoughts.
- Comments are ignored by the Python interpreter but are useful information for the programmer.
- **In Python:** there are two kinds of comments
  - Single line comments start with a #
  - Multiline comments start and end with three quotes (single or double: `"""` or `'''`)

Comments in Python

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

2.3.2 JupyterLab, a Python Web IDE for IWGS

JupyterLab A Cloud IDE for Python

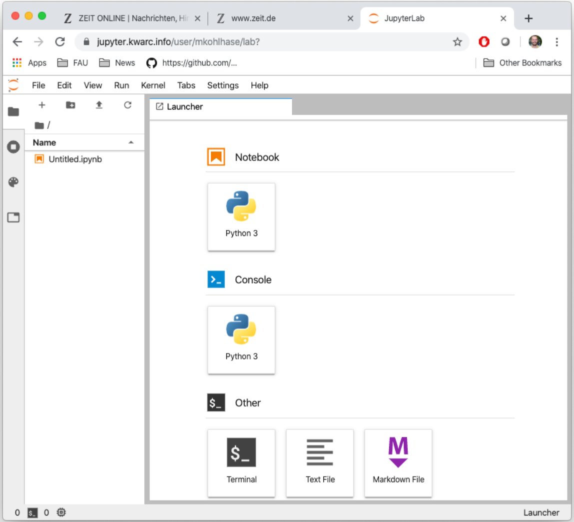
- **For helping you** it would be good if the TAs could access to your code
- **Idea:** Use a web IDE (a web based integrated development environment): JupyterLab, which you can use for interacting with the interpreter.

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`

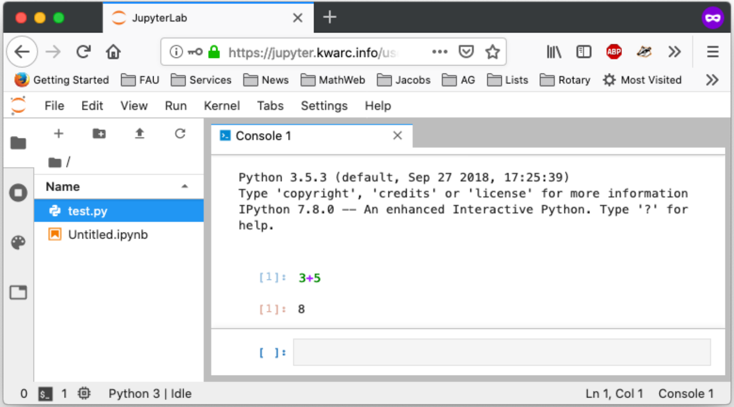
JupyterLab Components

- **Definition 3.2.** The JupyterLab dashboard gives you access to all components.



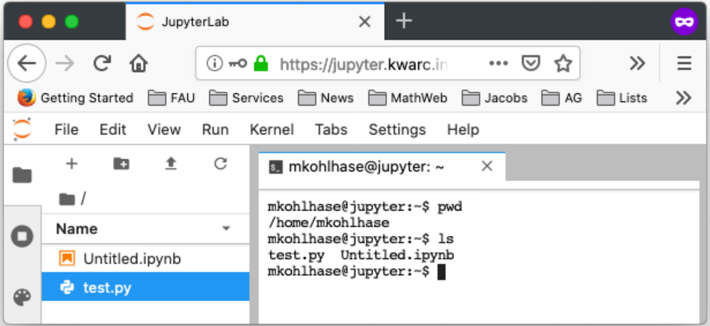
JupyterLab Components

- **Definition 3.6.** The JupyterLab dashboard gives you access to all components.
- **Definition 3.7.** The JupyterLab python console, i.e. a Python interpreter in your browser. (use this for Python interaction and testing.)



JupyterLab Components

- **Definition 3.10.** The JupyterLab dashboard gives you access to all components.
- **Definition 3.11.** The JupyterLab python console, i.e. a Python interpreter in your browser. (use this for Python interaction and testing.)
- **Definition 3.12.** The JupyterLab terminal, i.e. a UNIX shell in your browser.(use this for managing files)

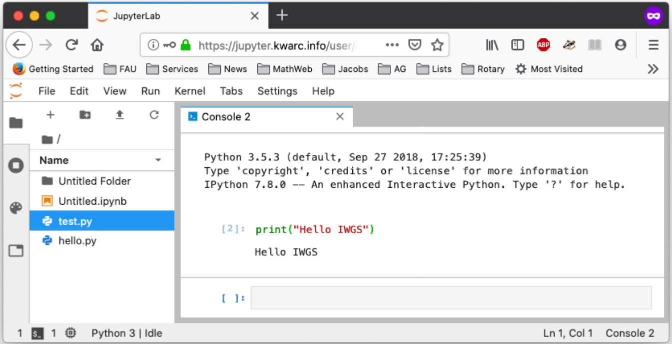


JupyterLab Components

- **Definition 3.14.** The JupyterLab dashboard gives you access to all components.
- **Definition 3.15.** The JupyterLab python console, i.e. a Python interpreter in your browser. (use this for Python interaction and testing.)
- **Definition 3.16.** The JupyterLab terminal, i.e. a UNIX shell in your browser.(use this for managing files)
- **Definition 3.17.** 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 ...

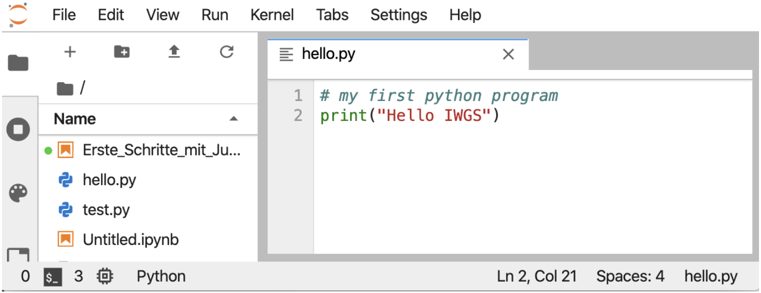
A first program in Python

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



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)

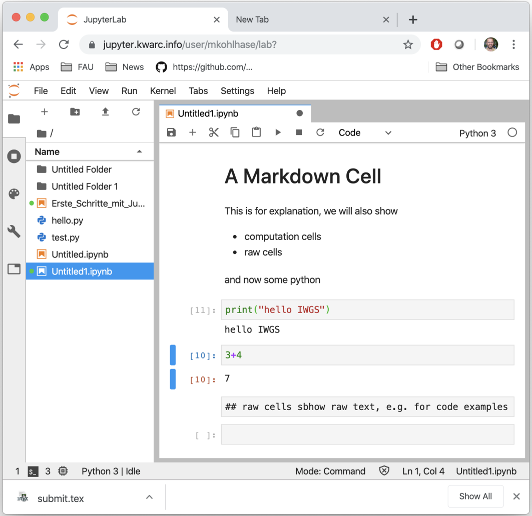
```
import hello
```

jupyter Notebooks

- **Definition 3.18.** Jupyter notebooks are documents that combine live runnable code with rich, narrative text (for comments and explanations).
- **Definition 3.19.** 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.

jupyter Notebooks

- **Example 3.20 (Showing off Cells in a Notebook).**



Markdown a simple Markup Format Generating HTML

.

- **Idea:** We can translate between markup formats.
- **Definition 3.21.** 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 3.22.** 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

Variables in Python

- **Idea:** Values (of expressions) can be given a name for later reference.
- **Definition 3.23.** A variable is an (the variable name) that references a memory location which contains a .
- **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 3.24 (in Python).** A variable assignment $\langle\langle\mathrm{var}\rangle\rangle = \langle\langle\mathrm{val}\rangle\rangle$ assigns a new value to a variable.
- **Example 3.25 (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
>>> |
```

Variables in Python: Extended Example

- **Example 3.26 (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 3.27 (Variables for Storing Intermediate Variables).**

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

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 3.28.** 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.

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 3.29.** *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 3.30 (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

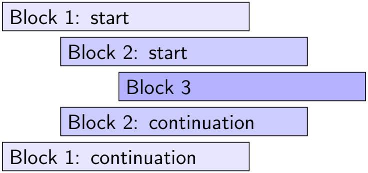
Conditionals and Loops

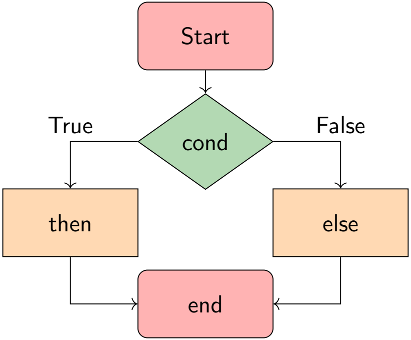
- **Problem:** Up to now programs seem to execute all the instructions in sequence, from the first to the last. (a linear program)
- **Definition 3.31.** The control flow of a program is the sequence of execution of the program instructions. It is specified via special program instructions called control structures.
- **Definition 3.32.** 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 3.33.** A condition is a Boolean expression in a control structure.
- **Definition 3.34.** 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.
- **Example 3.35.** 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

Conditionals in Python

- **Definition 3.36.** Conditional execution via **if**/**else** statements

if ⟪condition⟫ :
    ⟪then − part⟫
else :
    ⟪else − part⟫
⟪morecode⟫





- 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.

Conditional Execution Example

- **Example 3.37 (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.

Variant: Multiple Branches

- Making multiple branches is similar

```
if ⟨⟨condition⟩⟩ :
    ⟨⟨then − part⟩⟩
elif ⟨⟨condition⟩⟩ :
    ⟨⟨otherthen − part⟩⟩
else :
    ⟨⟨else − part⟩⟩
```

  - The there can be more than one **elif** clause.
  - The conditions 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 3.38 (Better Empathy in Python).** In 3.37 we print Good! even if the input is e.g. I feel terrible, so extend **if**/**else** by

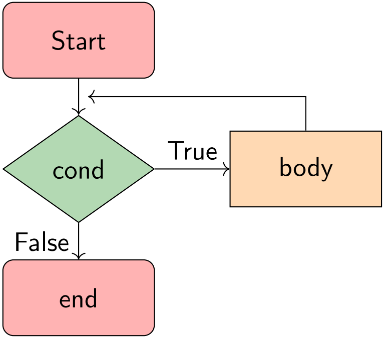
```
elif answer == 'Yes' or answer == 'yes' :
    print("Good!")
else :
    print("I␣do␣not␣understand␣your␣answer")
```

Loops in Python

- **Definition 3.39.** Python makes loops via **while** blocks
  - syntax of the **while** loop

    while ⟨⟨condition⟩⟩ :
        ⟨⟨body⟩⟩
    ⟨⟨morecode⟩⟩

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



Examples of Loops

- **Example 3.40 (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 3.41 (Breaking an unbounded Loop).**

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

Examples of Loops

- **Example 3.42 (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

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)

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 4.5 (Deep Fundamental Result).* There are things no computer can compute.
- **Example 4.6.** There cannot be a program that decides whether another program will terminate in finite time.

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 4.9 (Deep Fundamental Result).* There are things no computer can compute.
- **Example 4.10.** There cannot be a program that decides whether another program will terminate in finite time.
- *Remark 4.11 (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 4.12 (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".

Top Principle of Programming: Compositionality

- **Observation 4.13.** *Modern programming languages compose various primitives and give them a pleasing, concise, and uniform syntax.*
- **Question:** What does all of this even mean?
- **Definition 4.14.** 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 4.15 (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 4.16.** *The semantics of a programming language, is determined by the meaning of its primitives and composition principles.*
- **Definition 4.17.** 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.

Consequences of Compositionality

- **Observation 4.18.** *To understand a programming language, we (only) have to understand its primitives, composition principles, and their syntax.*
- **Definition 4.19.** The "art of programming" consists of composing the primitives of a programming language.
- **Observation 4.20.** *We only need very few – about half a dozen – primitives to obtain a Turing complete programming language.*
- **Observation 4.21.** *The space of program behaviors we can achieve by programming is infinites large nonetheless.*
- *Remark 4.22.* More primitives make programming more convenient.
- *Remark 4.23.* Primitives in one language can be composed in others.

A note on Programming: Little vs. Large Languages

- **Observation 4.24.** *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

2.5.1 Sequences and Iteration

Lists in Python

- **Definition 5.1.** A list is a finite sequence of objects, its element.
- 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 5.2.** We call [⟪seq⟫] the list constructor.
- **Example 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 5.4.** List elements can be accessed by specifying ranges

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

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

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

Sequences in Python

- **Definition 5.5.** Python has more types that behave just like lists, they are called sequence types.
- The most important sequence types for IWGS are lists, strings and ranges.
- **Definition 5.6.** A range is a finite sequence of numbers it can conveniently be constructed by the range function: range(⟪start⟫,⟪stop⟫,⟪step⟫) construts a range from ⟪start⟫ (inclusive) to ⟪stop⟫ (exclusive) with step size ⟪step⟫.
- **Example 5.7.** Lists can be constructed from ranges:

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

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

Iterating over Sequences in Python

- **Definition 5.8.** A for loop iterates a program fragment over a sequence; we call the process iteration. Python uses the following general syntax:

```
for ⟨⟨var⟩⟩ in ⟨⟨range⟩⟩:
    ⟨⟨body⟩⟩
⟨⟨othercode⟩⟩
```

- **Example 5.9.** 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 5.10.** 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)
```

Python Dictionaries

- **Definition 5.11.** 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 5.12.** 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.

Interacting with Dictionaries

- **Example 5.13 (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 5.14 (Printing Keys and Values).**

keys	values	key/value pairs
`for x in thisdict.keys():` `    print(x)`	`for x in thisdict.values():` `    print(x)`	`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.

2.5.2 Input and Output

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)

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

- **Definition 5.15.** A file is a resource for recording data in a storage device. File size is measured in bit.
- **Definition 5.16.** 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 folder 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 5.17.** 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.

Disk Input/Output in Python

- **Definition 5.18.** 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 5.19 (Duplicating the contents of a file).**

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

Disk Input/Output in Python (continued)

- **Example 5.20 (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)

2.5.3 Functions and Libraries in Python

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 5.21.** 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 5.22 (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.

Functions in Python (Definition)

- **Definition 5.23.** 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)
⟨⟨morecode⟩⟩
```

  - 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.

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 3.28 are classes, . . .
- **Definition 5.24.** In Python all values belong to a class, which provide special functions we call methods. Values are also called objects, to emphasise class aspects. Method application is written with dot notation:
  $\langle\langle\mathrm{obj}\rangle\rangle.\langle\langle\mathrm{meth}\rangle\rangle(\langle\langle\mathrm{args}\rangle\rangle)$ corresponds to $\langle\langle\mathrm{meth}\rangle\rangle(\langle\langle\mathrm{obj}\rangle\rangle,\langle\langle\mathrm{args}\rangle\rangle)$.
- **Example 5.25.** 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 5.26 (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).

Python Libraries

- **Idea:** Functions, classes, and methods are re usable, so why not package them up for others to use.
- **Definition 5.27.** 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 (≙ packages on `http://pypi.org`)
  - search for them at `http://pypi.org` (e.g. 815 packages for "music")
  - install them with pip install ⟪packagename⟫
  - look at how they were done (all have links to source code)
  - maybe even contribute back (report issues, improve code, ...) (open source)

2.5.4 A Final word on Programming in IWGS

For more information on Python

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

Chapter 3
Numbers, Characters, and Strings

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

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)

Unary Natural Numbers on the Computer

- **Definition 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 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	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 /)

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 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 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

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

Commonly Used Positional Number Systems

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

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 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 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 1.8 (Convert 456 to base 8).** Result: 710_8

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

3.2 Characters and their Encodings: ASCII and UniCode

The ASCII Character Code

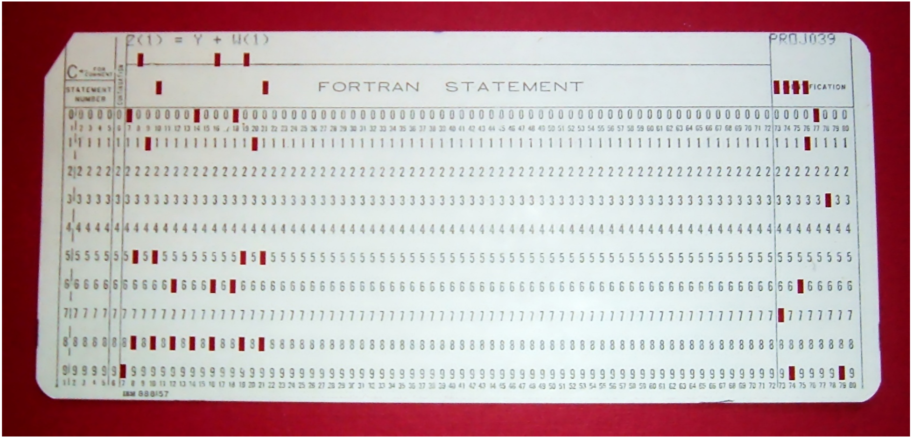
- **Definition 2.1.** The American Standard Code for Information Interchange (ASCII) is a character encoding that assigns characters to numbers 0 127.

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

- The first 32 characters are control characters for ASCII devices like printers.
- **Motivated by 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)

A Punchcard

- **Definition 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 2.3.** This punch card encodes the `FORTRAN` statement $Z(1) = Y + W(1)$



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 2.4 (German ASCII).** Remap e.g. [$\mapsto$Ä,]$\mapsto$Ü in German ASCII ("Apple][" comes out as "`Apple ÜÄ`")
- **Definition 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)

Unicode and the Universal Character Set

- **Definition 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 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 2.8.** Most (non-Chinese) characters have code points in $[1,65536]$: the basic multilingual plane (BMP).
- **Definition 2.9 (Notation).** For code points in the (BMP), four hexadecimal digits are used, e.g. $\mathrm{U}+0058$ for the character $\mathrm{LATINCAPITALLETTERX}$;

Character Encodings in Unicode

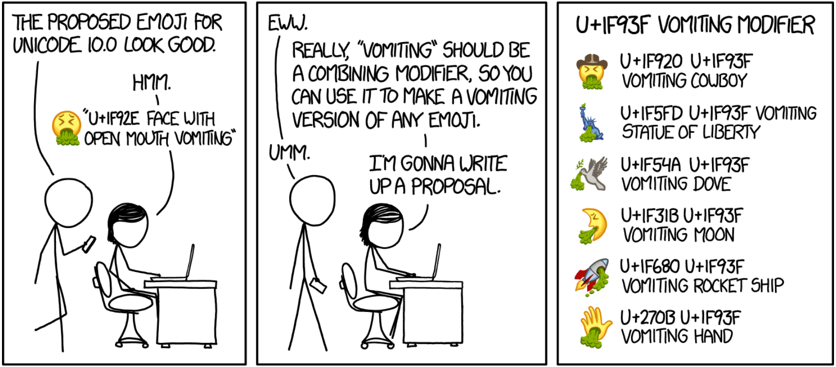
- **Definition 2.10.** A character encoding is a mapping from bit strings to UCS code points.
- **Idea:** Unicode supports multiple character encodings (but not character sets) for efficiency.
- **Definition 2.11 (Unicode Transformation Format).**
  - $\mathrm{UTF}-8$, 8-bit, variable width character encoding, which maximizes compatibility with ASCII.
  - $\mathrm{UTF}-16$, 16-bit, variable width character encoding (popular in Asia)
  - $\mathrm{UTF}-32$, a 32-bit, fixed width character encoding (as a fallback)
- **Definition 2.12.** The $\mathrm{UTF}-8$ 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 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)

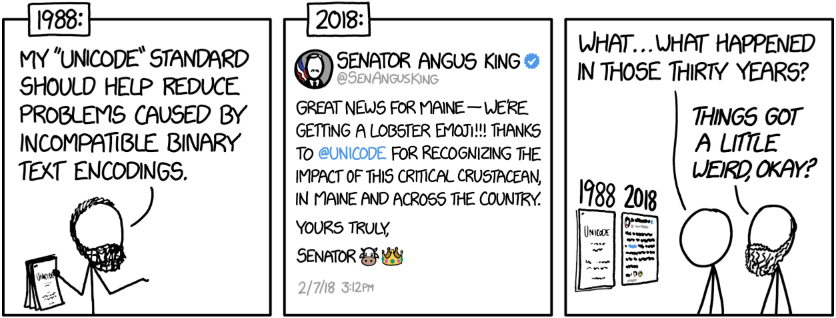
XKCD's Take on Recent Unicode Extensions

- UniCode 6.0 adopted hundreds of emoji characters in 2010 (2666 in July 2017)
- Modifying characters (https://xkcd.com/1813/)



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

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



3.3 More on Computing with Strings

Playing with Strings and Characters in Python

- **Definition 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 3.2 (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 $[i{:}j]$ $\qquad ([i] \mathrel{\widehat{=}} [i{:}i])$
- **Example 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')
```

String Literals in Python

- **Problem:** How to write strings including special characters?
- **Definition 3.4.** A literal is a notation for representing a fixed value for a data structure in source code.
- **Definition 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 3.6.** 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 3.7.** 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.

Unicode in Python

- *Remark 3.8.* The Python string data type is UniCode, encoded as $\mathrm{UTF}-8$.
- **How to write UniCode characters?:** there are five ways
  - write them in your editor (make sure that it uses $\mathrm{UTF}-8$)
  - 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'
```

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)
- **Definition 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 3.10 (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 3.11 (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

Anonymous Functions (lambda)

- **Observation 4.1.** *A Python function definition combines making a function object with giving it a name.*
- **Definition 4.2.** Python also allows to make anonymous functions via the function literal lambda for function objects:

  lambda $p_1,\ldots,p_n$: $\langle\langle\text{expr}\rangle\rangle$

- **Example 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)

Higher-Order Functions in Python

- **Definition 4.4.** We call a function a higher order function, iff it takes a function as argument.
- **Definition 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 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]
```

Argument Passing in Python: Keyword Arguments

- **Definition 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 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")
```

Argument Passing in Python: Flexible Arity

- **Definition 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 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 4.11.** The star operator unpacks a list into an argument sequence.
- **Example 4.12 (Passing a starred list).**

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

Argument Passing in Python: Flexible Keyword Arguments

- **Definition 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 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
```

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

- **Definition 4.15.**3 The double star operator unpacks a dictionary into a sequence of keyword arguments.
- **Example 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 4.17 (Mixing formal and keyword arguments).**

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

3.5 Regular Expressions: Patterns in Strings

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: 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)

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.

Regular Expressions, see [Pyt]

- **Definition 5.1.** A regular expression (also called regex) is a formal expression that specifies a set of strings.
- **Definition 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
$[x-y]/[\hat{}x-y]$	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 \: \?.

Regular Expression Examples

- **Example 5.3 (Regular Expressions and their Values).**

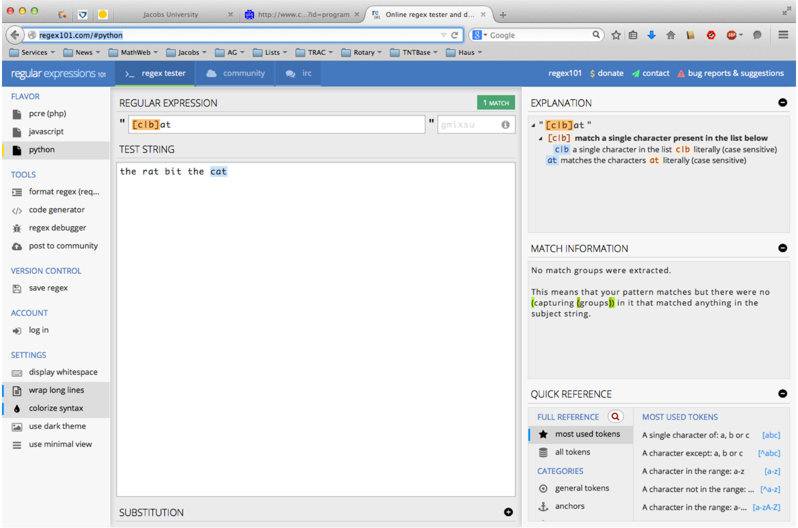
regexp	values
car	car
.at	cat, hat, mat, ...
[hc]at	cat, hat
[^c]at	hat, mat, ... (but not cat)
^[hc]at	hat, cat, but only at the beginning of the line
[0–9]	Digits
[1–9][0–9]∗	natural numbers
(.∗)\1	mama, papa, wakawaka
cat|dog	cat, dog

- 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 5.4 (A more complex example).** The following regex 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)))?$
```

Playing with Regular Expressions

- If you want to play with regexs, go e.g. to `http://regex101.com`



Regular Expressions in Python

- We can use regular expressions directly in Python by importing the re module (just add **import** re at the beginning)
- As Python has UniCode strings, regular expressions support UniCode as well.
- Useful Python functions that use regular expressions.
  - re.findall(⟨⟨pat⟩⟩,⟨⟨str⟩⟩): Return a list of non-overlapping matches of ⟨⟨pat⟩⟩ in ⟨⟨str⟩⟩.

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

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

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

  - re.split(⟨⟨pat⟩⟩,⟨⟨str⟩⟩): Split ⟨⟨str⟩⟩ into substrings that match *pmetavar 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']
    ```

Example: Correcting and Anonymizing Documents

- **Example 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)
```

Example: Correcting and Anonymizing Documents (cont.)

- **Example 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?

Example: Correcting and Anonymizing Documents (all)

- **Example 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

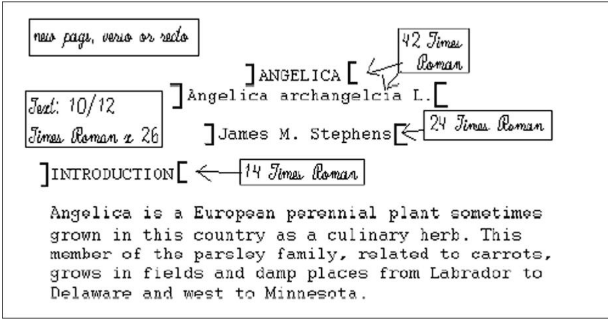
4.1 Representing & Manipulating Documents on a Computer

Electronic Documents

- **Definition 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 1.2.** PDFs, digital images, videos, audio recordings, web pages, ...
- **Definition 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 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 1.5.** Python programs are plain text, PDFs are formatted.

Document Markup

- **Definition 1.6.** Document markup (or just markup) is the process of adding control words (special character sequences also called markup code) 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 1.7.** A text with markup codes (for printing)



- **Definition 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 1.9.* Markup turns plain text into formatted text.

File Types

- **Observation 1.10.** *We mostly encounter electronic documents in the form of files on some storage medium.*
- **Definition 1.11.** A text file is a file that contains text data, a binary file one that contains binary data
- *Remark 1.12.* Text files are usually encoded with ASCII, ISO Latin, or increasingly $\mathrm{UniCode}$ encodings like $\mathrm{UTF}-8$.
- **Example 1.13.** Python programs are stored in text files.
- In practice, text files are often processed as a sequence of text line (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.

Text Editors

- **Definition 1.14.** A text editor is a program used for rendering and manipulating text files.
- **Example 1.15.** Popular text editors include
  - `Notepad` is a simple editor distributed with `Windows`.
  - `emacs` and `vi` are powerful editors originating from `UNIX` and optimized for programming.
  - `sublime` is a sophisticated programming editor for multiple operating systems.
  - `EtherPad` is a browser-based real-time collaborative editor.
- **Example 1.16.** 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.

Word Processors and Formatted Text

- **Definition 1.17.** 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 1.18.** Popular word processors include
  - `MSWord`, an elaborated word processor for `Windows`, whose native format is Office Open XML (OOXML; file extension .docx).
  - `OpenOffice` and `LibreOffice` are similar word processors using the ODF format (Open Office Format; file extension .odf) natively, but can also import other formats..
  - `Pages`, a word processors for `MacOSX` it uses a proprietary format.
  - `OfficeOnline` and `GoogleDocs` are browser-based real-time collaborative word processors.
- **Example 1.19.** Text editor are usually not considered to be word processors, even though they can sometimes be used to edit markup based formatted text.

4.2 Measuring Sizes of Documents/Units of Information

Units for Information

- **Observation:** The smallest unit of information is knowing the state of a system with only two states.
- **Definition 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 2.2.** The byte is a derived unit for information capacity: $1\mathrm{B} = 8\mathrm{b}$.

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 numbers, therefore the derived information units are prefixed by special prefixes that are based on powers of 2.
- **Definition 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 2.4.** You can buy hard-disks that say that their capacity is "one terabyte", but they actually have a capacity of one tebibyte.

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.

4.3 Hypertext Markup Language

4.3.1 Introduction

HTML: Hypertext Markup Language

- **Definition 3.1.** The HyperText Markup Labnguage (HTML), is a representation format for web pages [Hic+14].
- **Definition 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)

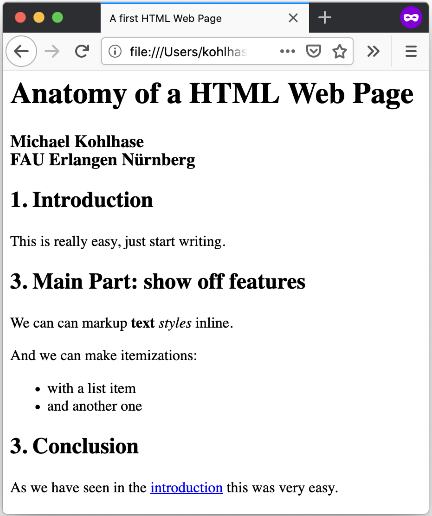
- **Example 3.3.** A (very simple) HTML file with a single paragraph.

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

A very first HTML Example (Result)



4.3.2 Interacting with HTML in Web Broswers

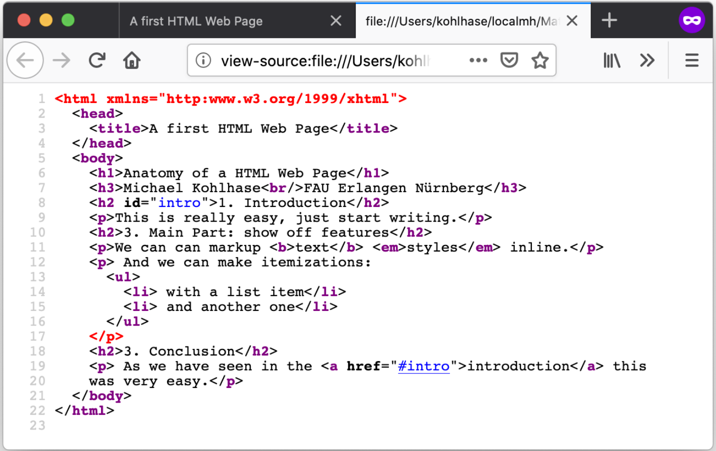
Web Browsers

- **Definition 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 3.5. A web browser usually supplies user tools like
  - history that gives the user access to the
  - an inspector to inspect the DOM

  Definition 3.6. A web browser usually supplies developer tools like
  - the console that logs system-level events in the browser
- **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 3.7 (Common Browsers).**
  - `MSInternetExplorer` is an once dominant, now obsolete browser for `Windows`.
  - `Edge` is provided by Microsoft for `Windows`. (replaces `MSInternetExplorer`)
  - `FireFox` is an open source browser for all platforms, it is known for its standards compliance.
  - `Safari` is provided by Apple for `MacOSX` and `Windows`.
  - `Chrome` is a lean and mean browser provided by Google Inc. (very cor

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

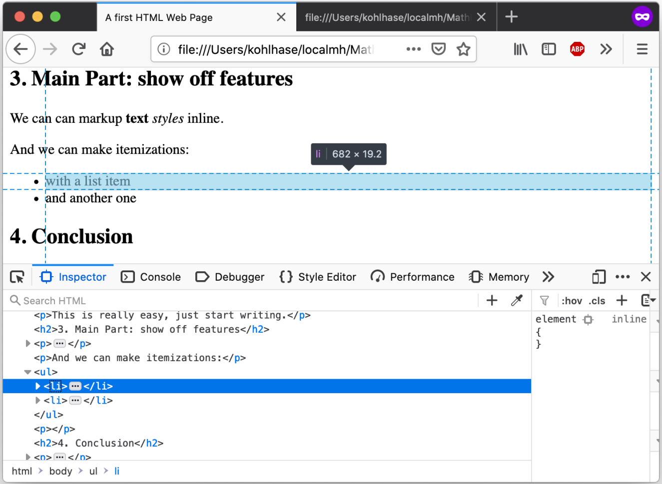
- Hit Control-U to see the page source in the browser




```html
<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 Nürnberg</h3>
    <h2 id="intro">1. Introduction</h2>
    <p>This is really easy, just start writing.</p>
    <h2>3. Main Part: show off features</h2>
    <p>We can can markup <b>text</b> <em>styles</em> inline.</p>
    <p> And we can make itemizations:
      <ul>
        <li> with a list item</li>
        <li> and another one</li>
      </ul>
    </p>
    <h2>3. Conclusion</h2>
    <p> As we have seen in the <a href="#intro">introduction</a> this
    was very easy.</p>
  </body>
</html>
```

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

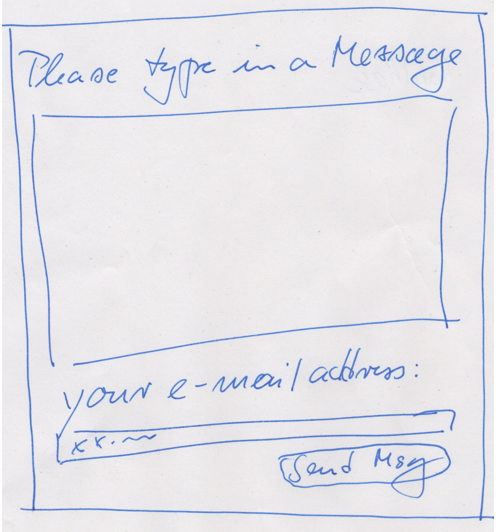
- Hit Control-U to see the page source in the browser
- go to an element and right-click $\leadsto$ "Inspect element"



4.3.3 A Worked Example: The Contact Form

HTML in Practice: Worked Example

- Make a design and "paper prototype" of the page:



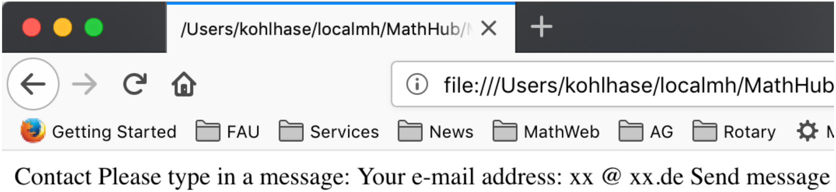
HTML in Practice: Worked Example

- 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
```

HTML in Practice: Worked Example

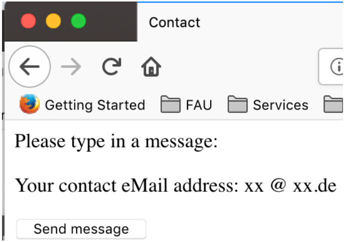
- Make a design and "paper prototype" of the page:
- Put the intended text into a file: contact.html:
- Load into your browser to check the state:



HTML in Practice: Worked Example

- Make a design and "paper prototype" of the page:
- Put the intended text into a file: contact.html:
- Load into your browser to check the state:
- Add title, paragraph and button markup:

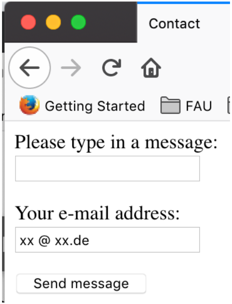
```
<title>Contact</title>
<h2>Please enter a message:</h2>
<h3>Your e—mail address: xx @ xx.de</h3>
<button>Send message</button>
```



HTML in Practice: Worked Example

- Make a design and "paper prototype" of the page:
- Put the intended text into a file: contact.html:
- Load into your browser to check the state:
- Add title, paragraph and button markup:
- 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>
```



HTML in Practice: Worked Example

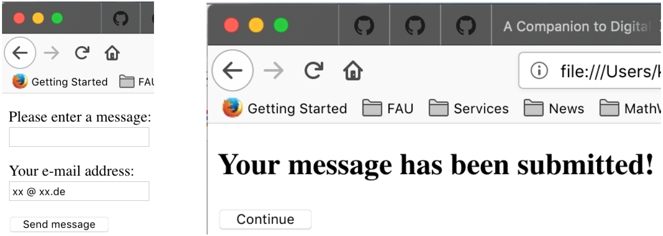
- Make a design and "paper prototype" of the page:
- Put the intended text into a file: contact.html:
- Load into your browser to check the state:
- Add title, paragraph and button markup:
- Add input fields and breaks:
- Convert into a HTML form with action (message receipt):

```
<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>
```

HTML in Practice: Worked Example

- Make a design and “paper prototype” of the page:
- Put the intended text into a file: contact.html:
- Load into your browser to check the state:
- Add title, paragraph and button markup:
- Add input fields and breaks:
- Convert into a HTML form with action (message receipt):



HTML in Practice: Worked Example

- Make a design and "paper prototype" of the page:
- Put the intended text into a file: contact.html:
- Load into your browser to check the state:
- Add title, paragraph and button markup:
- Add input fields and breaks:
- Convert into a HTML form with action (message receipt):
- That's as far as we will go, the rest is page layout and interaction. (up next)

HTML Forms

- **Question:** But how does the interaction with the contact form really work?
- **Definition 3.8.** The HTML form tags groups the layout and input elements:
  - <**form action**="⟪URI⟫"...> specifies the form action (as a web page address).
  - the input element <**input type**="submit".../> triggers the form action: it sends the form data to web page specified there.
- **Example 3.9 (In the Contact Form).** We send the request

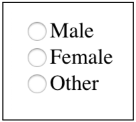
  ```
  GET contact−after.html?
      msg=Hi;addr=foo@bar.de
  ```

  We current ignore the form data (the part after the ?)
- We will come to the full story of processing actions later.

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
```



More useful types of Input fields

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

More useful types of Input fields

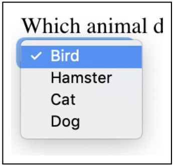
- Radio buttons: type="radio" (grouped by name attribute)
- Check boxes: type="checkbox"
- File selector dialogs (interaction is system specific here for MacOS Mojave)

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

More useful types of Input fields

- Radio buttons: type="radio" (grouped by name attribute)
- Check boxes: type="checkbox"
- File selector dialogs (interaction is system specific here for MacOS Mojave)
- 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

Well-Bracketed Structures in Computer Science

- **Observation 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)*

Well-Bracketed Structures in Computer Science

- **Observation 4.2.** *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>
```

Well-Bracketed Structures in Computer Science

- **Observation 4.3.** *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:*
  - *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?")
```

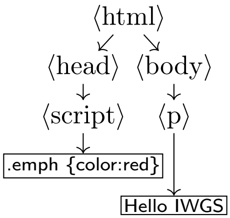
Well-Bracketed Structures in Computer Science

- **Observation 4.4.** *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:*
  - *Programming languages like python:*
- **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 4.5.** *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 4.6.** 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>
```



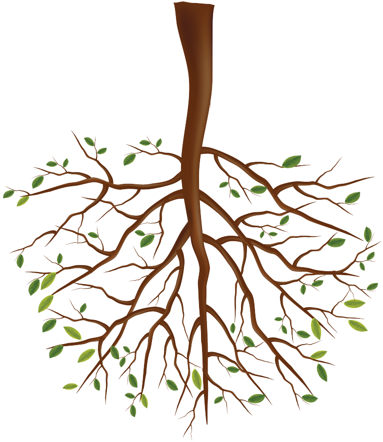
- **Definition 4.7.** We call such structures tree. (more on trees next)

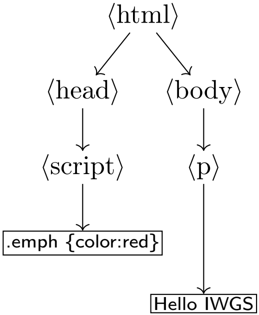
Well-Bracketed Structures: Tree Nomenclature

- **Definition 4.8.** In mathematics and CS, such well-bracketed structures are called trees (with root, branches, leaves, and height). (but written upside down)

Well-Bracketed Structures: Tree Nomenclature

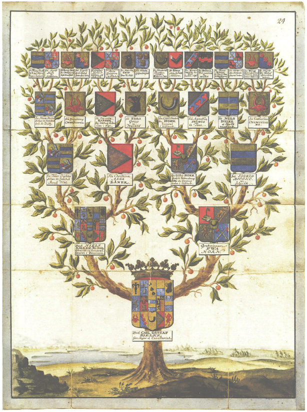
- **Definition 4.11.** In mathematics and CS, such well-bracketed structures are called trees (with root, branches, leaves, and height). (but written upside down)
- **Example 4.12.** In a tree, there is only one path from the root to the leaves

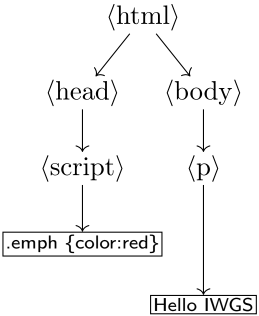




Well-Bracketed Structures: Tree Nomenclature

- **Definition 4.14.** In mathematics and CS, such well-bracketed structures are called trees (with root, branches, leaves, and height). (but written upside down)
- **Example 4.15.** In a tree, there is only one path from the root to the leaves
- **Definition 4.16.** We speak of parent, child, ancestor, and descendant nodes (genealogy nomenclature).





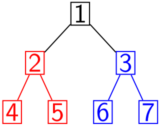
Upside Down Trees in Nature

- 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`

Computing with Trees in Python

- **Observation 4.17.** *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 4.18 (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:]))
```

- **That was a bit cryptic:** i.e. very difficult to read/debug
- **Idea:** why not use dictionaries? (they are more explicit)
- **Example 4.19.** 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
```

The Document Object Model

- **Definition 4.20.** 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

4.5.1 Introduction to XML

XML (EXtensible Markup Language)

- **Definition 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)

XML is Everywhere (E.g. Web Pages)

- **Example 5.2.** Open web page file in `FireFox`, then click on *View* ↘ *PageSource*, you get the following text: (showing only a small part and reformatting)

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

- **Definition 5.3.** XHTML is the XML version of HTML. (just make it valid XML)

XML is Everywhere (E.g. Catalogs)

- **Example 5.4 (The NYC Galleries Catalog).** A public XML file at `https://data.cityofnewyork.us/download/kcrmj9hh/application/xml`

```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>
```

XML is Everywhere (E.g. Office Suites)

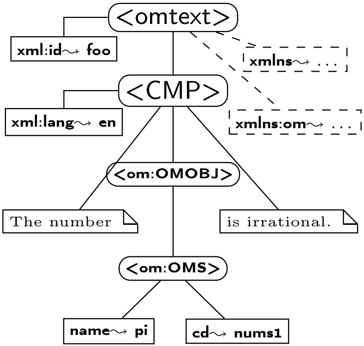
- **Example 5.5 (MS Office uses XML).** The MSOffice suite and LibreOffice use compressed XML as an electronic document format.
  1. Save a MSOffice file test.docx, add the extension .zip to obtain test.docx.zip.
  2. Uncompress with unzip (UNIX) or open File Explorer, right-click $\leadsto$ "Extract All" (Windows)
  3. You obtain a folder with 15+ files, the content is in word/contents.xml
  4. Other files have packaging information, images, and other objects.

  ⚠ This is huge and offensively ugly.
  - But you have everything you wanted and more
  - 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 5.6.** The XML document tree is made up of element nodes, attribute nodes, text nodes (and namespace declarations, comments,...)

XML Documents as Trees (continued)

- **Definition 5.7.** For communication this tree is serialized into a balanced bracketing structure, where
  - an inner element node is represented by the brackets <el> (called the opening tag) and </el> (called the closing tag),
  - the leaves of the XML tree are represented by empty element tags (serialized as <el></el>, which can be abbreviated as <el/>,
  - and text node (serialized as a sequence of UniCode characters).
  - An element node can be annotated by further information using attribute nodes serialized as an attribute in its opening tag.
- **Note:** As a document is a tree, the XML specification mandates that there must be a unique document root.

4.5.2 Computing with XML in Python

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)

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
```

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
```

Computing with XML in Python (Elements)

- The lxml library [LXMLa] provides Python bindings for the (low-level) LibXML2 library. (install it with pip3 install lxml)
- The ElementTree API is the main way to programmatically interact with XML. Activate it by importing etree from lxml:

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

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

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

```
>>> print(root.tag)
root
```

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

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

Computing with XML in Python (Elements)

- The lxml library [LXMLa] provides Python bindings for the (low-level) LibXML2 library. (install it with pip3 install lxml)
- The ElementTree API is the main way to programmatically interact with XML. Activate it by importing etree from lxml:

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

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

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

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

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

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

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

Computing with XML in Python (Result)

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

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

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

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

The lxml API documentation [LXMLb] has the details.

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.

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"/>'
```

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')]
```

Computing with XML in Python (Text nodes)

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

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

Case Study: Creating an HTML document

- We create nested html and body element

```
>>> 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>'
```

Computing with XML in Python (XML Literals)

- **Definition 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

XML is Everywhere (E.g. document metadata)

- **Example 5.9.** Open a PDF file in `AcrobatReader`, then click on

  File ↘ *DocumentProperties* ↘ *DocumentMetadata* ↘ *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 5.10.** 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

Mixing Vocabularies via XML Namespaces

- **Problem:** We would like to reuse elements from different XML vocabularies
  What happens if elements names coincide, but have different meanings?
- **Idea:** Disambiguate them by vocabulary name. (prefix)

Mixing Vocabularies via XML Namespaces

- **Problem:** We would like to reuse elements from different XML vocabularies
  What happens if elements 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

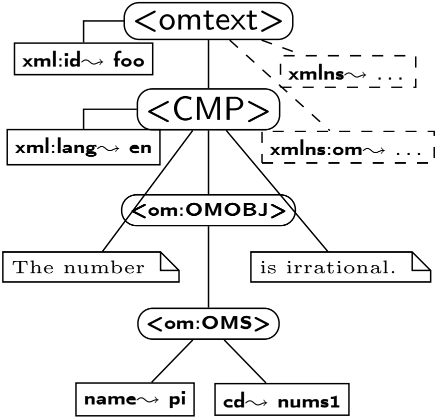
Mixing Vocabularies via XML Namespaces

- **Problem:** We would like to reuse elements from different XML vocabularies What happens if elements 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 5.15.** An XML namespace is a string that identifies an XML vocabulary. Every elements and attribute name in XML consists of a local name and a namespace.
- **Definition 5.16.** A namespace declaration is an attribute xmlns:prefix|=| whose value is an XML namespace *n* on an XML element *e*. The first associates the namepsace prefix prefix with the namespace *n* in *e*: Then, any XML element in *e* with a prefixed name $\langle\langle\text{prefix}\rangle\rangle$:$\langle\langle\text{name}\rangle\rangle$ has namespace *n* and local name $\langle\langle\text{name}\rangle\rangle$.
  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.

4.5.4 XPath: Specifying XML Subtrees

XPath, A Language for talking about XML Tree Fragments

- **Definition 5.17.** The XML path language (XPath) is a language framework for specifying fragments of XML trees.
- **Intuition:**
  XPath is for trees what regular expressions are for strings.
- **Example 5.18.**



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

Computing with XML in Python (XPath)

- Say we have an XML tree:

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

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'
```

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'
```

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
```

XPath Example: Scraping Wikipedia

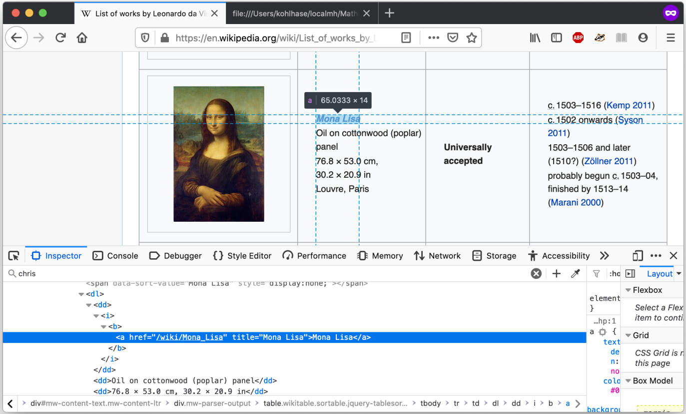
- **Example 5.19 (Extracting Information from HTML).**
  - We want a list of all titles of paintings by Leonardo da Vinci.

XPath Example: Scraping Wikipedia

- **Example 5.20 (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)

XPath Example: Scraping Wikipedia

- **Example 5.21 (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)



XPath Example: Scraping Wikipedia

- **Example 5.22 (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, $\leadsto$ "copy" $\leadsto$ "XPath", gives /html/body/div[3]/div[3]/div[4]/div/table[4]/tbody/tr[3]/td[2]/dl/dd/i/b/a.
  - **Idea**: We want to use the second table cells td[2].
  - Program it in Python using the lxml library: titles is list of title strings.

```
from lxml import html

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

Chapter 5
Web Applications

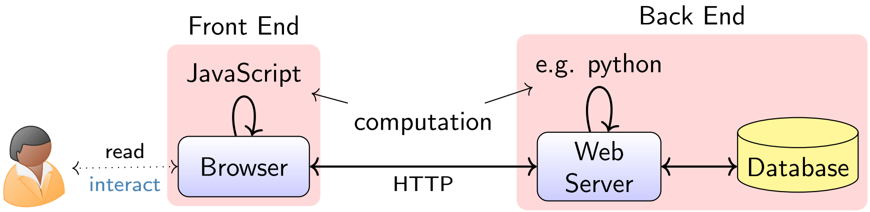
5.1 Web Applications: The Idea

Web Applications: Using Applications without Installing

- **Definition 1.1.** A web application is a program that runs on a web server and delivers its user interface as a web site consisting of programmatically generated web pages using a web browser as the client.
- **Example 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.

Anatomy of a Web Application

- **Definition 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)

5.2 Basic Concepts of the World Wide Web

5.2.1 Preliminaries

The Internet and the Web

- **Definition 2.1.** The Internet is a global computer network that connects hundreds of thousands of smaller networks.
- **Definition 2.2.** The World Wide Web (WWW) is an open source information space where documents and other web resources are identified by URLs, interlinked by hypertext links, and can be accessed via the Internet.
- **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)

Concepts of the World Wide Web

- **Definition 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 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 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 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

Uniform Resource Identifier (URI), Plumbing of the Web

- **Definition 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 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).

Relative URIs

- **Definition 2.9.** URIs can be abbreviated to relative URIs; missing parts are filled in from the context.
- **Example 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 2.11.** To distinguish them from relative URIs, we call URIs absolute URIs.

Uniform Resource Names and Locators

- **Definition 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 2.13.**
  The following URI is a URL (try it in your browser)
  http://kwarc.info/kohlhase/index.html
- **Example 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)

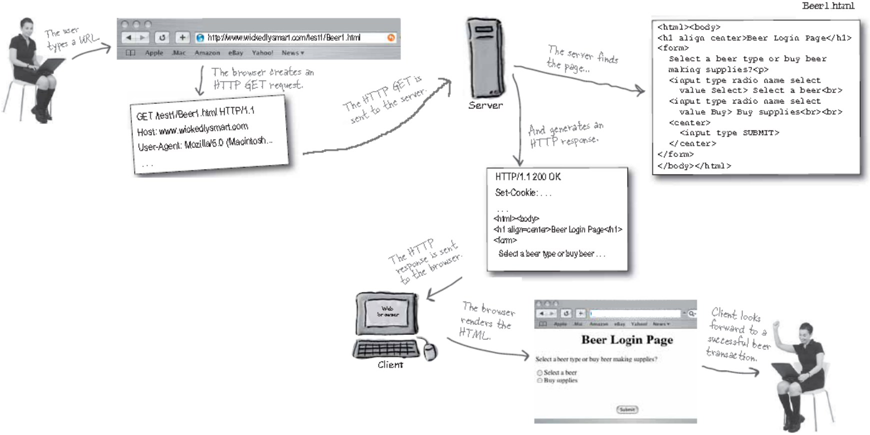
Internationalized Resource Identifiers

- *Remark 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 2.16.** Internationalized resource identifiers (IRIs) extend the ASCII-based URIs to the universal character set.
- **Definition 2.17.** URI encoding maps non-ASCII characters to ASCII strings:
  1. Map each character to its $\mathrm{UTF}-8$ representation.
  2. Represent each byte of the $\mathrm{UTF}-8$ 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 2.18.** The letter "ł" ($\mathrm{U}+142$) would be represented as `%C5%82`.
- **Example 2.19.** `http://www.Übergrößen.de` becomes `http://www.%C3%9Cbergr%C3%B6%C3%9Fen.de`
- *Remark 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 World Wide Web as a Client/Server System



HTTP: Hypertext Transfer Protocol

- **Definition 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 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 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 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.)

Web Servers

- **Definition 2.25.** Ein Web Server ist ein Netzwerk Programm (ein Server in der Client/Server Architektur des WWW) das über das Hypertext Transfer Protocol (HTTP) Web Resourcen an den Client ausliefert und Inhalte von ihm from erhält.
- **Example 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 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 `Windows` [Xam].

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

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 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>
```

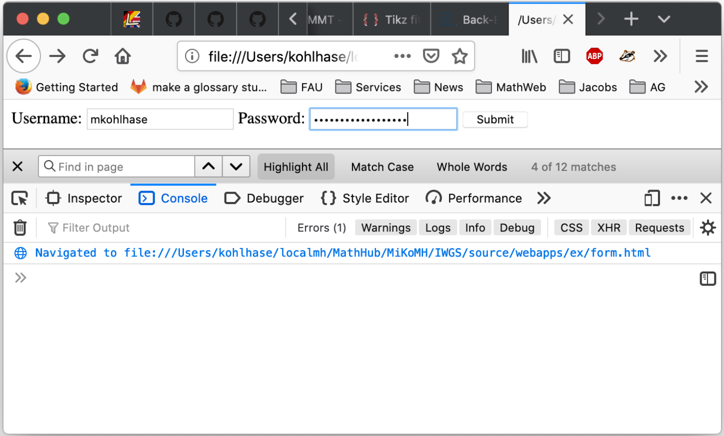
yields the following in a web browser:

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)

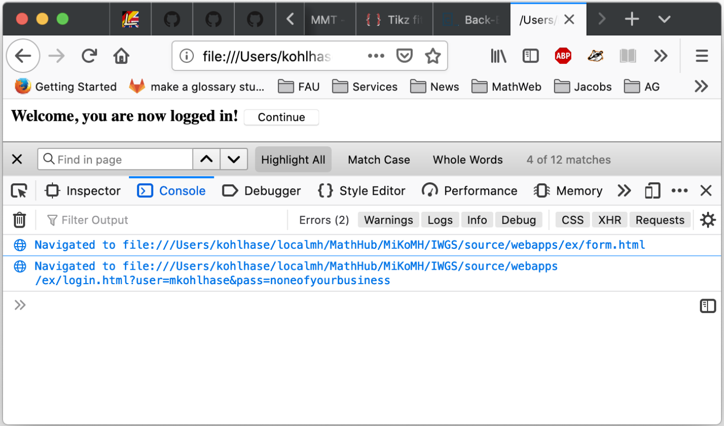
Checking up on the Transmission

- 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)



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)



HTML Forms and Form Data Transmission

- We specify the HTTP communication of HTML forms in detail.
- **Definition 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.

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

Server-Side Scripting: Programming Web pages

- **Idea:** Why write HTML pages if we can also program them! (easy to do)
- **Definition 4.1.** A server-side scripting framework is a web server extension that generates web pages upon HTTP requests.
- **Example 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 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.

5.4.1 Routing and Argument Passing in Bottle

The Web Server and Routing in Bottle WSGI

- **Definition 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 given in ⟨⟨keys⟩⟩.
  - The @route decorator connects path components to Python function that return strings.
- **Example 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`

Dynamic Routes in Bottle

- **Definition 4.6.** A dynamic route is a route annotation that contains named wildcards, which can be picked up in the route function.
- **Example 4.7.** Multiple @route annotations per route function f are allowed $\rightsquigarrow$ 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)

Restricting Dynamic Routes

- **Definition 4.8.** A dynamic route can be restricted by a route filter to make it more selective.
- **Example 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 4.10 (A route with two wildcards).**

```
@route('/<action>/<user:re:[a-z]+>') # matches /follow/miko
def user_api(action, user):
    ...
```

Method-Specific Routes: HTTP GET and POST

- **Definition 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 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)

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 4.13 (Login 2).**
  Continuing from 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.

5.4.2 Templating in Python via STPL

What would we do in Python

- **Example 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 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)

Template Processing for HTML

- **Definition 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 4.17.** PHP (originally "Programmable Home Page Tools") is a very successful server-side scripting framework following this model.

stpl: the "Simple Template Engine" from Bottle

- **Definition 4.18.** Bottle WSGI supplies the template engine stpl (Simple Template Engine). (documentation at [STPL])
- **Definition 4.19.** A template engine for a document format F is a program that transforms templates, i.e. strings or files with 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).
- 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'
```

stpl Syntax and Template Files

- **But what about. . . :** HTML files with embedded Python?
- stpl uses template files (extension .tpl) for that.
- **Definition 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 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>
```

Template Functions

- **Definition 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⟫)

Template Functions

- **Definition 4.25.** 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 4.26 (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')
```

Template Functions

- **Definition 4.28.** 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 4.29 (Including Header and Footer in a template).** In a coherent web site, the web pages often share common header and footer parts. Realize this via the following page template:

```
% include('header.tpl', title='Page Title')
 ... Page Content ...
% include('footer.tpl')
```

- **Example 4.30 (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
```

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)

State in Web Applications and Cookies

- **Recall:** Web applications contain multiple pages, HTTP is a stateless protocol.
- **Problem:** How do we pass state between pages? (e.g. username, password)
- **Simple Solution:** Pass information along in query part of page URLs.
- **Example 4.34 (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.

State in Web Applications and Cookies

- **Recall:** Web applications contain multiple pages, HTTP is a stateless protocol.
- **Problem:** How do we pass state between pages? (e.g. username, password)
- **Simple Solution:** Pass information along in query part of page URLs.
- **Example 4.37 (HTTP GET for Single Login).** Since we are generating pages we can generated augmented links
- **Problem:** Only works for limited amounts of information and for a single session.
- **Other Solution:** Store state persistently on the client hard disk.
- **Definition 4.38.** 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.

State in Web Applications and Cookies

- **Recall:** Web applications contain multiple pages, HTTP is a stateless protocol.
- **Problem:** How do we pass state between pages? (e.g. username, password)
- **Simple Solution:** Pass information along in query part of page URLs.
- **Example 4.40 (HTTP GET for Single Login).** Since we are generating pages we can generated augmented links
- **Problem:** Only works for limited amounts of information and for a single session.
- **Other Solution:** Store state persistently on the client hard disk.
- **Definition 4.41.** A cookie is a text file stored on the client hard disk by the web browser. Web servers can request the browser to store and send cookies.
- **Note:** Cookies are data, not programs, they do not generate pop ups or behave like viruses, but they can include your log-in name and browser preferences.
- **Note:** Cookies can be convenient, but they can be used to gather information about you and your browsing habits.
- **Definition 4.42.** Third-party cookies are used by advertising companies to track users across multiple sites. (but you can turn off, and even delete cookies)

5.4.3 Completing the Contact Form

Back to our Contact Form (Current State)

- A contact form and message receipt (communicate via HTTP requests)

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>
```

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>
```

Back to our Contact Form (Current State)

- A contact form and message receipt (communicate via HTTP requests)

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
```

Back to our Contact Form (Current State)

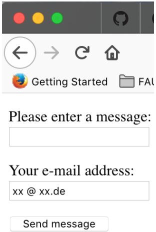
- A contact form and message receipt (communicate via HTTP requests)

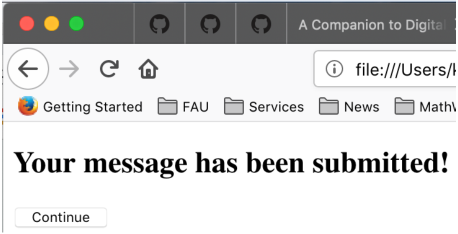
contact4.html

```
<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>
```

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>
```





Back to our Contact Form (Current State)

- A contact form and message receipt (communicate via HTTP requests)

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>
```

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>
```

- **Problem:** The answer is a static HTML document independent of form data.
- **Solution:** Generate the answer programmatically using the form data. (up next)

Completing the Contact Form

- 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 4.43 (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>
```

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 4.44 (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")

Chapter 6
Frontend Technologies

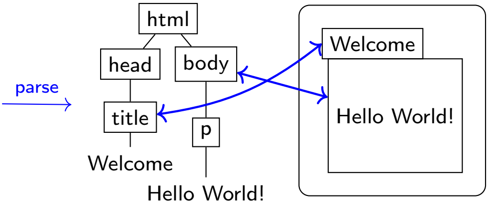
6.1 Dynamic HTML: Client-side Manipulation of HTML Documents

Background: Rendering Pipeline in browsers

- **Observation:** The nested markup codes turn HTML documents into trees.
- **Definition 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)

HTML Document **DOM** **Browser**

```
<html>
  <head>
    <title>Welcome</title>
  </head>
  <body>
    <p>Hello World!</p>
  </body>
</html>
```



The DOM is notified of any user events (resizing, clicks, hover,. . .)

6.1.1 JavaScript in HTML

Dynamic HTML

- **Idea:** generate parts of the web page dynamically by manipulating the DOM.
- **Definition 1.2.** JavaScript is an object-oriented scripting language mostly used to enable programmatic access to the DOM in a web browser.
- JavaScript is standardized by ECMA in [Ecm].
- **Example 1.3.** We write the some text into a HTML document object (the document API)

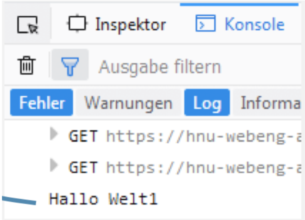
```
<html>
 <head>
  <script type="text/javascript">document.write("Dynamic␣HTML!");</script>
 </head>
 <body><!-- nothing here; will be added by the script later --></body>
</html>
```

- **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

Browser-level JavaScript functions: 1

- **Example 1.4 (Logging to the browser console).**

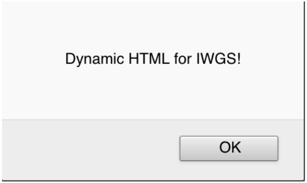
```
console.log("hello IWGS")
```



Browser-level JavaScript functions: 2

- **Example 1.6 (Raising a Popup).**

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



Browser-level JavaScript functions: 3

- **Example 1.7 (Asking for Confirmation).**

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

Embedding JavaScript into HTML

- In a <script> element in HTML, e.g.

```
<script type="text/javascript">
  function sayHello() { console.log('Hello IWGS!'); }
</script>
```

- External JavaScript file via a <**script**> element with src

```
<script type="text/javascript" src="../js/foo.js"/>
```

  Advantage: HTML and JavaScript code are clearly separated

- In event attributes of various HTML elements, e.g.

```
<input type="button" value="Hallo" onclick="alert('Hello IWGS')"/>
```

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

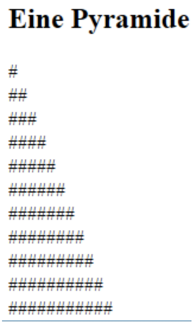
```
<a href="javascript:..."/>
```

Example: Changing Web Pages Programmatically

- **Example 1.9 (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>
```



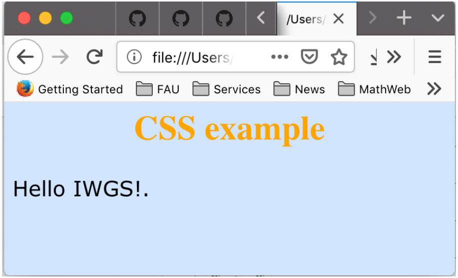
6.2 Cascading Stylesheets

6.2.1 Separating Content from Layout

CSS: Cascading Style Sheets

- **Idea:** Separate structure/function from appearance.
- **Definition 2.1.** Cascading Style Sheets (CSS) is a style sheet language that allows authors and users to attach style (e.g., fonts, colors, and spacing) to HTML and XML documents.
- **Example 2.2.** Our text file from 3.3 with embedded CSS:

```
<html>
  <head>
    <style type="text/css">
       body {background-color:#d0e4fe;}
       h1 {color:orange;
             text-align:center;}
       p {font-family:"Verdana";
             font-size:20px;}
    </style>
  </head>
  <body>
    <h1>CSS example</h1>
    <p>Hello IWGS!.</p>
  </body>
</html>
```



CSS: Rules, Selectors, and Declarations

- **Definition 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 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 2.5.** In 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.

A Styled HTML Title Box (Source)

- **Example 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;}
```

A Styled HTML Title Box (Result)

6.2.2 A small but useful Fragment of CSS

CSS Selectors

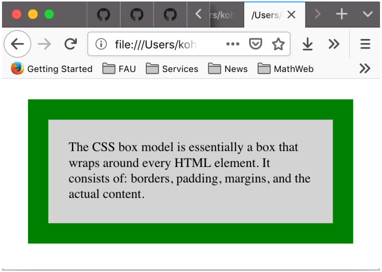
- **Question:** Which elements are affected by a CSS rule?
- Elements of a given name (optionally with given attributes)
  - Selectors: name $\widehat{=}$ ⟨⟨elname⟩⟩, attributes $\widehat{=}$ [⟨⟨attname⟩⟩=⟨⟨attval⟩⟩]
- **Example 2.7.** p[xml:lang='de'] applies to <**p** xml:**lang**="de">...</**p**>
- Any elements with a given class attributes
  - Selector: .⟨⟨classname⟩⟩
- **Example 2.8.** .important applies to <⟨⟨el⟩⟩ class='important'>...</⟨⟨el⟩⟩>
- The element with a given id attribute
  - Selector: #⟨⟨id⟩⟩
- **Example 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`.

The CSS Box Model

- **Definition 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:

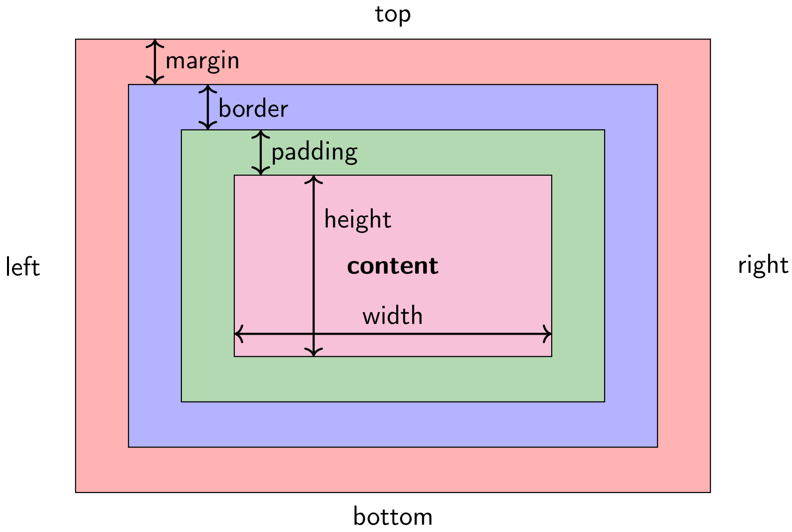
```
div {
    background—color: lightgrey;
    width: 300px;
    border: 25px solid green;
    padding: 25px;
    margin: 25px;
}
```



Note that the overall width of the CSS box is $300 + 2 \cdot 3 \cdot 25 = 450$ pixels.

The CSS Box Model: Diagram

- The following diagram summarizes the CSS box model



Cascading of selectors in CSS: Prioritization

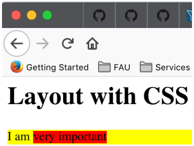
- Multiple CSS selectors apply with the following priorities:
  1. important (i.e. marked with !important) before unimportant
  2. inline (specified via the style attribute)
  3. media-specific rules before general ones
  4. user-defined CSS stylesheet (e.g. in the `FireFox` profile)
  5. specialized before general selectors (complicated; see e.g. [CSS])
  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

Cascading of selectors in CSS: Prioritization Example

- **Example 2.11.** 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}
```



Cascading in CSS: Inheritance

- **Definition 2.12.** If an element is fully contained in another, the inner inherits some properties (called inheritable) of the outer. In a nutshell
  - text-related properties are inheritable; e.g. color, font, letter—spacing, line—height, list—style, and text—align
  - box-related properties are not; e.g. background, border, display, float, clear, height, width, margin, padding, position, and text—align.
- **Note:** Inheritance is integrated into prioritization (recall case 7. above)
- Inheritance makes for consistent text properties and smaller CSS stylesheets.

CSS-Flow: How Boxes Flow to their Place

- CSS Flow describes how different elements are distributed in the visible area (how they flow; hence the name)
- **Example 2.13.** Block-level Boxes (here divs) flow to the left

```
<div class="square">1</div>
<div class="square">2</div>
<div class="square">3</div>
<div class="square">4</div>
```

\+

```
.square {font−size:200%;
         height:100px;
         width:100px;
         border:1px solid black;
         margin:2px;
         background−color:orange;}
```

=

CSS-Flow: How Boxes Flow to their Place

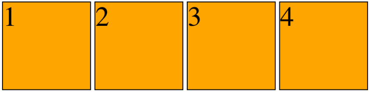
- CSS Flow describes how different elements are distributed in the visible area (how they flow; hence the name)
- **Example 2.17.** Block-level Boxes (here divs) flow to the left
- **Example 2.18.** float:left floats boxes as far as they will go (without overlap)

```
<div class="square">1</div>
<div class="square">2</div>
<div class="square">3</div>
<div class="square">4</div>
```

+

```
.square {font-size:200%;
         height:100px;
         width:100px;
         border:1px solid black;
         margin:2px;
         background-color:orange;
         float:left}
```

=



CSS-Flow: How Boxes Flow to their Place

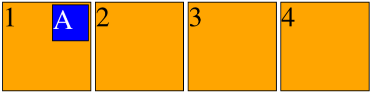
- CSS Flow describes how different elements are distributed in the visible area (how they flow; hence the name)
- **Example 2.21.** Block-level Boxes (here divs) flow to the left
- **Example 2.22.** float:left floats boxes as far as they will go (without overlap)
- **Example 2.23.** float:right in a div will float inside the corresponding box

```
<div class="square">1
  <div class="smallsq">A</div>
</div>
<div class="square">2</div>
<div class="square">3</div>
<div class="square">4</div>
```

+

```
.smallsq {color:white;
  height: 40px;width: 40px;
  border: 1px solid black;
  margin: 2px;
  background-color: blue;
  float: right}
```

=



CSS-Flow: How Boxes Flow to their Place

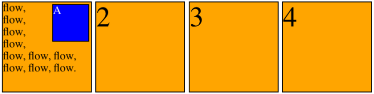
- CSS Flow describes how different elements are distributed in the visible area (how they flow; hence the name)
- **Example 2.25.** Block-level Boxes (here divs) flow to the left
- **Example 2.26.** float:left floats boxes as far as they will go (without overlap)
- **Example 2.27.** float:right in a div will float inside the corresponding box
- **Example 2.28.** float:left will let contents flow around an obstacle

```
<div class="square"
  style="font-size:small">
  <div class="smallsq">A</div>
  flow, flow, flow, flow, flow,
  flow, flow, flow, flow, flow.
</div>
```

+

```
.smallsq {color:white;
  height: 40px;width: 40px;
  border: 1px solid black;
  margin: 2px;
  background-color: blue;
  float: right}
```

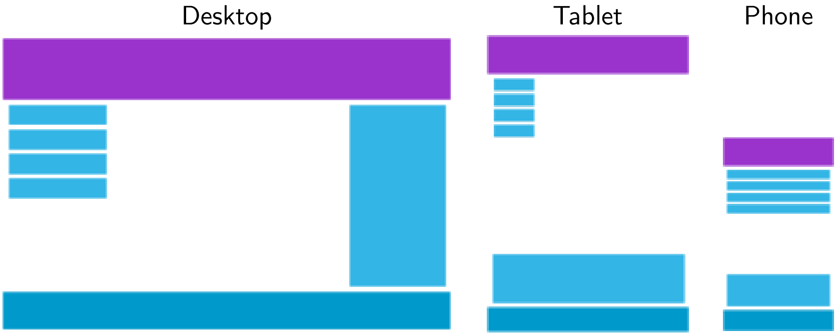
=



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

CSS Application: Responsive Design

- **Problem:** What is the screen size/resolution of my device?
- **Definition 2.29.** 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 2.30.** A web page with content blocks

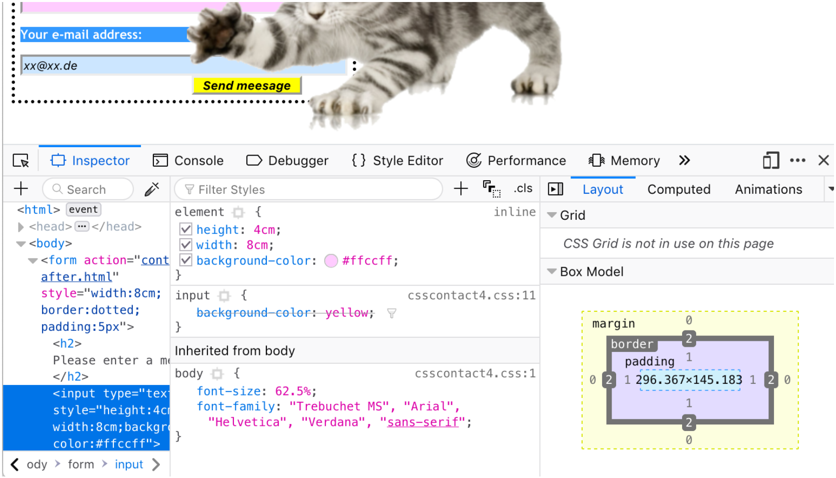


- **Implementation:** CSS based layout with relative sizes and media queries– CSS conditionals based on client screen size/resolution/...

6.2.3 CSS Tools

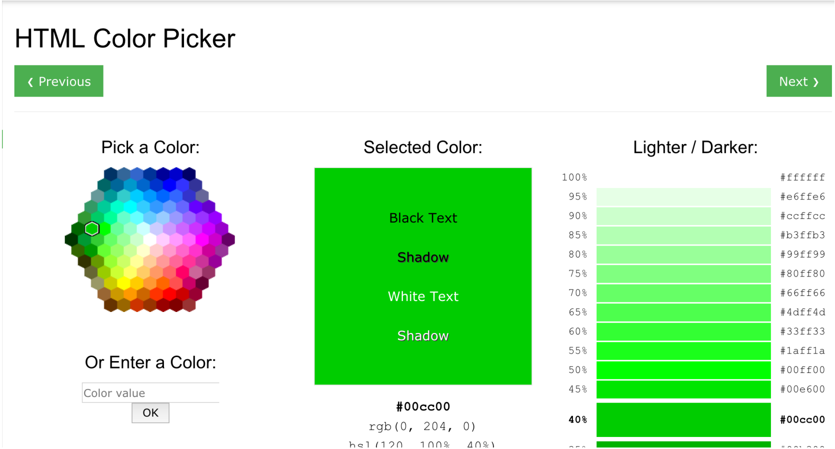
But how to find out what the browser really sees?

- CSS has many interesting inheritance rules
- **Definition 2.31.** The page inspector tool gives you an overview over the internal state of the browser.
- **Example 2.32.**



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`

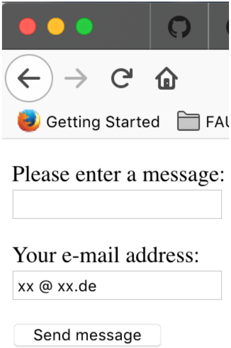


6.2.4 Worked Example: The Contact Form

CSS in Practice: The Contact Form Example (Continued)

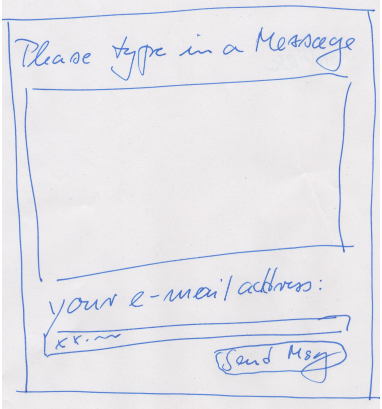
- Recap: The unstyled contact form –

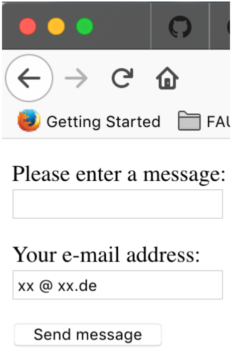
```
<title>Contact</title>
<form action="contact-after.html">
  <h2>Please enter a message:</h2>
    <input name="msg" type="text"/>
  <h3>Your e-mail address:</h3>
  <input name="addr" type="text"
           value="xx␣@␣xx.de"/>
  <br/>
  <input type="submit"
           value="Send␣message"/>
</form>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



CSS in Practice: The Contact Form Example (Continued)

- Recap: The unstyled contact form – Dream vs. Reality



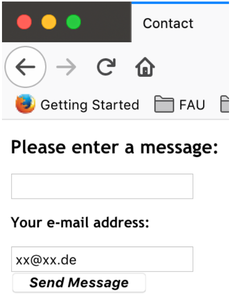


CSS in Practice: The Contact Form Example (Continued)

- Recap: The unstyled contact form – Dream vs. Reality
- 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;}
```

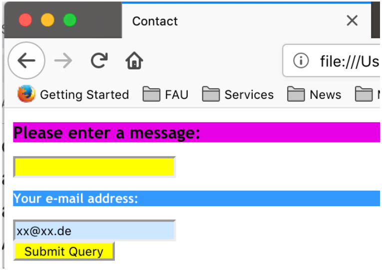


CSS in Practice: The Contact Form Example (Continued)

- Recap: The unstyled contact form – Dream vs. Reality
- Add a CSS file with font information
- 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"/>
```

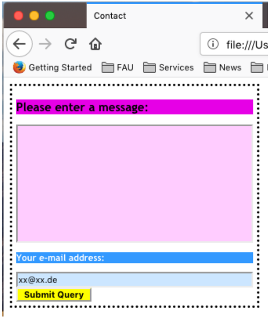
```
h2 {background−color: #e600e6;}
h3 {background−color: #3399ff;
    color: white;}
input{background−color:yellow}
```



CSS in Practice: The Contact Form Example (Continued)

- Recap: The unstyled contact form – Dream vs. Reality
- Add a CSS file with font information
- Add lots of color (ooops, what about the size)
- Add size information and a dotted frame

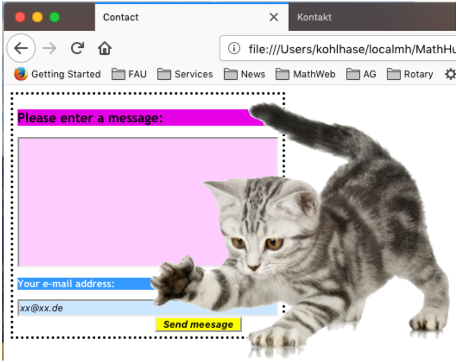
```
<form action="contact-after.html"
        style="width:8cm;border:dotted;padding:5px">
  <h2>Please enter a message:</h2>
  <input name="msg" type="text"
           style="height:4cm;width:8cm;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣background-color:#ffccff"/>
  <br/>
  <h3>Your e-mail address:</h3>
  <input class="important" name="addr"
           type="text"
           value="xx@xx.de" style="width:8cm;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣background-color:#cce6ff"/>
```



CSS in Practice: The Contact Form Example (Continued)

- Recap: The unstyled contact form – Dream vs. Reality
- Add a CSS file with font information
- Add lots of color (ooops, what about the size)
- Add size information and a dotted frame
- 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"/>
```



6.3 JQuery: Write Less, Do More

JQuery: Write Less, Do More

- **Definition 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" />
```

JQuery Philosophy and Layers

- **JQuery Philosophy:** Select an object from the DOM, and operate on it.
- **Syntax Convention:** JQuery instructions start with a $ to distinguish it from JavaScript.
- **Example 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 selectors

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')
```

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 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 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 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.
  - first parameter of toggle() is a duration for the animation.
  - The second 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 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
  - plays it via the play() method.

6.4 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 $\leadsto$ Selection
    - DOM changes easy by clear dependencies.

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 $\leftrightsquigarrow$ 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

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 $\leadsto$ 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 (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

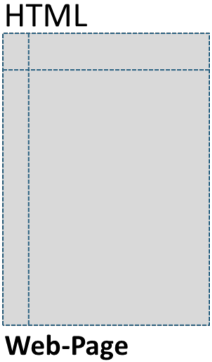
What Tools have we seen so far?

- HTML (Hypertext Markup Language)
  - Text-based markup language for the web
  - tree structure (realized as the DOM in the browser)
    - easy search&find $\leftrightsquigarrow$ 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 (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 $\rightsquigarrow$ dynamic HTML
    - navigating the DOM via JS-selection (relatively clumsy, but sufficient)
    - jQuery navigate the DOM via CSS-selection (reuses successful concepts)

Recap: Web Application Frontend

- **Recap: Web Application Frontend:**

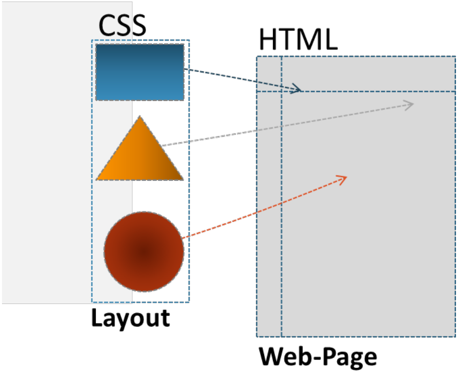
  Web pages are just HTML files.



Recap: Web Application Frontend

- **Recap: Web Application Frontend:**

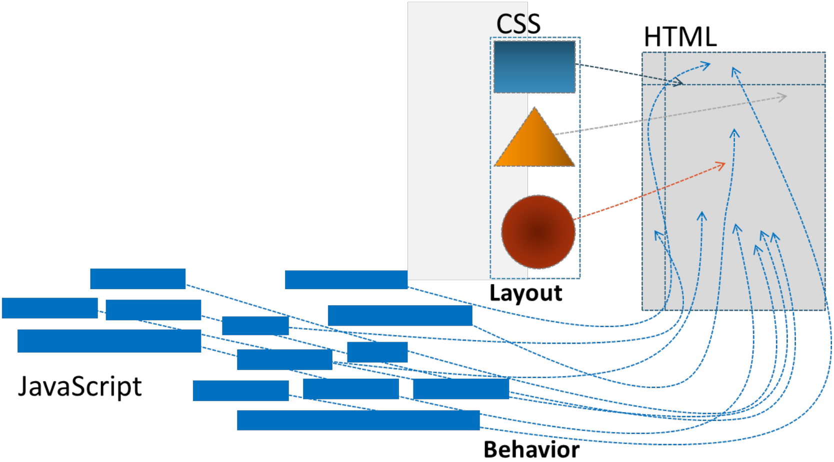
  Layout is specified by CSS instructions and selectors



Recap: Web Application Frontend

- **Recap: Web Application Frontend:**

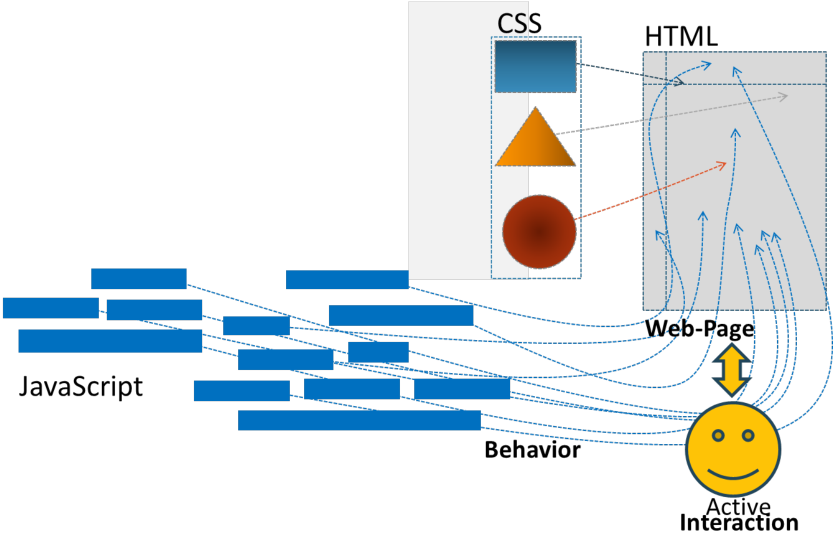
Javascript specifies behavior



Recap: Web Application Frontend

- **Recap: Web Application Frontend:**

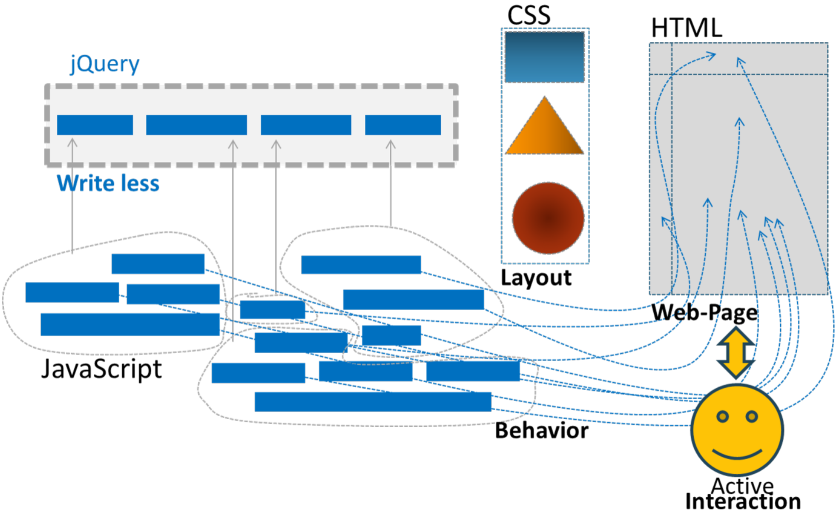
for interacting with the user



Recap: Web Application Frontend

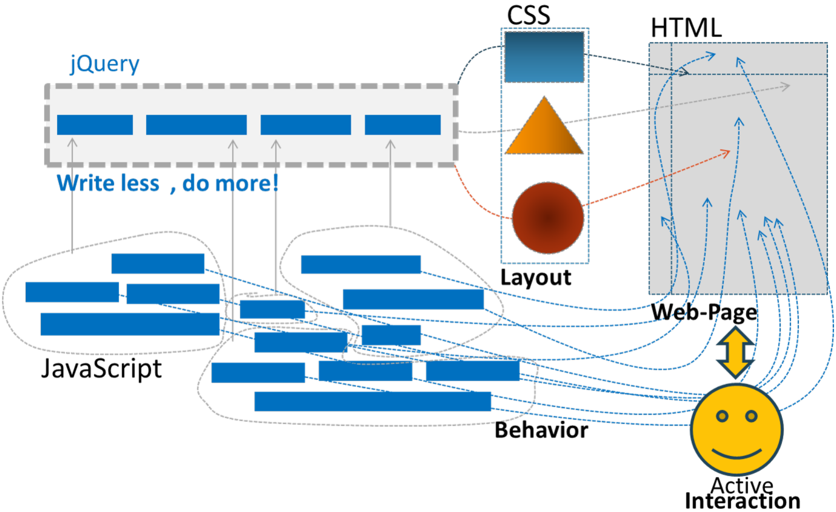
- **Recap: Web Application Frontend:**

JQuery $\widehat{=}$ more succinct Javascript



Recap: Web Application Frontend

- **Recap: Web Application Frontend:**
  JQuery attaches behaviors to DOM elements via CSS selectors



Chapter 7
What did we learn in IWGS-1?

Outline of IWGS 1:

- Programming in Python: (main tool in IWGS)
  - Systematics and culture of programming
  - Program and control structures
  - Basic data strutures like numbers and strings, character encodings, unicode, and regular expressions
- Digital documents and document processing:
  - text files
  - markup systems, HTML, and CSS
  - XML: Documents are trees.
- Web technologies for interactive documents and web applications
  - internet infrastructure: web browsers and servers
  - serverside computing: bottle routing and
  - 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

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

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

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 $\leadsto$ get out more than you put in)
  - semantic web Technologies (standardize ontology formats and inference)
  - Using semantic web Tech for cultural heritage research data $\leadsto$ the WissKI System

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 $\leadsto$ get out more than you put in)
  - semantic web Technologies (standardize ontology formats and inference)
  - Using semantic web Tech for cultural heritage research data $\leadsto$ the WissKI System
- Legal Foundations of Information Systems
  - Copyright & Licensing
  - Data Protection (GDPR)

References I

[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.

[CSS] *CSS Specificity*. url: https://en.wikipedia.org/wiki/Cascading_Style_Sheets#Specificity (visited on 12/03/2018).

[Ecm] *ECMAScript Language Specification*. ECMA Standard. 5th Edition. Dec. 2009.

[Fie+99] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. Internet Engineering Task Force (IETF), 1999. url: http://www.ietf.org/rfc/rfc2616.txt.

References II

[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/`.

[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`.

[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).

References III

[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).

[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.

[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).

References IV

[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.