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

November 9, 2020

Preface

Course Concept

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

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

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

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

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

Course Contents

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

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

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

Building on this, we will cover

1. digital documents and document processing, in particular; text files, markup systems, HTML, and XML; see **?sec.digdocs?**.
2. basic concepts of the World-Wide-Web; see **?sec.www-basics?**
3. Web technologies for interactive documents and applications; in particular Internet infrastructure, web browsers and servers, PHP, dynamic HTML, Javascript, and CSS; see **?sec.webapps?**.

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

1. Data bases; in particular Entity Relationship diagrams, CRUD operations, and DB querying; see **?sec.databases?**.
2. large-scale collaborative development tools: revision control system and issue trackers, in particular Git and GitLab; see **?sec.vci?**
3. Image processing tools, see **?sec.image-processing?**
4. Copyright and Data Privacy as legal foundations of DH tools; see **?sec.legal?**
5. Using the Ontologies and the Semantic Web for Cultural Heritage; see **?sec.ch-ontologies?**
6. The WissKI System: A Virtual Research Environment for Cultural Heritage; see **?sec.wisski?**

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

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

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

Programming Exercises and JuptyterLab as a Web IDE

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

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

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

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

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

- outright programming exercises that introduce the python language or allow to play with the respective concepts in python
- code reading/debugging exercises where the character of Beatrice Beispiel almost solves interesting problems, and
- development steps towards larger applications, which often involve completing python skeletons using the concepts taught in class.

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

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

This Document

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

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

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

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

Acknowledgments

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

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

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

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

WissKI Specialists and Colleagues: The WissKI Chapter has profited from discussions with Peggy Große and Juliane Hamisch, two WissKI specialists at FAU. My colleagues Prof. Peter Bell has

provided the idea and data for the "Kirmes Pictures Project" that grounds some of the second semester.

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

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

Contents

Recorded Syllabus

In this document, we record the progress of the course in the academic year 2020/21 in the form of a "recorded syllabus", i.e. a syllabus that is created after the fact rather than before. For the topics planned for this course, see above.

Recorded Syllabus Winter Semester 2020/21:

#	date	until	slide	page
1	Oct 18.	admin, overview	16	9

Here the syllabus of the last academic year for reference, the current year should be similar; see the course notes of last year available for reference at `http://kwarc.info/teaching/IWGS/notes-2019-20.pdf`.

Recorded Syllabus Winter Semester 2019/20:

#	date	until	slide	page
1	Oct 18.	admin, overview		
2	Oct 24.	python intro		
3	Oct 31.	python fundamentals		
4	Nov. 7.	lists, dictionaries, input/output		
5	Nov. 14.	number/character representation, unicode		
6	Nov. 21.	strings, funcitons, regular expressions		
7	Dec. 28.	plain/formatted text, HTML		
8	Dec. 5.	HTML & XML		
9	Dec. 12.	Documents as trees & XML		
10	Dec. 19.	XML, XPath, URIs		
10	Jan. 9.	WWW Architecture, URIs, HTTP		
11	Jan. 16.	web applications, bottle		
12	Jan. 23.	Cascading Style Sheets		

Recorded Syllabus Summer Semester 2020:

#	date	until	slide	page
1.	April 23.	admin, overview, artefact lifecycles		
2.	April 30.	revision control		
3.	May 7.	distributed revision control, workflows issues		
4.	May 14.	Databases, DDL, sqlite3, python exceptions		
	May 21.	Public Holiday: Christi Himmelfahrt		
5.	May 28.	SQL Queries, Views		
6.	June 4.	The Books App Project		
	June 11.	Public Holiday: Fronleichnam		
7.	June 18.	Image Processing		
8.	June 25.	Image Maps via SVG/CSS		
9.	July 2.	Legal Foundations of IT		
10.	July 9.	Information Privacy, Semantic Networks		
11.	July 16.	Modeling Artefacts in CIDOC CRM		
11.	July 23.	WissKI Architecture		
12.	July 30.	Linked Open Data, What have we learned		
	Aug. 6.	Exam		

Chapter 1

Preliminaries

1.1 Administrativa

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

Prerequisites

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

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

Assessment, Grades

- ▷ Grading Background/Theory: only modules are graded (by the law)
  - ▷ module "DH-Einführung" $\widehat{=}$ courses IWGS1/2, DH-Einführung
  - ▷ DHE module grade $\rightsquigarrow$ 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: $\sim Feb.10.$ (to show you master them)
- ▷ Retake Exam: 60 min exam at the end of the semester ($\sim$ April 10.)

▷ To help you succeed: we offer you

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

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

IWGS Homework Assignments

▷ Homeworks: will be small individual problem/programming/system assignments (but take time to solve) group submission if and only if explicitly permitted

▷ Admin: To keep things running smoothly

- ▷ Homeworks will be posted on StudOn (see `https://studon.fau.de/studon/crs3219641.html`)
- ▷ Homeworks are handed in electronically (plain text, program files, PDF)
- ▷ go to the tutorials, discuss with your TA (they are there for you!)

▷ Homework Discipline:

- ▷ start early! (many assignments need more than one evening's work)
- ▷ Don't start by sitting at a blank screen
- ▷ Humans will be trying to understand the text/code/math when grading it.

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

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

IWGS Tutorials

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

Teaching Assistants: (Doctoral Students in CS)


- ▷ Jonas Betzendahl: jonas.betzendahl@fau.de
- ▷ Philipp Kurth: philipp.kurth@fau.de

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

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

  in IWGS: Lecture + Homework assignments + Tutorials $\widehat{=}$ inverted classroom

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

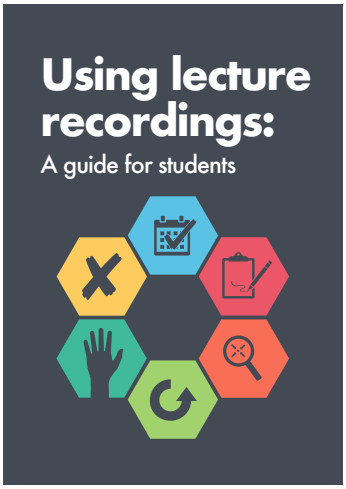
But what if you are not in a lecture or tutorial and want to find out more about the IWGS topics?

Textbook, Handouts and Information, Forums

- ▷ No Textbook: but lots of online 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)
- ▷ Announcements will be posted on the StudOn course forum: https://www.studon.fau.de/studon/goto.php?target=frm_3219641
- ▷ Check the forum frequently for
  - ▷ announcements, homework questions, ...
  - ▷ discussion among your fellow students
- ▷ If you become an active discussion group, the forum turns into a valuable resource!

Practical recommendations on Lecture Videos

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



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

▷ 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 `MS Word`)
- ▷ any operating system you like (I can only help with `UNIX`)
- ▷ Any browser you like (I use `FireFox`: just a better browser (for Math))

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)

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

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

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

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

1.2 Goals, Culture, & Outline of 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.

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

Academic Culture in Computer Science

- ▷ **Definition 1.2.1** The academic culture is the overall style of working, research, and discussion in an academic field.
- ▷ **Observation 1.2.2** *There are significant differences in the academic culture between computer science, 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)

 9

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

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

Outline of IWGS 1:

- ▷ programming in python (main tool in IWGS)
  - ▷ systematics and culture of programming
  - ▷ program and control structures
  - ▷ basic data 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
  - ▷ clientside interaction: dynamic HTML, Javascript, HTML forms
- ▷ Web Application Project (fill in the blanks to obtain a working web app)

 10

1.3 About My Lecturing ...

First let me state the obvious, but there is an important point I want to make.

Do I need to attend the lectures

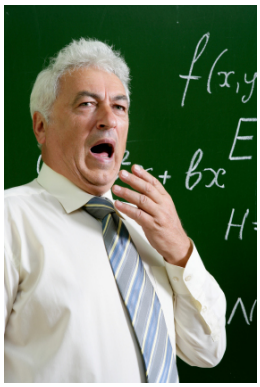
- ▷ Attendance is not mandatory for the IWGS lecture
- ▷ There are two ways of learning IWGS: (both are OK, your mileage may vary)
  - ▷ Approach B: Read a Book
  - ▷ Approach I: come to the lectures, be involved, interrupt me whenever you have a question.

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

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

Traditional Lectures (cum kilo salis)

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



[1] with much more than the proverbial grain of salt.

▷ It is well-known that frontal teaching does not optimize learning

▷ But it scales very well (especially when televised)

So there is a tension between

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

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

▷ We know how to keep large audiences engaged and motivated (even televised)

▷ But the topic is different (IWGS is arguably more complex than Sports/Media)

▷ We're not gonna be able to go all the way to TV entertainment ("IWGS total")

▷ But I am going to (try to) incorporate some elements . . .

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

The very first Questionnaire in IWGS

▷ Question: How many journal articles as "Digital Humanities" up to 2018

a) 7?
b) 1116?
c) 56.000?

▷ Answer:

a) 7 is much much too small (you could not study such a thin field at FAU)

b) 1116 this is the size of the DARIAH bibliography

c) 56.000 is the number of hits labeled "digital humanities" on google scholar(lots of duplicates likely)

▷ Questionnaires: are my attempt to get you to interact

▹ At end of each logical unit (most, if I can get around to preparing them)

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

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

More Generally: My Questions to You

▷ When will I ask them?

▹ In questionnaires.

▹ At various points during the lectures.

▹ We'll do examples together.

▷ Why do I ask them?

▹ They give you the option to follow the lectures *actively*.

▹ They allow me to check whether or not you are able to follow.

▷ How will I look for answers?

▹ "Streber syndrom": 3 students answer all the questions, $N - 3$ sleep.

▹ If this happens, I may resort to picking students randomly.

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

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

Call for Help/Ideas with/for Questionnaires

▷ I have some questionnaires ..., but more would be good!

▷ I made some good ones . . . , but better ones would be better

▷ Please help me with your ideas (I am not Stefan Raab)

- ▷ You know something about IWGS by then.
- ▷ You know when you would like to break the lecture by a questionnaire.
- ▷ There must be a lot of hidden talent! (you are many, I am only one)
- ▷ I would be grateful just for the idea. (I can work out the details)

Part I

IWGS-1: Programming, Documents, Web Applications

Chapter 2

Introduction to Programming

2.1 Programming in IWGS

2.1.1 Introduction to Programming

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

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

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

i) storing and manipulating data about the world,

ii) encoding, generating, and interpreting images, audio, and video,

iii) transporting information for communication,

iv) representing knowledge and reasoning,

v) transforming, optimizing, and verifying other programs,

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

Computer Hardware/Software & Programming

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

▷ **Definition 2.1.2** Hardware consists of

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

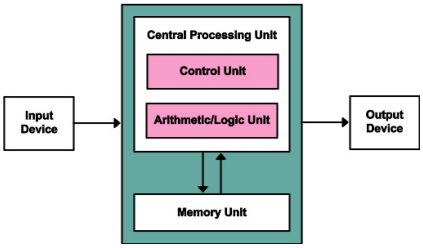
▷ a central processing unit (CPU)

▷ memory: e.g. RAM, ROM, ...

▷ storage devices: e.g. Disks, SSD, tape, ...

▷ input: e.g. keyboard, mouse, touchscreen, ...

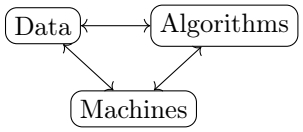
▷ output: e.g. screen, earphone, printer, ...



▷ **Definition 2.1.3** Software consists of

▷ data represents objects and their relationships in the world

▷ programs input, manipulate, output data



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

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

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

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

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

Programming Languages

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

▷ **Remark 2.1.5** The computer does exactly as told

[2] Marvin Minsky; one of the founders of AI

▷ extremely fast extremely reliable

▷ completely stupid: will not do what you mean unless you tell it exactly

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

▷ **Definition 2.1.6** A programming language is the formal language in which we write programs (express an alogrithm concretely)

▷ formal, symbolic, precise meaning (a machine must understand it)

▷ There are lots of programming languages

▷ design huge effort in computer science

▷ all programming languages equally strong

▷ each is more or less appropriate for a specific task depending on the circumstances

▷ Lots of paradigms: imperative, functional programming, logic programming, object oriented programming

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

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

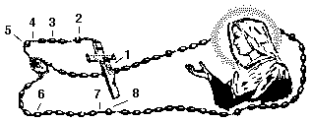
- the behavior of programs is – in principle – predictable
- all errors of program behavior are your own (the programmer's)

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

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

Program Execution

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



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

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

▷

▷ Interpreter: reads a program and executes it directly

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

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

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

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

2.1.2 Programming in IWGS

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

Programming in IWGS: python

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

▷ We cover just enough python, so that you

▷ understand the joy and principle of programming

▷ can play with objects we present in IWGS.

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

▷ For more information on python (homework/preparation)

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

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

▷ ▷ I like [LP; Sth; Swe13];

▷ but also see the language documentation [P3D].

▷ [Kar] is an introduction geared to the (digital) humanities

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

But Seriously... Learning programming in IWGS

▷ The IWGS lecture teaches you

▷ a general introduction to programming and python (next)

▷ various useful concepts and how they can be done in python (in principle)

▷ The IWGS tutorials

▷ teach the actual skill and joy of programming (hacking ≠ security breach)

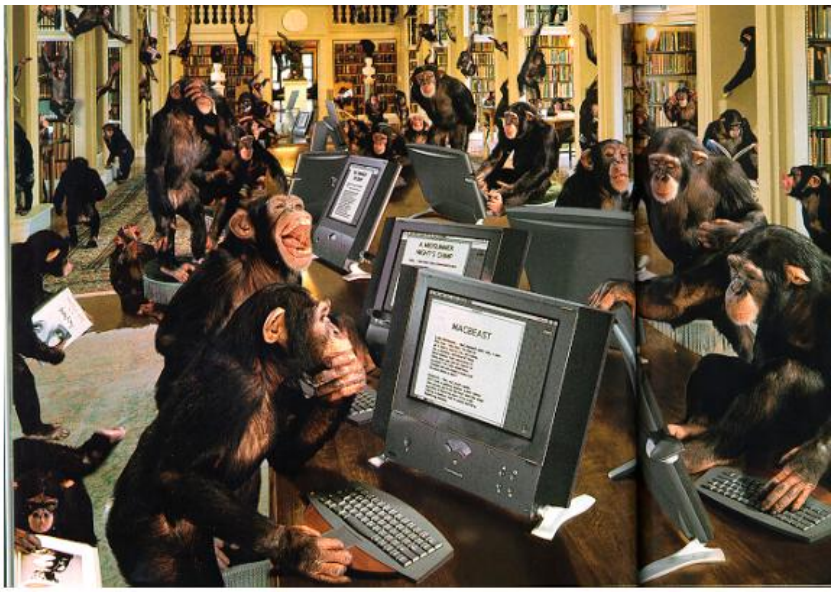
▷ supply you with problems so you can practice that.

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

However, the result would probably be the following:

⚠ 2am in the Kollegienhaus CIP Pool ⚠



 ©: Michael Kohlhase 22 FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

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

⚠ no, let's think ⚠

▷ We have to fully understand the problem, our tools, and the solution space first (That is what the IWGS lecture is for)

▷ read Richard Stallman's quote carefully ⇝ problem understanding is a crucial prerequisite for hacking.

▷ "*The GIGO Principle: Garbage In, Garbage Out*" (– ca. 1967)

▷ "*Applets, Not Craplets*^{tm}" (– ca. 1997)

 ©: Michael Kohlhase 23 FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

2.2 Programming in Python

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

2.2.1 Hello IWGS

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

python in a Nutshell

- ▷ Why python?:
  - ▷ general purpose programming language
  - ▷ imperative, interactive interpreter

  - ▷ syntax very easy to learn (spend more time on problem solving)
  - ▷ scales well:
    - ▷ easy for beginners to write simple programs,
    - ▷ but advanced software can be written with it as well.
- ▷ Interactive mode: The python shell IDLE3
- ▷ For the eager (Optional): Establish a python interpreter (version 3.7) (not 2.?.?, that has different syntax)
  - ▷ install python from `http://python.org` (for offline use)
  - ▷ make sure (tick box) that the python executable is added to the path. (makes shell interaction much easier)

 24 FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

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

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

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

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

```
python ⟨filename⟩
```

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

```
py ⟨filename⟩
```

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

Arithmetic Expressions in python

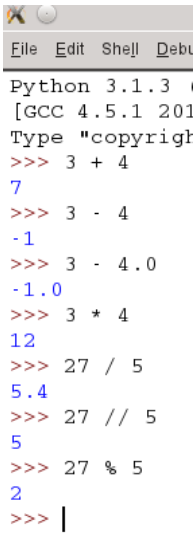
- ▷ Expressions are "programs" that compute values (here: numbers)

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

- Integers (numbers without a decimal point)
  - operators: addition +, subtraction −, multiplication ∗, division /, integer division //, remainder/-modulo %, ...
  - Division yields a float
- Floats (numbers with a decimal point)
  - Operators: integer below floor, integer above ceil, exponential exp, square root sqrt, ...

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

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



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

JupyterLab A Cloud IDE for python

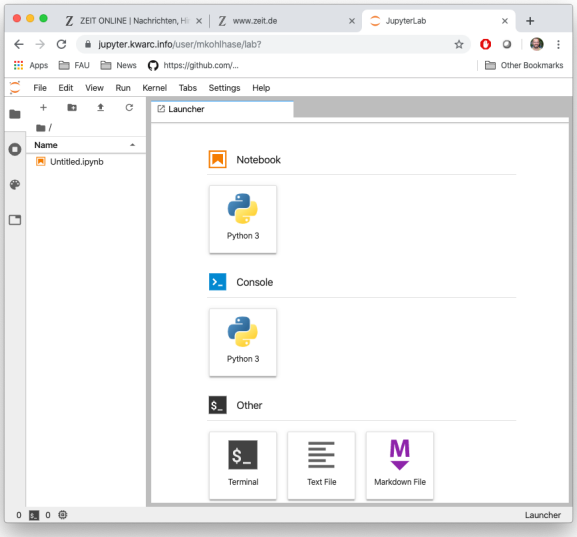
- For helping you it would be good if the TAs could access to your code
- Idea: Use a web IDE (a web-based integrated development environment), which you can use for interacting with the interpreter.
- We will use JupyterLab for IWGS. (but you can also use python locally)
- Homework: Set up JupyterLab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

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

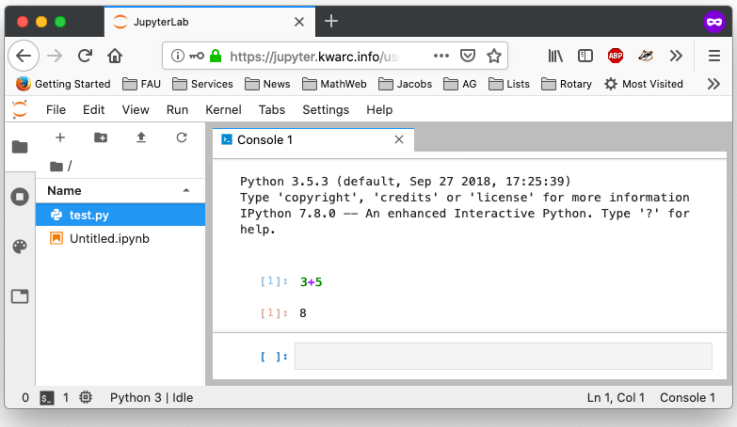
Both IDLE3 as well as JupyterLab come with an integrated editor for writing python programs. These editors gives you python syntax highlighting, and interpreter and debugger integration. In short, IDLE3 and JupyterLab are integrated development environments for python. Let us now go through the interface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

JupyterLab Components

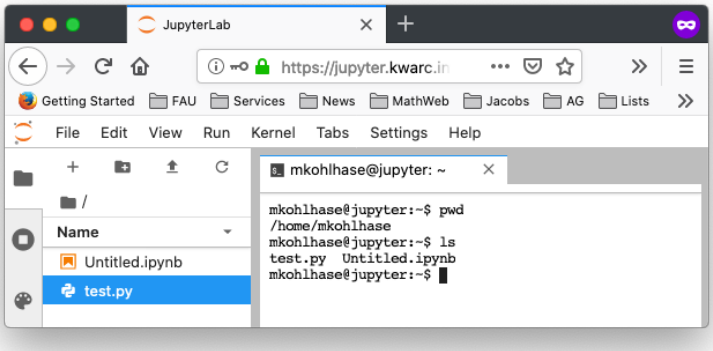
▷ The JupyterLab dashboard gives you access to all components



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



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



▷ **Definition 2.2.1** A shell is a command-line interface for accessing the services of a computer's operating system.

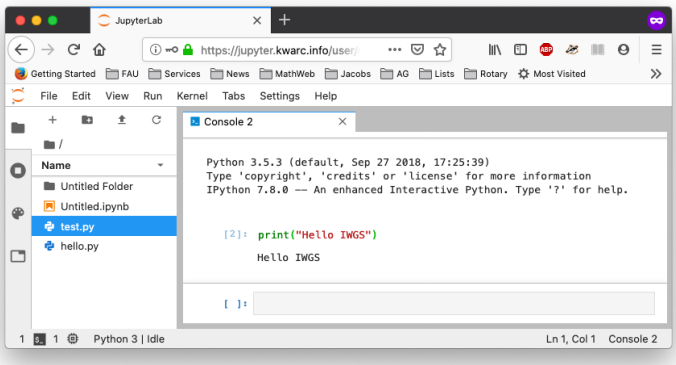
▷ Useful shell commands: See e.g. [All18] for a basic tutorial

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

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

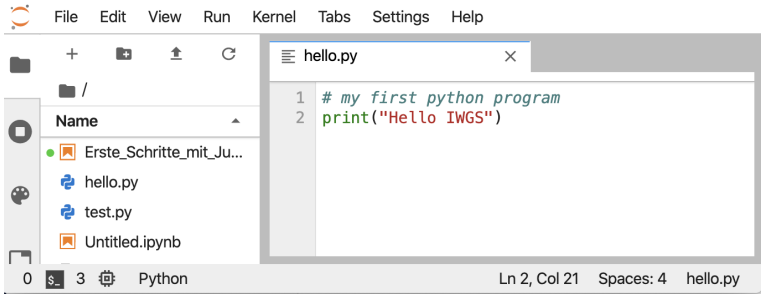
A first program in python

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



▷ Alternatively:

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



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

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

```
import hello
```

 28 FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

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

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

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

   which the terminal interprets as "call the program python3 on me".
3. Finally, we make the file hello executable, i.e. tell the terminal the file should behave like a shell command by issuing

   ```
   chmod u+x hello
   ```

   in the directory where the file hello is stored.

4. We add the line

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

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

With this simple recipe we could in principle extend the repertoire of instructions of the terminal and automate repetitive tasks.

jupyter Notebooks

▷ **Definition 2.2.2** Jupyter notebooks are documents that combine live runnable code with rich, narrative text (for comments and explanations).

▷ **Definition 2.2.3** Jupyter notebooks consist of cells which come in three forms

- ▷ a raw cell shows text as is
- ▷ a markdown cell interprets the contents as markdown text (later more)
- ▷ a code cell interprets the contents as (e.g. python) code

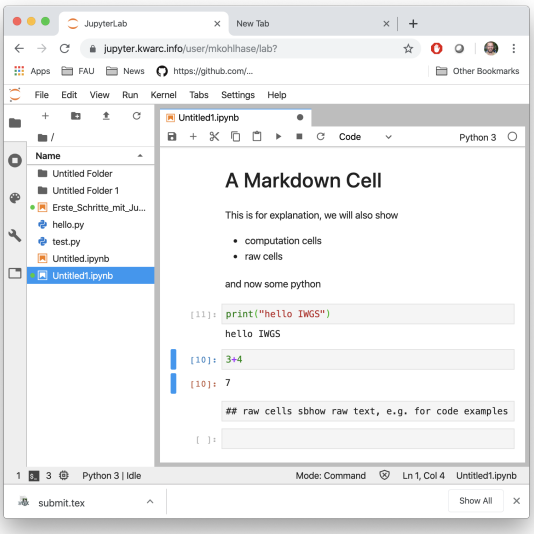
▷ Cells can be executed by pressing "shift-enter" (Just "enter" gives a new line)

▷ Idea: Jupyter notebooks act as a REPL, just as IDLE3, but allow

- ▷ documentation in raw and markdown cells
- ▷ changing and re-executing existing cells.

jupyter Notebooks

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



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

Comments in python

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

2.2.2 Variables and Types

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

Variables in python

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

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

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

▷ A variable assignment ⟪var⟫=⟪val⟫ assigns a value.

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

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


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

Variables in python: Extended Example

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

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

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

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

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


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

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

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

Data Types in python

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

Data type	Keyword	contains	Examples
integers	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
complexess	complex	complex numbers	2+3j,...

- ▷ We will ecounter more types later.

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

Data Types in python (continued)

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

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

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

▷ Data Types in python (continued)

▷ **Observation 2.2.10** *python is strongly typed, i.e. types have to match*

▷ Use data type conversion functions int(), float(), complex()), bool()), and str() to adjust types

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

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


2.2.3 Python Control Structures

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

Conditionals and Loops

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

▷ **Definition 2.2.12** The control flow of a program is the sequence of execution of the program instructions. It is specified via special program instructions called control structures.

▷ **Definition 2.2.13** Conditional execution allows to execute (or not to execute) certain parts of a program (the branches) depending on a condition. We call a code block that enables conditional execution a conditional statement.

▷ **Definition 2.2.14** A loop is a control structure that allows to execute certain parts of a program (the body) multiple times depending on conditions.

▷ **Definition 2.2.15** A condition is a Boolean expression in a control structure.

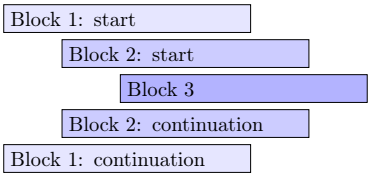
▷ **Example 2.2.16** In python, conditions are constructed by applying a Boolean operator to arguments, e.g. 3>5, x==3, x!=3, ...
or by combining simpler conditions by Boolean connectives or, and, and not (using brakets if necessary), e.g. x>5 or x<3

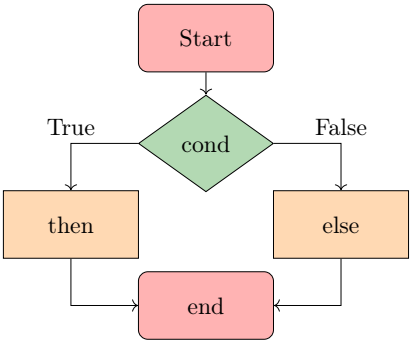
After this general introduction – conditional execution and loops) are supported by all programming language in some form – we will see how this is realized in python

Conditionals in python

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

```
if ⟨condition⟩ :
    ⟨then-part⟩
else :
    ⟨else-part⟩
⟨more code⟩
```





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

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

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

Conditional Execution Example

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

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

Note the indenting of the body parts.

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

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

Variant: Multiple Branches

▷ making multiple branches is similar

```
if ⟨condition⟩ :
    ⟨then-part⟩
```

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

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

▷ **Example 2.2.19 (Better Empathy in python)** In Example 2.2.18 we print Good! even if the input is e.g. I feel terrible, so extend **if**/**else** by

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


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

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

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

Now let us see the syntax for loops in python.

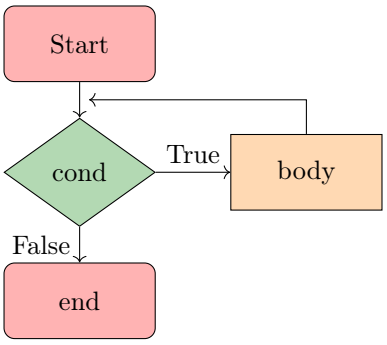
Loops in python

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

- ▷ syntax of the **while** loop

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

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



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

Examples of Loops

▷ **Example 2.2.21 (Counting in python)**

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

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

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

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


Examples of Loops

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

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


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

Direct Implementation

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

More Efficient

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

2.3 Some Thoughts about Computers and Programs

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

Computers as Universal Machines (a taste of theoretical CS)

▷ Observation: Computers are universal tools: their behavior is determined by a program; they can do anything, the program specifies.

▷ Context: Tools in most other disciplines are specific to particular tasks. (except in e.g. ribosomes in cell biology)

▷ **Remark 2.3.1 (Deep Fundamental Result)** There are things no computer can compute.

▷ **Example 2.3.2** whether another program will terminate in finite time.

▷ **Remark 2.3.3 (Church-Turing Hypothesis)** There are two classes of languages

- ▷ Turing complete (or computationally universal) ones that can compute what is theoretically possible.
- ▷ data languages that cannot. (but describe data sets)

▷ **Observation 2.3.4 (Turing Equivalence)** *All programming languages are (made to be) universal, so they can compute exactly the same. (compilers/interpreters exist)*

. . . in particular . . . : Everybody who tells you that one programming languages is the best has no idea what they're talking about (though differences in efficiency, convenience, and beauty exist)

▷ Artificial Intelligence

▷ Another Universal Tool: The human mind. (We can understand/learn anything.)

▷ Strong Artificial Intelligence: claims that the brain is just another computer.

▷ If that is true then

- ▷ the human mind underlies the same restrictions as computational machines
- ▷ we may be able to find the "mind-program".

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

Top Principle of Programming: Compositionality

▷ **Observation 2.3.5** *Modern programming languages compose various primitives and give them a pleasing, concise, and uniform syntax.*

▷ Question: What does all of this even mean?

▷ **Definition 2.3.6** In a programming language, a primitive is a "basic unit of processing", i.e. the simplest element that can be given a procedural meaning (its semantics) of its own.

▷ **Definition 2.3.7 (Compositionality)** All programming languages provide composition principles that allow to compose smaller program fragments into larger ones in such a way, that the semantics of the larger is determined by the semantics of the smaller ones and that of the composition principle employed.

▷ **Observation 2.3.8** *The semantics of a programming language, is determined by the meaning of its primitives and composition principles.*

▷ **Definition 2.3.9** Programming language syntax describes the surface form of the program: the admissible character sequences. It is also a composition of the syntax for the primitives.

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

Consequences of Compositionality

▷ **Observation 2.3.10** *To understand a programming language, we (only) have to understand its primitives, composition principles, and their syntax.*

▷ **Definition 2.3.11** The "art of programming" consists of composing the primitives of a programming language.

▷ **Observation 2.3.12** *We only need very few – about half a dozen – primitives to obtain a Turing complete programming language.*

▷ **Observation 2.3.13** *The space of program behaviors we can achieve by programming is infinitely large nonetheless.*

▷ **Remark 2.3.14** More primitives make programming more convenient.

▷ **Remark 2.3.15** Primitives in one language can be composed in others.

A note on Programming: Little vs. Large Languages

▷ **Observation 2.3.16** *Most such concepts can be studied in isolations, and some can be given a syntax on their own.* *(standardization)*

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

2.4 More about Python

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

2.4.1 Sequences and Iteration

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

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

Lists in python

▷ **Definition 2.4.1** A list is a finite sequence of objects, its elements.

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

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

▷ **Definition 2.4.2** We call [⟪seq⟫] the list constructor.

▷ **Example 2.4.3 (Three lists)** elements can be of different types in python

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

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

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

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

```
>>> list1[−2]
1997
```

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


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

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

Sequences in python

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

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

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

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

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

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

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

Iterating over Sequences in python

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

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

▷ **Example 2.4.9**

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

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

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


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

python Dictionaries

▷ **Definition 2.4.11** A dictionary is an unordered, indexed collection of ordered pairs (k, v), where we call k the key and v the value.

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

▷ **Example 2.4.12** 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.

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

Interacting with Dictionaries

▷ Dictionary commands by example

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

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

keys

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

values

```
for x in thisdict:
    print(thisdict[x])
```

items

```
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.

2.4.2 Input and Output

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

Input/Output in python

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

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

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

- ▷ **Definition 2.4.14** A file is a resource for recording data in a storage device.
- ▷ **Definition 2.4.15** Files are identified by a file name are managed by a file system which organize them hierarchically into named folders and locate them by a path; a sequence of folder names. The file name and the path together fully identify a file.

  A file name usually consists of a base name and an extension separated by a dot character.
- ▷ Some file systems restrict the characters allowed in the file name and/or lengths of the base name or extension.
- ▷ **Definition 2.4.16** Once a file has been opened, the CPU can write to it and read from it. After use a file should be closed to protect it from accidental reads and writes.

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

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

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

Disk Input/Output in python

▷ In python we have special instructions for dealing with files:

- ▷ open(⟪path⟫,⟪iospec⟫) returns a file object f; ⟪iospec⟫ is one of r (read only; the default), a (append $\widehat{=}$ write to the end), and r+ (read/write).
- ▷ f.read() reads the file f into a string.
- ▷ f.readline() reads a single line from the file (including the newline character (\n) otherwise returns the empty string ''.
- ▷ f.write(⟪str⟫) appends the string ⟪str⟫ to the end of f, returns the number of characters written.
- ▷ f.close() closes f to protect it from accidental reads and writes.

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

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


Disk Input/Output in python (continued)

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

```
>>> 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 modules (additional functionality)
- ▷ useful for loading python-encoded data (e.g. dictionaries)

2.4.3 Functions and Libraries in Python

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

Functions in python (Introduction)

▷ Observation: sometimes programming tasks are repetitive

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

▷ Idea: We can automate the repetitive part by functions

▷ **Example 2.4.19**

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

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

Functions in python (Example)

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

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

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

We can now make the intuitions above formal and give the exact python syntax of functions.

Functions in python (Definition)

▷ **Definition 2.4.21** A python function is defined by a code snippet of the form

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

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

▷ the indented part is called the body of f, (⚠: whitespace matters in python)

▷ the p_i are called parameters, and n the arity of f.

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

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

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

Functions vs. Methods in python

▷ There is another mechanism that is similar to functions in python. (we briefly introduce it here to delineate)

▷ Background: Actually, the types from Definition 2.2.9 are classes, ...

▷ **Definition 2.4.22** In python all values belong to a class, which provide special functions we call methods. Values are also called objects, to emphasise class aspects. Method application is written with dot notation: ⟨obj⟩.⟨meth⟩(⟨args⟩) corresponds to ⟨meth⟩(⟨obj⟩,⟨args⟩).

▷ **Example 2.4.23** Finding the position of a substring

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


Functions vs. Methods in python

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

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

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

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

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

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

python Libraries

▷ Idea: Functions, classes, and methods are re-usable, so why not package them up for others to use.

▷ **Definition 2.4.25** A python library is a python file with a collection of functions, classes, and methods. It can be loaded via the **import** command.

▷ There are ≥ 150.000 libraries for python ($\widehat{=}$ packages on `http://pypi.org`)

- ▷ search for them at `http://pypi.org` (e.g. 815 packages for "music")
- ▷ install them with pip install ⟪package-name⟫
- ▷ look at how they were done (all have links to source code)
- ▷ maybe even contribute back (report issues, improve code, ...) (open source)

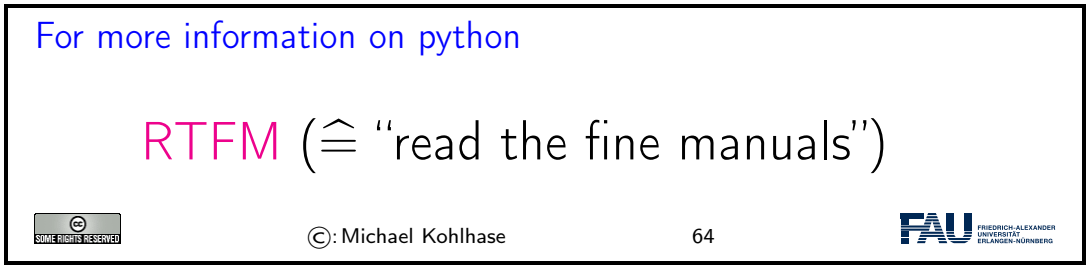
The python community is an open source community, therefore many developers organize their code into libraries and license them under open source licenses.

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

2.4.4 A Final word on Programming in IWGS

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

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



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

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

2.5 Exercises

Problem 1 (Hello World)
Write an extended "Hello World Program" in a file called exthello.py. The program should print information about you and your account. Specifically, the information should be:

```
Hello World!  I am <your name>.
This is my first exercise in IWGS.
```

Problem 2 (Variable Assignment and Output)
Write a program in python that calculates the total number of seconds in a leap year, stores the result in a variable and then displays that to the user.

Problem 3 (Variable Reuse)
Programming often has efficiency as one of its goals. After all, why go through the trouble of telling a computer how to do something, if you could do it better and quicker yourself?

Write a program in python that prints the string "supercalifragilisticexpialidocious" five times, but *without* typing the word five times yourself.

Problem 4 (Human Readable Time)
In programming, it is often the case that your program collects a lot of data from various sources. It then becomes essential to present this data in a way that the user (usually a human!) can easily understand. For example, most humans don't know how long a longer timespan is if it is given only in seconds.

Write a program in python that first initialises a variable seconds = 1234567. Then, the program should calculate and print how long this timespan is in days, hours, minutes and seconds instead of just seconds.

Problem 5 (String Presentation)
Keeping with the importance of well-presented information: You can use certain special symbols in strings to give them a better formatting when they are ultimately printed. For example, when you put "\n" into a string, instead of printing these symbols, the output switches to a *new line*.

Write a python program that prints your favourite haiku (a poem with five syllables on the first line, seven on the second and five on the third) on three three lines, but using only *one* **print** statement.

P.S.: If you don't have a favourite haiku and can't think of one yourself, you can use this one:

My cow gives less milk,
now that it has been eaten,
by a fierce dragon.

Problem 6 (User Input)
One of the most important things to learn about a programming language is how to get input from the user in front of the screen. In python, one way of doing this is the input instruction.

For example: if you write answer = input("Do you like sharks?"), this will print the message you gave ("Do you like sharks?"), wait for the user to submit a response and store it as a string in the variable answer when you run the program. You can then use it like any other value stored in a variable.

Write a simple program that prints a generic greeting message, then asks the user to input their name, stores the input in a variable and then finishes with a goodbye message that uses the name the user gave.

Problem 7 (Simple Branching)
The next important concept is control flow. A program that always does the same thing gets boring fast. We want to write programs that do different things under different circumstances. In python, one way to do this are conditional statements.

Write a python program that asks the user if they have a pet. If their answer was "yes", the program should ask what kind of pet they have. Since sloths are the cutest animals (at least for this exercise), the program should print "awww!" if the user's second answer was "sloth" and "cool!" if it was something else. If the user does not answer with "yes" the first time around, the program should quit with a goodbye message.

Problem 8 (Simple Looping)
Computers are very good at doing the same thing over and over again without complaining or messing up. Humans are not. In python, we can use a loops if we want something done multiple times.

Suppose your boss wants the string "Programming **is** cool!" printed exactly 1337 times (for some reason ...). Typing up the string yourself takes about nine seconds each time, printing it in a loop takes no time.

To save time, write a python program that prints the sentence "Programming **is** cool!" 1337 times using a loop. Your program should also keep track of (store in a variable) how much time the loop saved the programmer in total (9 seconds per iteration of the loop). Print this value after the loop finishes.

Problem 9 (Temperature Conversion)
Write two python programs, named celsius2fahrenheit and fahrenheit2celsius, that given a number as input from the user convert it to the respective other temperature scale and print the result.

The conversion formulas are as follows:

$$[^\circ C] = ([^\circ F] - 32) \cdot \frac{5}{9} \qquad [^\circ F] = [^\circ C] \cdot \frac{9}{5} + 32$$

Remember that input will save the input as text, not as a number. You can convert a string to a number using the function float.

Example: float("3.1415") will evaluate to the *number* 3.1415. If the text given to float does not actually represent a number (e.g. float("bad")), python will throw an error.

Afterwards, please test your programs against another converter (easily found via your internet search engine of choice) to make sure that your functions produce the correct results.

Bibliography

[All18] Jay Allen. *New User Tutorial: Basic Shell Commands*. 2018. URL: https://www.liquidweb.com/kb/new-user-tutorial-basic-shell-commands/ (visited on 10/22/2018).

[HWC] *The Hello World Collection*. URL: http://helloworldcollection.de/ (visited on 11/23/2018).

[JKI] Jonas Betzendahl. *juptyter.kwarc.info Documentation*. URL: https://kwarc.info/teaching/IWGS/jupyter-documentation.pdf (visited on 08/29/2020).

[Kar] Folgert Karsdorp. *Python Programming for the Humanities*. URL: http://www.karsdorp.io/python-course/ (visited on 10/14/2018).

[Koh08] Michael Kohlhase. "Using LaTeX as a Semantic Markup Format". In: *Mathematics in Computer Science* 2.2 (2008), pp. 279–304. URL: https://kwarc.info/kohlhase/papers/mcs08-stex.pdf.

[Koh20] Michael Kohlhase. *sTeX: Semantic Markup in TeX/LaTeX*. Tech. rep. Comprehensive TeX Archive Network (CTAN), 2020. URL: http://www.ctan.org/get/macros/latex/contrib/stex/sty/stex.pdf.

[LP] *Learn Python – Free Interactive Python Tutorial*. URL: https://www.learnpython.org/ (visited on 10/24/2018).

[Nor+18a] Emily Nordmann et al. *Lecture capture: Practical recommendations for students and lecturers*. 2018. URL: https://osf.io/huydx/download.

[Nor+18b] Emily Nordmann et al. *Vorlesungsaufzeichnungen nutzen: Eine Anleitung für Studierende*. 2018. URL: https://osf.io/e6r7a/download.

[P3D] *Python 3 Documentation*. URL: https://docs.python.org/3/ (visited on 09/02/2014).

[PyRegex] Rodolfo Carvalho. *PyRegex - Your Python Regular Expression's Best Buddy*. URL: http://www.pyregex.com/ (visited on 12/03/2018).

[Pyt] *re – Regular expression operations*. online manual at https://docs.python.org/2/library/re.html. URL: https://docs.python.org/2/library/re.html.

[Sth] *A Beginner's Python Tutorial*. http://www.sthurlow.com/python/. seen 2014-09-02. URL: http://www.sthurlow.com/python/.

[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.

Index