General Information & Communication Technology
350101 GenICT I & II 2015
Partial Lecture Notes

Michael Kohlhase

School of Engineering & Science
Jacobs University, Bremen Germany
m.kohlhase@jacobs-university.de

September 20, 2015

Preface

This Document

This document contains the course notes for the those parts of the course General Information & Communication Technology I & II held at Jacobs University Bremen in the academic year 2014.

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

Caveat: This document is made available for the students of this course only. It is still a draft and will develop over the course of the current course and in coming academic years.

Course Concept

Aims: The course 350101 "General Information & Communication Technology I/II" (GenICT) is a two-semester course that introduces concepts of Computer Science Concepts to non-CS students. The course is co-taught by four Jacobs Computer Science Faculty each covering a quarter of the materials.

Course Contents

Goal: We want to demonstrate both theoretical foundations of CS as Computer Science, and we want to provide practical knowledge helping students to cope with understanding and handling Computers, electronic documents and data, and the Web. Roughly the first half of the first semester is devoted to theoretical foundations and core concepts (Kohlhase and Jaeger), and the second half of the semester to the practical real-world stuff (Schnwlder and Baumann). Throughout the semester, students will be introduced stepwise to one of the main programming languages of today, Python.

Acknowledgments

Materials: The presentation of the programming language python uses materials prepared by Dr. Heinrich Stamerjohanns and Dr. Florian Rabe for the ESM Phython modules.

GenICT Students: The following students have submitted corrections and suggestions to this and earlier versions of the notes: Kim Philipp Jablonski, Tom Wiesing.

Contents

Chapter 1

Outline of the Course

This course gives a broad (and in a few places, also a bit more in-depth) introduction to Computer Science for non-CS students. We want to demonstrate both theoretical foundations of CS as Computer Science, and we want to provide practical knowledge helping students to cope with understanding and handling Computers, electronic documents and data, and the Web. Roughly the first half of the semester is devoted to theoretical foundations and core concepts, and the second half of the semester to the practical real-world stuff. Throughout the semester, students will be introduced stepwise to one of the main programming languages of today, Python.

Outline of the Course I: Theory

- ▷ Introduction to Information Processing (Michael Kohlhase)
  - ▷ Introduction to Programming in general and python as the course language
  - ▷ Encoding and computing numbers on the computer
  - ▷ Encoding and translating characters and documents (also known as "texts"...)
  - ▷ Describing the structure of programs and documents (regular expressions, XML)
  - ▷ Legal foundations of Intellectual Property (Copyright, etc.)
- ▷ From Symbols to Computing (Herbert Jaeger)
  - ▷ Boolean logic (what "bits" are and how to combine them to to "digital computation")
  - ▷ Complexity: some computer programs take very long to load or complete, others are fast – must this be so?

Outline of the Course I: Applications

- ▷ Computer Architecture and Distributed Systems (Jürgen Schönwälder)
  - ▷ Computer components (processor, core, memory, parallel and serial busses)

- ▷ Interpreter / compiler / emulator / virtual machines
- ▷ Program execution (processes), system level view on processes
- ▷ Resource management (memory, cpu) and interprocess communication
- ▷ Introduction to the Internet (IP addresses, domain names, autonomous systems, content caches, . . .)
- ▷ Network (in)security (TLS, VPNs, PGP, wifi security, . . .)

▷ Databases and Web Applications (Peter Baumann)

- ▷ Databases: what is a database; tables; the database query language SQL
- ▷ Web services: WWW, http, HTML, Web requests
- ▷ Security: Web, mail, databases

Chapter 2

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 about Computer Science concepts as efficient and painless as possible.

Grading and Exams

- ▷ The course is taught in four sections.
- ▷ Every section ends with an exam (∼45 minutes) covering the content of the ending section.
- ▷ Each exam is worth 20% of the final grade. (80% together)
- ▷ The homeworks in each section will count 5% of the final grade. (20% together)

Homework Submissions, Grading, Tutorials

- ▷ Submissions: We use Heinrich Stamerjohanns' `JGrader` system
  - ▹ submit all homework assignments electronically to `https://jgrader.de`.
  - ▹ you can login with your Jacobs account and password. (should have one!)
  - ▹ feedback/grades to your submissions
  - ▹ get an overview over how you are doing! (do not leave to midterm)
- ▷ Tutorials: select a tutorial group and actually go to it regularly
  - ▹ to discuss the course topics after class (lectures need pre/postparation)
  - ▹ to discuss your homework after submission (to see what was the problem)
  - ▹ to find a study group (probably the most determining factor of success)

Even though the lecture itself will be the main source of information in the course, there are various resources from which to study the material.

Textbooks, Handouts and Information, Forum

- ▷ No required textbook, but course notes, posted slides
- ▷ Course notes in PDF will be posted at `http://minds.jacobs-university.de/teaching/GenICTFall2015`
- ▷ Everything will be posted on PantaRhei (Notes+assignments+course forum)
  - ▷ announcements, contact information, course schedule and calendar
  - ▷ discussion among your fellow students(careful, I will occasionally check for academic integrity!)
  - ▷ `http://panta.kwarc.info` (use your Jacobs login)

Set Up PantaRhei Access: to get notifications

- ▷ 1) Log into PantaRhei,
  2) find the course `GenICT Fall 2015`,
  3) request membership (I will approve you)
- ▷ if there are problems send e-mail me (`mkohlhase@jacobs-university.de`)

No Textbook: There is no single textbook that covers the course. Instead we have a comprehensive set of course notes (this document). They are provided in two forms: as a large PDF that is posted at the course web page and on the PantaRhei system. The latter is actually the preferred method of interaction with the course materials, since it allows to discuss the material in place, to play with notations, to give feedback, etc. The PDF file is for printing and as a fallback, if the PantaRhei system, which is still under development, develops problems.

But of course, there is a wealth of literature on the subject, and the references at the end of the lecture notes can serve as a starting point for further reading. We will try to point out the relevant literature throughout the notes.

Part I

GenICT 1, Module 1: Programming & Documents

We start off the course by giving a very brief introduction to programming. We use python as the main programming language. As many students already have experience through the python lab, we will only briefly recap – and introduce to those students who have no experience yet – the python language

Chapter 3

Introducction to Programming

To understand programming we need to understand a bit about computers – the devices programs run on first

3.1 Introduction to Programming

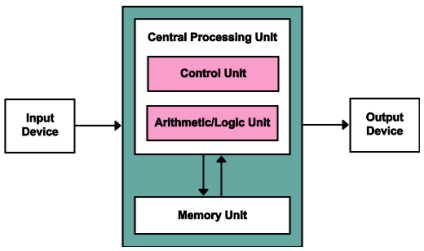
Programming is an important and distinctive part of "Information and Communication Technology" – the topic of this course. Before we delve into learning Python, we will review some of the basics of computing to situate the discussion.

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".

Computer Hardware/Software & Programming

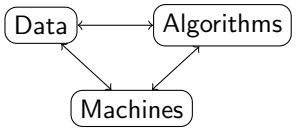
▷ **Definition 3.1.1** computer hardware consists of devices that execute commands/instructions:

- ▷ central processing unit (CPU)
- ▷ memory: e.g. RAM, Disks, ...
- ▷ input: e.g. keyboard, touchscreen, ...
- ▷ output: e.g. screen, earphone, ...



▷ software = programs and data

- ▷ programs input, manipulate, output data
- ▷ hardware stores data and runs programs



▷ Programming = writing programs (Telling the computer what to do)

▷ The computer does exactly as told

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

- ▷ 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)

hardware-software-programming

Programming Languages

▷ The language in which we write the program

- ▷ formal, symbolic, precise meaning

▷ There are lots of programming languages

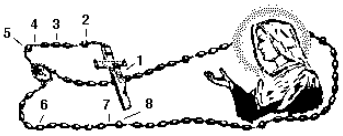
- ▷ design huge effort in computer science
- ▷ all programming languages equally strong
- ▷ each is more or less appropriate for a specific task depending on the circumstances

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

▷ Everybody who tells you that one PL is the best has no idea what they're talking about

Program Execution

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



▷ Program: formal version of the algorithm (needed for computers)

▷ Interpreter: reads a program and executes it directly

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

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

▷ **Observation 3.1.2** *Compilers are more efficient, but more cumbersome for development*

3.2 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 GenICT.

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

python in a Nutshell

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

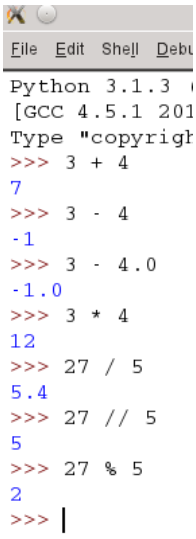
  - syntax very easy to learn (spend more time on problem solving)
  - scales well
    - easy for beginners to write simple programs
    - but advanced software can be written with it as well
- Interactive mode: The python shell IDLE3
- Homework: install python (version 3.4.?) from `http://python.org` (not 2.?.?, that has different syntax)

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

The download will install the python interpreter and the python shell IDLE3 that can be used for interacting with the interpreter directly. The latter also comes with an integrated editor for writing python programs. This editor gives you python syntax highlighting, and interpreter and debugger integration. In short, IDLE3 is an integrated development environment for python.

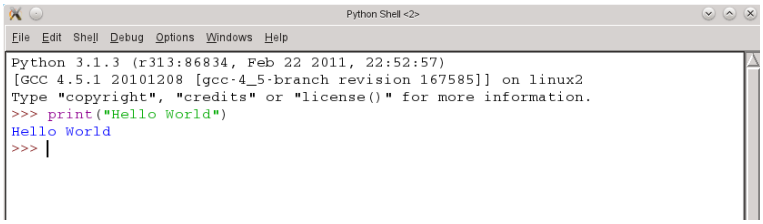
Arithmetic Expressions in python

- ▷ Integers (numbers without a decimal point)
- ▷ Floats (numbers with a decimal point)
- ▷ Division yields a float
- ▷ Special // operator for integer division
- ▷ Special % (modulo) operator (remainder)



A first program in python

▷ A classic "Hello World" program:
start IDLE3, type `print("Hello World")` (print a string)



▷ Alternatively: treat it as a program in a file

- ▷ Open an editor of your choice (any editor; but use "text mode")
- ▷ Type your program,

```
# My first python program
print("Hello world")
```

- ▷ Save as `hello.py`, run the python interpreter on it from a shell

```
python3 hello.py
```


Comments in python

▷ It is highly advisable to insert comments into your programs

▷ Single line comments start with a #

▷ Multiline comments start and end with three quotes (single or double: """ or ''')

▷ Comments are ignored by python but are useful information for the programmer

Variables in python

▷ Idea: values (of expressions) can be given a name for later reference

▷ **Definition 3.2.1** A variable is a storage location which contains a value and an associated identifier – the variable name.

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

▷ in python:

- ▷ = declares variable name and assigns value.
- ▷ variable names start with letter or _, cannot be keywords (case-sensitive)

```
>>> 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
>>>
```

▷ **Example 3.2.2 (Swapping Variables)**

```
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)
```


Data Types in python

▷ programs process data, which can be combined by operators

▷ Data types group data into types

- ▷ 1, 2, 3, etc. are data of type "integer"

▷ `hello` is data of type "string"

▷ Data types determine which operators can be applied

▷ In python, every value has a type, variables can have any type

▷ **Definition 3.2.3** python has the following three basic data types

Data type	Name	Example
Integers	`int`	1, -5, 0
Floats	`float`	1.2, .125, -1.0
Strings	`str`	"Hello", 'Hello', "123", 'a'

▷ The type of a variable is automatically determined when first used

```
firstVariable = 23 # integer
weight = 3.45 # float
first = 'Hello' # str
```

▷ **Observation 3.2.4** *python is strongly typed, i.e. types have to match*

▷ Use datatye conversion functions `int()`, `float()`, and `str()` to adjust types

▷ **Example 3.2.5**
```
>>> 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'
```


Functions in python

▷ Observation: sometimes programming tasks are repetitive

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

▷ Idea: We can automate the repetitive part by functions

▷ **Definition 3.2.6** 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}
    """
    {this is the code in the function}
    {more code, it can contain p_1,...,p_n, and f}
    return {value (e.g. text or number) of the function call}
{some code outside the function}
```

▷ the indented part is called the body of f, (⚠2: 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.

▷ **Example 3.2.7**

```
def greet (who):
    print("Hello",who,"how are you today? How about some GenICT?")
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)

Branching and Looping in python

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

▷ Change the control flow of a program with branching and statements.

▷ **Definition 3.2.8** Branching allows to execute (or not to execute) certain parts of a program depending on conditions.

▷ **Definition 3.2.9** Looping allows to execute certain parts of a program multiple times depending on conditions.

▷ **Definition 3.2.10** A condition (or Boolean expression) is a python expression that can be evaluated to `True` o `False`.

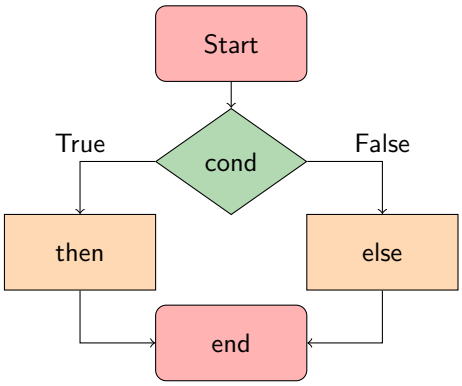
▷ **Example 3.2.11** Conditions are constructed by applying a Boolean operator to arguments, e.g. `3>5`, `x==3`, `x!=3`, ...
or by combining simpler conditions by Boolean connectives, e.g. `x>5 or x<3`

Branching in python

▷ **Definition 3.2.12** Branching via if/then/else statements

▷ Binary branching:

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



▷ multiple branching:

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

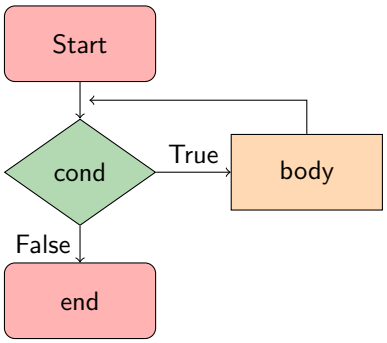
▷ conditional parts indented!

Looping in python

▷ **Definition 3.2.13** Looping via while-blocks

syntax of the while loop

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



▷ breaking out of loops with `break`

▷ body must be indented!

Lists in python

▷ **Definition 3.2.14** A list is a finite enumeration of objects

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

▷ **Example 3.2.15 (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 3.2.16** List elements can be assessed by specifying ranges

```
>>> list1[0]
```

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

▷ **Example 3.2.17** lists can be constructed by python functions

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

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

Iterating over Lists in python

▷ **Definition 3.2.18** A for loop iterates a program fragment over a list (or list-like object)

python uses the following general syntax

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

▷ **Example 3.2.19**

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

▷ Lists and strings can also act as ranges

Example 3.2.20

```
def mirror (x) :
    for i in x :
        print(i)
    for i in reversed(list(x)) :
        print(i)
```


Now that we have acquired some basic programming skills, we will drill into how the basic data structures – numbers and strings – are actually represented in the computer. This will give us some insights on how to deal with them in practice.

3.3 Basic Data Structures

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

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

3.3.1 Numbers

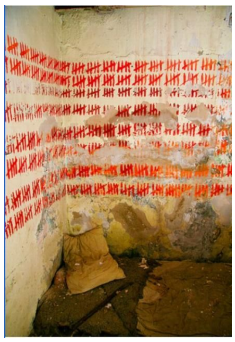
We start with the representation of numbers. There are multiple number systems, as we are interested in the principles only, we restrict ourselves to the natural numbers – all other number systems can be built on top of these. But even there we have choices about represesentation, which influence the space we need and how we compute with natural 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 3.3.1** We call the representation of natural numbers by slashes on a surface the unary natural numbers

▷ Question: How do we represent them on a computer? (not bones or walls)

▷ Idea: If we have a memory bank of n binary digits, initialize all by 0, represent each slash by a 1 from the right.

▷ **Example 3.3.2** Memory bank with 32 binary digits, represening 11.

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

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

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

▷ Positional Number Systems

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

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

▷ **Definition 3.3.3** A positional number system $\mathcal{N}$ is a pair $\mathcal{N} = \langle D_b, \varphi_b \rangle$ with

- ▷ D_b is a finite alphabet of b digits. b is called the base or radix of $\mathcal{N}$
- ▷ assign each digit $d \in D_b$ a number $\varphi_b(d)$ between 0 and $b-1$.
- ▷ Extend φ_b to sequences of digits by $\varphi_b(\langle n_k, \ldots, n_1 \rangle) := \sum_{i=1}^{k} \varphi_b(n_i) \cdot b^{i-1}$

▷ **Example 3.3.4** $\langle \{a, b, c\}, \varphi \rangle$ with with $\varphi(a) := 0$, $\varphi(b) := 1$, and $\varphi(c) := 2$ is a positional number system for base three. We have

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

▷ **Observation 3.3.5** *To convert a number n to base b, use successive integer division (division with remainder) by b:*

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

▷ **Example 3.3.6 (Convert 456 to base 8)** Result: 710_8

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

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

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

systems. However, here, we also want to ensure that the meaning mapping φ_b is a bijection, since we want to define the arithmetics on the encodings by reference to The arithmetical operators on the unary natural numbers.

Commonly Used Positional Number Systems

▷ **Example 3.3.7** The following positional number systems are in common use.

name	set	base	digits	example
unary	$\mathbb{N}_1$	1	/	$/////_1$
binary	$\mathbb{N}_2$	2	0,1	0101000111_2
octal	$\mathbb{N}_8$	8	0,1,...,7	63027_8
decimal	$\mathbb{N}_{10}$	10	0,1,...,9	162098_{10} or 162098
hexadecimal	$\mathbb{N}_{16}$	16	0,1,...,9,A,...,F	$FF3A12_{16}$

▷ **Notation 3.3.8** attach the base of $\mathcal{N}$ to every number from $\mathcal{N}$. (default: decimal)

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

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

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

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

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

Arithmetics in Positional Number Systems

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

▷ Tom Lehrer's "New Math"

▷ **Example 3.3.9**

Addition base 4

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

binary multiplication

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

3.3.2 Characters & Strings

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

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

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

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

The ASCII Character Code

▷ **Definition 3.3.10** The American Standard Code for Information Interchange (ASCII) code assigns characters to numbers 0-127

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

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

⋙ Motivated by punchcards: The character 0 (binary 0000000) carries no information NUL, (used as dividers)
Character 127 (binary 1111111) can be used for deleting (overwriting) last value (cannot delete holes)

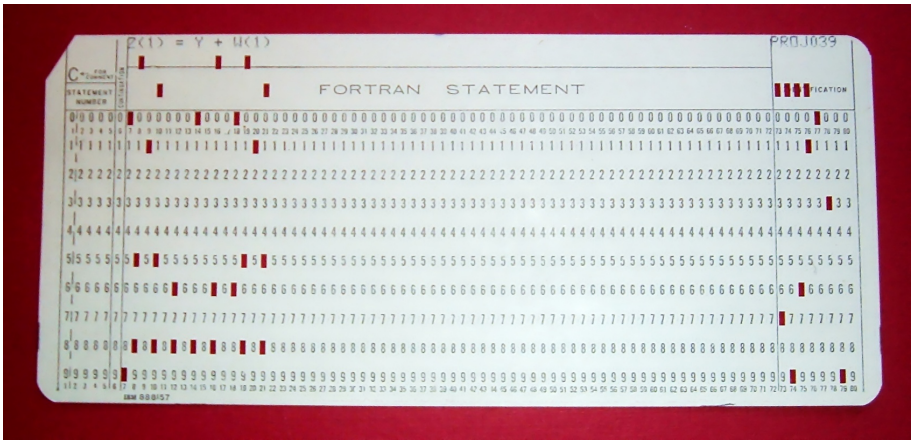
▷ The ASCII code was standardized in 1963 and is still prevalent in computers today (but seen as US-centric)

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

A Punchcard

▷ A punch card is a piece of stiff paper that contains digital information represented by the presence or absence of holes in predefined positions.

▷ **Example 3.3.11** This punch card encoded the `FORTRAN` statement `Z(1) = Y + W(1)`



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

Playing with Strings and Characters in python

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

▷ `ord` gives the `ASCII` number of the character, `chr` `ASCII` character for a number.

▷ **Example 3.3.12 (Playing with Characters)**

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

▷ strings can be accessed by ranges $[i{:}j]$ ($[i] \hat{=} [i{:}i]$)

▷ **Example 3.3.13** taking strings apart and re-assembling them.

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


⚠ Note: Example 3.3.12 and Example 3.3.13 (or any other examples in this lecture) is not

production code, but didactially motivated – to show you what you can do with the objects we are presenting in python.

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

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

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

Problems with ASCII encoding

- ▷ Problem: Many of the control characters are obsolete by now (e.g. NUL,BEL, or DEL)
- ▷ Problem: Many European characters are not represented (e.g. è,ñ,ü,ß,...)
- ▷ European ASCII Variants: Exchange less-used characters for national ones
- ▷ **Example 3.3.14 (German ASCII)** remap e.g. [↦ Ä,] ↦ Ü in German ASCII ("Apple][" comes out as "Apple ÜÄ")
- ▷ **Definition 3.3.15 (ISO-Latin (ISO/IEC 8859))** 16 Extensions of ASCII to 8-bit (256 characters) ISO-Latin 1 ≙ "Western European", ISO-Latin 6 ≙ "Arabic",ISO-Latin 7 ≙ "Greek"...
- ▷ Problem: No cursive Arabic, Asian, African, Old Icelandic Runes, Math,...
- ▷ Idea: Do something totally different to include all the world's scripts: For a scalable architecture, separate
  - ▷ what characters are available from the (character set)
  - ▷ bit string-to-character mapping (character encoding)

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

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

Unicode and the Universal Character Set

- ▷ **Definition 3.3.16 (Twin Standards)** A scalable Architecture for representing all the worlds scripts
  - ▷ The Universal Character Set defined by the ISO/IEC 10646 International

Standard, is a standard set of characters upon which many character encodings are based.

- ▷ The Unicode Standard defines a set of standard character encodings, rules for normalization, decomposition, collation, rendering and bidirectional display order

▷ **Definition 3.3.17** Each UCS character is identified by an unambiguous name and an integer number called its code point.

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

▷ **Definition 3.3.18** Most (non-Chinese) characters have code points in $[1, 65536]$ (the basic multilingual plane).

▷ **Notation 3.3.19** For code points in the Basic Multilingual Plane (BMP), four digits are used, e.g. U+0058 for the character LATIN CAPITAL LETTER X;

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

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

Character Encodings in Unicode

▷ **Definition 3.3.20** A character encoding is a mapping from bit strings to UCS code points.

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

▷ **Definition 3.3.21 (Unicode Transformation Format)**

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

▷ **Definition 3.3.22** The UTF-8 encoding follows the following encoding scheme

Unicode	Byte1	Byte2	Byte3	Byte4
U+000000 – U+00007F	0xxxxxxx			
U+000080 – U+0007FF	110xxxxx	10xxxxxx		
U+000800 – U+00FFF	1110xxxx	10xxxxxx	10xxxxxx	
U+010000 – U+10FFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

▷ **Example 3.3.23** $ = U+0024 is encoded as 00100100 (1 byte)
¢ = U+00A2 is encoded as 11000010,10100010 (two bytes)
€ = U+20AC is encoded as 11100010,10000010,10101100 (three bytes)

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

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

Programming with UniCode strings is particularly simple, strings in python are UTF-8-encoded UniCode strings and all operations on them are UniCode-based[2] This makes the introduction to UniCode in python very short, we only have to know how to produce non-`ASCII` characters – which are on regular keyboards.

Unicode in python

▷ the python `str` data type is UniCode encoded as UTF-8.

▷ How to write UniCode characters?: there are four ways

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

```
>>> "\N{GREEK CAPITAL LETTER DELTA}" # Using the character name
'\u0394'
>>> "\u0394" # Using a 16-bit hex value
'\u0394'
>>> "\U00000394" # Using a 32-bit hex value
'\u0394'
```

[2]Older programming languages have `ASCII` strings only, and UniCode strings are supplied by external modules.

Chapter 4

Computing with Documents

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

There are several dialects of regular expression languages that differ in details, but share the general setup and syntax. Here we introduce the `UNIX` variant.

Regular Expressions, see [RE]

▷ **Definition 4.0.1** A regular expression (also called regexp) is a formal expression that specifies a set of strings.

▷ **Definition 4.0.2 (Meta-Characters for Regexps)**

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

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

Let us now fortify our intuition with some (simple) examples.

Regular Expression Examples

▷ **Example 4.0.3 (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.

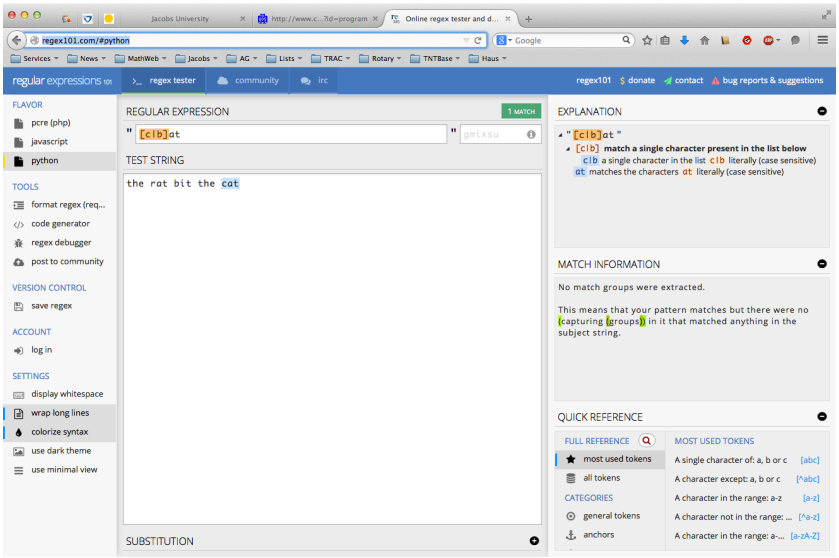
▷ **Example 4.0.4 (A more complex example)** The following regexp times in a variety of formats, such as `10:22am`, `21:10`, `08h55`, and `7.15 pm`.

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


As we have seen regular expressions can become quite cryptic and long (cf. e.g. Example 4.0.4), so we need help in developing them. One way is to use one of the many regexp testers online

Playing with Regular Expressions

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



The `sed` stream editor is an example of a standalone utility – it is shipped with most operating systems – that uses regular expressions. It can be used to automate repetitive editing operations on files.

The sed Stream Editor

▷ **Definition 4.0.5** The `sed` utility is a stream editor, it takes a stream (think file) and some regexp replacement commands as an input and gives a stream as a output.

▷ **Example 4.0.6** A `sed` command is of the form

- ▷ s/⟪regexp⟫/⟪replacement⟫/ (replace once), or
- ▷ s/⟪regexp⟫/⟪replacement⟫/g (replace globally).

▷ To invoke `sed` in a shell (e.g. on `linux`, `MacOSX`, or `cygwin` on `Windows`)

```
sed -e 's/oldstuff/newstuff/g' inputFileName > outputFileName
```

or (if sedfile.**sed** contains many `sed` commands)

```
sed -f sedfile.sed inputFileName > outputFileName
```

▷ **Example 4.0.7 (Update the Jacobs Web Site)**

```
sed -e 's/International Univ/Jacobs Univ/g;s/IUB/Jacobs/g' index.html > index.html
```

▷ **Example 4.0.8 (Stalin eliminates Trotzki)** Let cleanse.**sed** be the `sed` file

```
s/Leon Trotzki//g;s/Trotzki//g
s/Lev Davidovich Bronstein//g;s/Davidovich//g;s/Bronstein//g
```

then Stalin can just use the following shell script to cleanse Kreml documents

```
find / -name -E ".*\.html|.*\.txt" -exec 'sed -f cleanse.sed {} > {} \;
```


Example 4.0.8 shows the power of `sed` in combination with other utilities. Here we use the `UNIX find` utility that searches a file system for files with certain characteristics – here file names that match the regexp `.*\.html`.*txt— and executes the `sed` script `cleanse` we defined earlier.

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(p,s)`: Return a list of non-overlapping matches of `p` in —s—.

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

```
['cat','rat']
```

▷ `re.sub(p,r,s)`: Replace substrings that match `p` in `s` by `r`

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

▷ `re.split(p,s)`: Split `s` into substrings that match `p`.

```
>>> re.split(r'\W+','When shall we three meet again?'))
['When','shall','we','three','meet','again']
```


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

Example: Correcting and Anonymizing Documents

▷ **Example 4.0.9** We write an a program that makes simple corrections on documents and also crosses out all names.

▷ *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 program:

▷ we first initialize and load modules

```
#!/usr/bin/env python3
import re
import sys
```

▷ then we decode the argument and put it into a variable

```
s = sys.argv[1]
```

▷ We put put a space after a comma, (use `r` string prefix for "raw strings")

```
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)
```

▷ 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("({l})({c}+)({l})"
    .format(l=l, c=c),
    lambda (m):"{0}{1}{2}".format(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(r"({c}{l}+ ({c}{l}*(\.?) )?{c}{l}+)"
    .format(l=l, c=c),
    lambda (m):re.sub("\S", "X", m.group(1)),
    s)
```

▷ finally, we print the result

```
print(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?

Chapter 5

Structured and Web Documents

5.1 Multimedia Documents on the World Wide Web

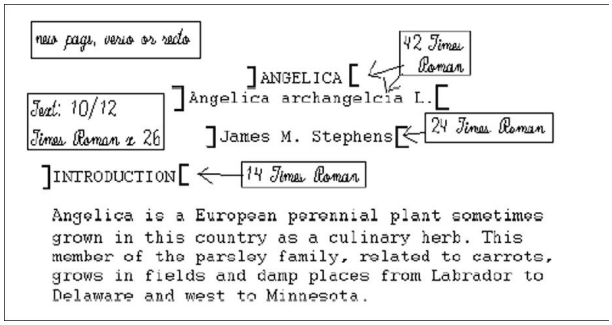
We have seen the client-server infrastructure of the WWWeb, which essentially specifies how hypertext documents are retrieved. Now we look into the documents themselves.

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

Document Markup

▷ **Definition 5.1.1 (Document Markup)** Document markupmarkup is the process of adding codes (special, standardized character sequences) to a document to control the structure, formatting, or the relationship among its parts.

▷ **Example 5.1.2** A text with markup codes (for printing)



There are many systems for document markup ranging from informal ones as in Definition 5.1.1 that specify the intended document appearance to humans – in this case the printer – to technical ones which can be understood by machines but serving the same purpose.

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

HTML: Hypertext Markup Language

▷ **Definition 5.1.3** The HyperText Markup Language (HTML), is a representation format for web pages. Current version 4.01 is defined in [RHJ98].

▷ **Definition 5.1.4 (Main markup elements of HTML)** HTML marks up the structure and appearance of text with tags of the form `<el>` (begin) and `</el>` (end), 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`
images	`img`	tables	`table`, `th`, `tr`, `td`, ...
styling	`style`, `div`, `span`	old style	`b`, `u`, `tt`, `i`, ...
interaction	`script`	forms	`form`, `input`, `button`

▷ **Example 5.1.5** A (very simple) HTML file with a single paragraph.

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


HTML was created in 1990 and standardized in version 4 in 1997. Since then there has HTML has been basically stable, even though the WWWeb has evolved considerably from a web of static web pages to a Web in which highly dynamic web pages become user interfaces for web-based applications and even mobile applets. Acknowledging the growing discrepancy, the W3C has started the standardization of version 5 of HTML.

HTML5: The Next Generation HTML

▷ **Definition 5.1.6** The HyperText Markup Language (HTML5), is believed to be the next generation of HTML. It is defined by the W3C and the WhatWG.

▷ HTML5 includes support for

- ▷ audio/video without plugins,
- ▷ a canvas element for scriptable, 2D, bitmapped graphics
- ▷ SVG for Scalable Vector Graphics
- ▷ MathML inline and display-style mathematical formulae

▷ The W3C is expected to issue a "recommendation" that standardizes HTML5 in 2014.

▷ Even though HTML5 is not formally standardized yet, almost all major web browsers already implement almost all of HTML5.

As the WWWeb evolved from a hypertext system purely aimed at human readers to an Web of multimedia documents, where machines perform added-value services like searching or aggregating,

it became more important that machines could understand critical aspects web pages. One way to facilitate this is to separate markup that specifies the content and functionality from markup that specifies human-oriented layout and presentation (together called "styling"). This is what "cascading style sheets" set out to do. Another motivation for CSS is that we often want the styling of a web page to be customizable (e.g. for vision-impaired readers).

CSS: Cascading Style Sheets

▷ Idea: Separate structure/function from appearance.

Definition 5.1.7 The Cascading Style Sheets (CSS), is a style sheet language that allows authors and users to attach style (e.g., fonts and spacing) to structured documents. Current version 2.1 is defined in [BCHL09].

▷ **Example 5.1.8** Our text file from Example 5.1.5 with embedded CSS

```
<html>
  <head>a
    <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 GenCSII!.</p>
  </body>
</html>
```

5.2 Web Applications

In this section we show how with a few additions to the basic WWWeb infrastructure introduced in ?www-basics?, we can turn web pages into web-based applications that can be used without having to install additional software.

The first thing we need is a means to send information back to the web server, which can be used as input for the web application. Fortunately, this is already forseen by the HTML format.

HTML Forms: Submitting Information to the Web Server

▷ **Example 5.2.1** Forms contain input fields and explanations.

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

The result is a form with three elements: a text, an input field, and a submit button, that will trigger a HTTP GET request to the URL specified in the `action` attribute.

As the WWWeb is based on a client-server architecture, computation in web applications can be executed either on the client (the web browser) or the server (the web server). For both we have a special technology; we start with computation on the web server.

Server-Side Scripting: Programming Web Pages

▷ Idea: Why write HTML pages if we can also program them! (easy to do)

▷ **Definition 5.2.2** A server-side scripting framework is a web server extension that generates web pages upon HTTP GET requests.

▷ **Example 5.2.3** `perl` is a scripting language with good string manipulation facilities. `perl` CGI is an early server-side scripting framework based on this.

▷ Server-side scripting frameworks allow to make use of external resources (e.g. databases or data feeds) and computational services during web page generation.

▷ Problem: Most web page content is static (page head, text blocks, etc.) (and no HTML editing support in program editors)

▷ Idea: Embed program snippets into HTML pages. (only execute these, copy rest)

▷ **Definition 5.2.4** A server-side scripting language is a server side scripting framework where web pages are generated from HTML documents with embedded program fragments that are executed in context during web page generation.

▷ Note: No program code is left in the resulting web page after generation (important security concern)

To get a concrete intuition on the possibilities of server-side scripting frameworks, we will present PHP, a commonly used open source scripting framework. There are many other examples, but they mainly differ on syntax and advanced features.

PHP, a Server-Side Scripting Language

▷ **Definition 5.2.5** PHP (originally "Programmable Home Page Tools", later "PHP: Hypertext Processor") is a server-side scripting language with a `C`-like syntax. PHP code is embedded into HTML via special "tags" `<?php` and `?>`

▷ **Example 5.2.6** The following PHP program uses `echo` for string output

```
<html>
  <body><?php echo 'Hello world';?></body>
</html>
```

▷ **Example 5.2.7** We can access the server clock in PHP (and manipulate it)

```
<?php
```

```
$tomorrow = mktime(0,0,0,date("m"),date("d")+1,date("Y"));
echo "Tomorrow is ".date("d. m. Y", $tomorrow);
?>
This fragment inserts tomorrow's date into a web page
```

▷ **Example 5.2.8** We can generate pages from a database (here MySQL)

```
<?php
$con = mysql_connect("localhost","peter","abc123");
if (!$con)
  {
  die('Could not connect: ' . mysql_error());
  }

mysql_select_db("my_db", $con);

$result = mysql_query("SELECT * FROM Persons");

while($row = mysql_fetch_array($result))
  {
  echo $row['FirstName'] . " " . $row['LastName'];
  echo "<br />";
  }

mysql_close($con);
?>
```

▷ **Example 5.2.9** We can even send e-mail via this e-mail form.

```
<html><body>
<?php
if (isset($_REQUEST['email']))//if "email" is filled out, send email
  {//send email
  $email = $_REQUEST['email'] ;
  $subject = $_REQUEST['subject'] ;
  $message = $_REQUEST['message'] ;
  mail("someone@example.com", $subject,
  $message, "From:" . $email);
  echo "Thank you for using our mail form";}
else //if "email" is not filled out, display the form
  {echo "<form method='post' action='mailform.php'>
   Email: <input name='email' type='text' /><br />
   Subject: <input name='subject' type='text' /><br />
   Message:<br />
   <textarea name='message' rows='15' cols='40'>
   </textarea><br />
   <input type='submit' />
   </form>";}
?>
</body></html>
```


What would we do in python

▷ **Example 5.2.10 (HTML Hello World in python)**

```
print("</html>")
print("<body>Hello world</body>")
print("</html>")
```

▷ Why is this not as good as PHP?

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

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

With server-side scripting frameworks like PHP, we can already build web applications, which we now define.

Web Applications: Using Applications without Installing

▷ **Definition 5.2.11** A web application is a website that serves as a user interface for a server-based application using a web browser as the client.

▷ **Example 5.2.12** Commonly used web applications include

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

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

⋙ **Definition 5.2.13** A web application framework is a software framework for creating web applications.

▷ **Example 5.2.14** The LAMP stack is a web application framework based on `linux`, `apache`, `MySQL`, and PHP.

▷ **Example 5.2.15** A variant of the LAMP stack is available for `Windows` as XAMPP [XAM].

Indeed, the first web applications were essentially built in this way. Note however, that as we remarked above, no PHP code remains in the generated web pages, which thus "look like" static web pages to the client, even though they were generated dynamically on the server.

There is one problem however with web applications that is difficult to solve with the technologies so far. We want web applications to give the user a consistent user experience even though they are made up of multiple web pages. In a regular application we we only want to login once and expect the application to remember e.g. our username and password over the course of the various interactions with the system. For web applications this poses a technical problem which we now discuss.

State in Web Applications and Cookies

▷ Recall: Web applications contain multiple pages, HTTP is a stateless protocol.

▷ Problem: how do we pass state between pages? (e.g. username, password)

▷ Simple Solution: Pass information along in query part of page URLs.

▷ **Example 5.2.16 (HTTP GET for Single Login)** Since we are generating pages we can generated augmented links

```
<a href="http://example.org/more.html?user=joe,pass=hideme">... more</a>
```

Problem: only works for limited amounts of information and for a single session

⋙ Other Solution: Store state persistently on the client hard disk

▷ **Definition 5.2.17** 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 5.2.18** third party cookies are used by advertising companies to track users across multiple sites. (but you can turn off, and even delete cookies)

Note that that both solutions to the state problem are not ideal, for usernames and passwords the URL-based solution is particularly problematic, since HTTP transmits URLs in GET requests without encryption, and in our example passwords would be visible to anybody with a packet sniffer. Here cookies are little better as cookies, since they can be requested by any website you visit.

We now turn to client-side computation

One of the main advantages of moving documents from their traditional ink-on-paper form into an electronic form is that we can interact with them more directly. But there are many more interactions than just browsing hyperlinks we can think of: adding margin notes, looking up definitions or translations of particular words, or copy-and-pasting mathematical formulae into a computer algebra system. All of them (and many more) can be made, if we make documents programmable. For that we need three ingredients: *i)* a machine-accessible representation of the document structure, and *ii)* a program interpreter in the web browser, and *iii)* a way to send programs to the browser together with the documents. We will sketch the WWWeb solution to this in the following.

Dynamic HTML

▷ Observation: The nested, markup codes turn HTML documents into trees.

▷ **Definition 5.2.19** The document object model (DOM) is a data structure for the HTML document tree together with a standardized set of access methods.

▷ Note: All browsers implement the DOM and parse HTML documents into it; only then is the DOM rendered for the user.

▷ Idea: generate parts of the web page dynamically by manipulating the DOM.

▷ **Definition 5.2.20** 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 [ECM09].

▷ **Example 5.2.21** 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>
```

Let us fortify our intuition about dynamic HTML by going into a more involved example.

Applications and useful tricks in Dynamic HTML

▷ **Example 5.2.22** hide document parts by setting CSS `style` attribs to `display:none`

```
<html>
  <head>
    <style type="text/css">#dropper { display: none; }</style>
    <script language="JavaScript" type="text/javascript">
      window.onload = function toggleDiv(element){
          if(document.getElementById(element).style.display == 'none')
              {document.getElementById(element).style.display = 'block'}
          else if(document.getElementById(element).style.display == 'block')
              {document.getElementById(element).style.display = 'none'}}
     </script>
  </head>
  <body>
    <button onclick="toggleDiv('dropper')">...more </button>
    <div id="dropper"><p>Now you see it!</p></div>
  </body>
</html>
```

Application: write "gmail" or "google docs" as JavaScript enhanced web applications. (client-side computation for immediate reaction)

⋙ Current Megatrend: Computation in the "cloud", browsers (or "apps") as user interfaces

Current web applications include simple office software (word processors, online spreadsheets, and presentation tools), but can also include more advanced applications such as project management, computer-aided design, video editing and point-of-sale. These are only possible if we carefully balance the effects of server-side and client-side computation. The former is needed for computational resources and data persistence (data can be stored on the server) and the latter to keep personal information near the user and react to local context (e.g. screen size).

Bibliography

[BCHL09] Bert Bos, Tantek Celik, Ian Hickson, and Høakon Wium Lie. Cascading style sheets level 2 revision 1 (CSS 2.1) specification. W3C Candidate Recommendation, World Wide Web Consortium (W3C), 2009.

[ECM09] ECMAScript language specification, December 2009. 5th Edition.

[RE] `re` - regular expression operations. online manual at `https://docs.python.org/2/library/re.html`.

[RHJ98] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.0 Specification. W3C Recommendation REC-html40, World Wide Web Consortium (W3C), April 1998.

[XAM] apache friends - xampp. `http://www.apachefriends.org/en/xampp.html`.