Python-Programming Course Programming in Bioinformatics Lecture notes on a course held in the winter 2019/2020

Stefan Kurtz

Research Group for Genome Informatics
Center for Bioinformatics Hamburg
University of Hamburg

October 15, 2019

Some parts of the material presented here were adapted from the book: *Beginning Perl for Bioinformatics* by James Tisdall, O'Reilly Media, 2001.

We thank Tim Kurmann for suggesting several improvements of the presentation.

Contents (1/4)

Contents (2/4)

Contents (3/4)

Contents (4/4)

	int	float	RE	str	list	dict	graph	own func	own class	generator	numpy	matplotl.
transcription			×	×								
reverse complement			×	×								
motif finding			×	×	×							
complex numbers	×		×									
histograms			×		×	×						
data matrices				×		×		×				
fractions									×			
genetic code			×	×	×	×		×	×			
restriction maps			×	×		×		×	×			
fibonacci numbers								×		×		
molecules								×		×		
XML parsing				×		×		×				
numerics		×						×			×	×
mandelbrot sets		×						×			×	×
dijstra's alg							×					

Algorithms and Programming (1/5)

Algorithm

- describes the solution to a problem in terms of
  - the data needed to represent the problem instance and
  - the set of steps necessary to produce the intended result
- developing an algorithm is first important step before programming
- without an algorithm there can be no program

Programming ...

- is the process of taking an algorithm and encoding it into the notation of a programming language
- allows computers to execute an algorithm
- is an important part of what computer scientists (and many other people dealing with data) do

This section is from `http://interactivepython.org/runestone/static/pythonds/index.html`

Algorithms and Programming (2/5)

A program is ...

- the result of programming
- (often) a textual representation of the algorithm solving a particular problem

A programming language

- must provide a notation allowing to represent both the algorithmic steps and the data relevant in an algorithm
- provides data types and control constructs

Algorithms and Programming (3/5)

Control constructs

- allow algorithmic steps to be represented in a convenient yet unambiguous way by
  - sequential processing
  - selection for decision-making
  - iteration for repetitive control
  - recursion
- abstract from the low level control instructions (like move, store, jump) executed in the Central Processing Unit of a computer

Algorithms and Programming (4/5)

Data types

- the computer represents data as strings of binary digits (bits/bytes)
- to give these strings meaning, we need to have data types
- data types provide an interpretation for this binary data and methods to access and modify this binary data
- they allow to think about the data in terms that make sense with respect to the problem under consideration
- e.g. in some context the binary data may be interpreted as characters which are parts of a text, that we need to analyze syntactically, or in another context, it may be interpreted as integers to be added
- low-level, built-in data types (sometimes called the primitive data types) provide the building blocks for programming
- we will later see how to build our own data types, but first consider integers as an example for a primitive data type

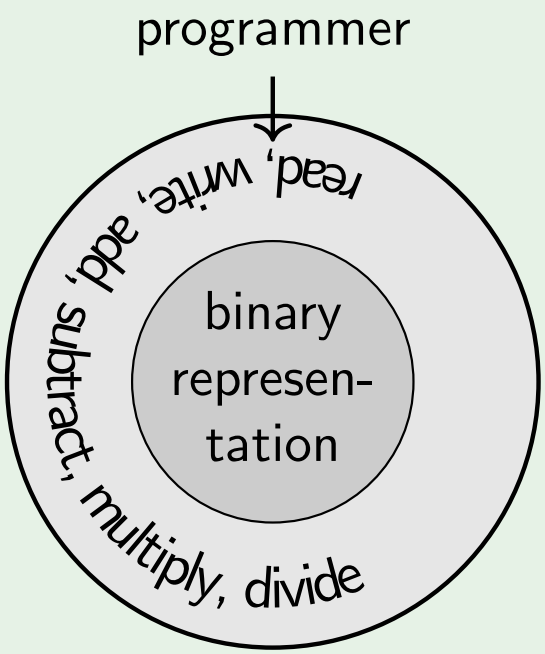
Algorithms and Programming (5/5)

Example (Data type for integers)

- available in all relevant programming languages
- interprets strings of binary digits in the computer's memory as decimal integers, like 10011 as 19:

$$19 = \mathbf{1} \cdot 2^4 + \mathbf{0} \cdot 2^3 + \mathbf{0} \cdot 2^2 + \mathbf{1} \cdot 2^1 + \mathbf{1} \cdot 2^0$$

- provides operations to read, write, add, subtract, multiply, divide, ... integers (based on these binary strings)
- we (usually) do not have to understand how these representations and their operations internally work
- the data type encapsulates them for us



Synopsis

- to turn an algorithm into a running programming, we need a programming language which
  - provides control structures
  - provides means to use existing data types and build our own
  - abstracts from the internal binary representation of data and control in the CPU

Here we choose Python3.

Some features of Python

- Python is a programming language used in all kinds of areas
  - web and internet development
  - numeric and scientific computing, machine learning
  - simulation
  - development of graphical user interfaces
  - data visualization
  - data integration
- many organizations are using Python these days to perform major computing tasks
- see e.g. the success stories at `https://www.python.org/about/success/`
- in all fields of sciences, Python is increasingly used as one of the main programming languages
  - Computational Biology
  - Drug Discovery
  - Computational Chemistry
  - Physics
  - Geosciences
  - Nanoscience

- Python is for free and runs on all commonly used operating systems, such as Linux, MS-Windows, Mac OS X, ...
- if you do not have Python on your computer, then install it
- see `https://www.python.org/downloads/`
- two meanings of Python: programming language and translator
- translator turns Python program (also called script or code) into instructions understood by the computer
- two ways of using the translator:

Python interpreter

reads Python program from file and executes it

```
$ cat helloworld.py
print('hello world')
$ python3 helloworld.py
hello world
```

Python shell

is interactively used and directly executes Python commands as you type them

```
$ python3
Python 3.4.6 (default, 2017-22-03)
Type "help" for more information.
>>> print('hello world')
hello world
```

Low and long learning curve

- get started quickly
- many useful programs can be written without much experience
- learning all of Python will take a while (and not be achieved within this course)
- Python is object oriented
  - everything you manipulate is an object and the results of those manipulations are objects as well
  - each object is generated as an instance of a class
  - a class consists of a state (with variable bindings) and a set of functions (called methods) to manipulate the state
  - a class can be seen as a construction plan according to which an object of this class, called instance, is created
- more about object oriented programming later

Python in the Sciences

- Python contains features simplifying several common tasks in science, for example
  - parse information from text files or XML files containing experimental data (parsing means to analyze the structure and extract relevant information)
  - manipulate medium size DNA or protein sequences
  - generate website content for a scientific project
  - generate program code in any language, e.g. C
  - manipulate matrices in a flexible and efficient way (Numpy)
  - plot data using a simple and powerful interface (Matplotlib)
  - store/retrieve information in/from relational databases (Python DB-API)
  - perform statistical evaluations (Scipy, Pandas, Seaborn)
  - glue different programs (in any language) into one large program
    - via system calls and output parsing
    - by accessing functions and datatypes of external language (possible for C-programs)

Rapid Prototyping

- Python is excellent language for rapid prototyping
- explore an idea by quickly writing a simple program
- Python programs are often much shorter and simpler than programs in other languages like C or C++

⇒ takes less time to write a Python program (depending on the application)

- if the program is not run too often, then shorter development time pays off

Portability, speed, space and program maintenance

- Python is a high level language ⇒ independent of machine specific instructions ⇒ highly portable, i.e. Python program can run with (almost) no changes on different operating systems
- Python is good when processing small and medium size data sets
- for large data sets it is better to use C/C++ (order of magnitude faster and more space efficient)
- standard approach:
  - first write the program in Python to verify algorithmic idea
  - implement time and space critical parts in C/C++
  - interface them with Python wrapper
- program maintenance: activity to keep the program working
  - i.e. bug fixing, adding features, changing input/output formats, porting to other platforms
- with some discipline one can write Python code that is easy to maintain

How to run Python programs on Linux or Mac (1/3)

- make sure that `python3`-interpreter is in your path list

```
$ which python3
/usr/bin/python3
```

- sometimes, the version of Python is relevant:

```
$ python3 --version
Python 3.4.6
```

- notation: the symbol $ in the command lines shown above stands for the *prompt* of the command line interpreter, the bash in our case
- one can configure the prompt to, for example, show the name of the computer and the ordinal number of the command typed
- try to run the Python script in file `myfile.py`
- you may have to give the path of `myfile.py` if you are not in the directory in which `myfile.py` resides, for example

```
$ python3 Basic/myfile.py
...
```

How to run Python programs on Linux or Mac (2/3)

- to simplify running a Python script, insert the following magic string as its first line

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

- this saves prepending `python3` to each call
- magic string beginning with `#!` executes the command (including arguments) which interprets the rest of the file
- python3-interpreter may be installed in different paths (depending on the system)

⇒ here one uses the program `env` (installed in `/usr/bin/`) which evaluates the `PATH`-Variable to look for the first path containing the executable `python3`

- all our Python scripts will have the magic string in the first line
- we will however usually not show this line in the slides

How to run Python programs on Linux or Mac (3/3)

- additionally `myfile.py` must be made an executable by executing:
  `$ chmod u+x myfile.py`
- this needs to be done only once
- typing
  `$ ./myfile.py`
  runs your script
- if `PATH` contains the current directory `.`, then `myfile.py` suffices:

```
$ export PATH=${PATH}:.
$ myfile.py
```

Finding help

- Python comes with an online help
  - start python3 and type `help()`
  - type any keyword
  - for example: keyword `open` will deliver detailed information about this method

`open`(file, mode='r') ⇒ stream

Open file and return a stream. Raise `IOError` upon failure.
Example:

```
stream_r = open('testfile', 'r')
stream_w = open('newfile', 'w')
```

- there are many sources of information about Python in the Web, e.g.
  - `https://www.python.org`
  - `https://docs.python.org`

The programming process: a case study

- solve problem of counting regulatory sequences in DNA
- regulatory sequence (r.s.): a segment of a nucleic acid molecule which is capable of increasing or decreasing the expression of specific genes within an organism.
- Examples of r.s.:

  - CAAT box
  - CCAAT box
  - Pribnow box
  - TATA box
  - SECIS element
  - Polyadenylation signal
  - A-box
  - Z-box
  - C-box
  - E-box
  - G-box

- to count r.s. in DNA, go step by step as follows:
  1. identify required inputs (e.g. data or information given by the user)
  2. develop the algorithm
     - clarify relevant data and its representation
     - specify steps for producing the intended result
  3. specify the format of the output (use standards as much a possible, e.g. tabular output in .tsv, sequence output in FASTA, ...)
  4. write the Python code

The design phase (1/1)

– collect necessary information from the user:

- where does the input (DNA and regulatory sequences) come from (e.g. filename, other programs etc.)?
- format of input and output?
- expected size of the input/output?

answers ...

to these questions are specified in exercise sheets

– develop algorithm to perform search for regulatory sequences, e.g.

take every regulatory sequence and search it in the DNA

pseudocode

```
get name of DNA file from user
read in DNA from file
for each regulatory sequence
  if sequence is contained in DNA, then
    add one to the count
print count
```

before implementing this pseudocode, need to learn basics of Python

Syntax rules (1/2)

- statements usually appear on single lines, but can be split if necessary

```
print('This is a long string I want to output, so I better ',
      'split it, as my teacher requires that lines are not ',
      'longer than 80 characters')
```

- semicolon at end of statement is **not** necessary
- syntax is sensitive of indentation, i.e. statement depending on other statement is indented with respect to this (usually 2 or 4 blanks)

```
if 1 == 1:
  print('1 equals 1')
```

- such an indented statement is called block
- the statements on which the block depends ends with a colon :
- variables do not have to be declared; a variable springs into existence once we assign an object to it
- following an initial letter, an identifier can be any combination of letters, digits and underscores

Syntax rules (2/2)

- convention 1: multiword variables or function names are written with underscores between the words, like in `initial_prime_number = 13`
- convention 2: multiword class names are written with MixedCase (with each word capitalized), like in `class FractionSimple ...`

rules for different kinds of identifiers, to ease readability:

package/module name	lower case
class name	upper case
function name	lower case
function argument	lower case, begins with `self` for instance method
method argument	lower case, begins with `self` for class method
constants	upper case

Representing sequence data (1/3)

- our case study has to handle sequences
- here we show how to represent and manipulate sequences representing DNA and proteins in Python
- DNA consists of nucleic acids (nucleotides, bases)

`A`	Adenine
`C`	Cytosine
`G`	Guanin
`T`	Thymine

Additionally:

`U` Uracil (for RNA)

`N` unknown base

- notation: DNA is a sequence of bases in upper or lower case

Representing sequence data (2/3)

- a protein consists of 20 aminoacids

C	Cysteine	Cys
A	Alanine	Ala
R	Arginine	Arg
N	Asparagine	Asn
D	Aspartic acid	Asp
Q	Glutamine	Gln
E	Glutamic acid	Glu
G	Glycine	Gly
H	Histidine	His
I	Isoleucine	Ile

L	Leucine	Leu
K	Lysine	Lys
M	Methionine	Met
F	Phenylalanine	Phe
P	Proline	Pro
S	Serine	Set
T	Threonine	Thr
W	Tryptophan	Trp
Y	Tyrosine	Tyr
V	Valine	Val

- notation: a protein is a sequence of aminoacids (one letter code and uppercase)
- sequence representation is often a simplification of reality
- but suffices for this course

Representing sequence data (3/3)

some computer science terms:

- each of the two tables above defines an *alphabet*, i.e. a finite set of symbols
- *string*: sequence of symbols
- in general: computers use ASCII or superset thereof (like UTF-8)
- ASCII contains 128 characters numbered from 0 to 127
- each member of the ASCII alphabet denotes printable or non-printable character
- for example: ASCII 65 is `A`, ASCII 10 is newline (\n), etc.
- no need to remember these numbers, as one can easily convert characters into their number representation using appropriate operators in Python (e.g. `ord('A')`) (or any other language)

A Python script to store a DNA sequence

- first we store the DNA in a variable called `dna`

```
dna = 'ACGGGAGGACGGGAAAATTAC' # assign string literal to var.
```

- next we print the DNA onto the screen

```
print(dna)                    # call print-function
```

- finally, we'll specifically tell the program to exit.

```
exit(0)                  # call exit-function, return code 0
```

- store the previous lines (without the comments) in a textfile, say `example4-1.py`
- on a Linux-Shell type the following two lines following the $-symbol

```
$ chmod ug+x example4-1.py
$ example4-1.py
```

```
ACGGGAGGACGGGAAAATTAC
```

- grey box displays output of Python script in terminal window
- occasionally, we report how the script is called from shell

- statements of script are executed step by step from top to bottom
- comments begin with the symbol # and end at the end of the line

Variables and assignments

- name of variable is arbitrary
- composed of upper and lower case letters, digits, and underscore _
- choose appropriate variable names
- name should reflect what the variable is for ⇒ self documenting code
- string is enclosed in single quotes to make it a string literal
- double quotes would also work
- when omitting the quotes Python interpreter would consider `ACG..` as an identifier which has some meaning in the program (which it does not have)
- = is the assignment operator: variable to the left and expression to the right
- after assignment variable stores the assigned value
- use the variable to print the DNA sequence

Concatenating DNA fragments (1/3)

```
# store two DNA sequences into two variables called dna1 and dna2
dna1 = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC'
dna2 = 'ATAGTGCCGTGAGAGTGATGTAGTA'

# print the DNA onto the screen
print('Here are the original two DNA sequences:')
print(dna1)
print(dna2)

# concatenate DNA sequences into a 3rd var with format
dna3 = '{}{}'.format(dna1, dna2)
print('concatenation of the first two sequences (version 1):')
print(dna3)

# alternative way using the concatenation operator +:
dna3 = dna1 + dna2
print('Concatenation of the first two sequences (version 2):')
print(dna3)

# print the same thing without using the variable dna3
print('Concatenation of the first two sequences (version 3):')
print(dna1 + dna2)
```

Concatenating DNA fragments (2/3)

```
Here are the original two DNA sequences:
ACGGGAGGACGGGAAAATTACTACGGCATTAGC
ATAGTGCCGTGAGAGTGATGTAGTA
concatenation of the first two sequences (version 1):
ACGGGAGGACGGGAAAATTACTACGGCATTAGCATAGTGCCGTGAGAGTGATGTAGTA
Concatenation of the first two sequences (version 2):
ACGGGAGGACGGGAAAATTACTACGGCATTAGCATAGTGCCGTGAGAGTGATGTAGTA
Concatenation of the first two sequences (version 3):
ACGGGAGGACGGGAAAATTACTACGGCATTAGCATAGTGCCGTGAGAGTGATGTAGTA
```

Concatenating DNA fragments (3/3)

- the statement `dna3 = '{}{}'.format(dna1,dna2)` concatenates the contents of variable `dna1` and `dna2` and stores the result in `dna3`
- each `{}` acts as a placeholder in which `format` injects the given arguments to build the resulting string
- a simpler way to do the concatenation uses operator `+`:

```
dna3 = dna1 + dna2
```

- variable can hold a string (as in the example) but also an integer, a floating-point number, or boolean value

```
num1 = 42
num2 = 56
print('sum is',num1 + num2)
print('sum is {}'.format(num1 + num2))
```

```
sum is 98
sum is 98
```

Transcription: DNA to RNA (1/2)

```
import re       # use methods from class re

dna = ’ACGGGAGGACGGGAAAATTACTACGGCATTAGC’

# print the DNA onto the screen
print(’Here is the DNA:’)
print(dna)

# transcribe the DNA to RNA by substituting all T’s with U’s.
rna = re.sub(’T’, ’U’, dna)   # use method sub from class re

print(’Here is the result of transcribing the DNA to RNA:’)
print(rna)
```

```
Here is the DNA:
ACGGGAGGACGGGAAAATTACTACGGCATTAGC
Here is the result of transcribing the DNA to RNA:
ACGGGAGGACGGGAAAAUUACUACGGCAUUAGC
```

Transcription: DNA to RNA (2/2)

- statement involving `sub` makes copy of DNA sequence in variable `rna`
- transcription is expressed by the `sub`-method

```
rna = re.sub('T','U',dna)
```

- operator . separates class name `re` from `sub`

`re.sub`(pattern, repl, string) ⇒ new string

return the string obtained by replacing the leftmost non-overlapping occurrences of the pattern in string by the replacement repl. repl can be either a string or a callable; Example:

```
re.sub(r'[aeiou]','*','hello')          ⇒   'h*ll*'
re.sub(r'([aeiou])', r'<\1>','hello')   ⇒   'h<e>ll<o>'
```

Calculating the reverse complement (1/5)

```
import re, string

dna = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC'
print('Here is the DNA:')
print(dna)

revcom = reverse(dna)

revcom = re.sub('A','T', revcom)
revcom = re.sub('T','A', revcom)
revcom = re.sub('G','C', revcom)
revcom = re.sub('C','G', revcom)

print('Here is the incorrect result:\n{}'.format(revcom))
```

- first step copies DNA into new variable `revcom` in reverse order
- this is done by applying a method reverse, explained later
- next step substitutes all bases by their complements
- the last step will lead to incorrect result

Calculating the reverse complement (2/5)

```
Here is the DNA:
ACGGGAGGACGGGAAAATTACTACGGCATTAGC
Here is the incorrect result:
GGAAAAGGGGAAGAAAAAAAGGGGAGGAGGGGA
```

- the reverse complement should have all the bases in it, since the original DNA had all the bases,
- but ours only has A and G
- problem: first two substitute commands above
  - change all A's to T's (so there are no A's) and then
  - change all T's to A's

⇒ so all original A's and T's are all now A's

- same thing happens to the G's and C's all turning into G's
- for the correct version we make a new copy of the DNA (luckily we saved the original in variable `dna`)

Calculating the reverse complement (3/5)

- and use `str.maketrans(intab, outtab)` returning a translate table to be used in `translate` function

```
revcom = reverse(dna)
transtab = str.maketrans('ACGTacgt','TGCAtgca')
revcom = revcom.translate(transtab)
print('Here is the reverse complement DNA:\n{}'.format(revcom))
```

```
Here is the reverse complement DNA:
GCTAATGCCGTAGTAATTTTCCCGTCCTCCCGT
```

Calculating the reverse complement (4/5)

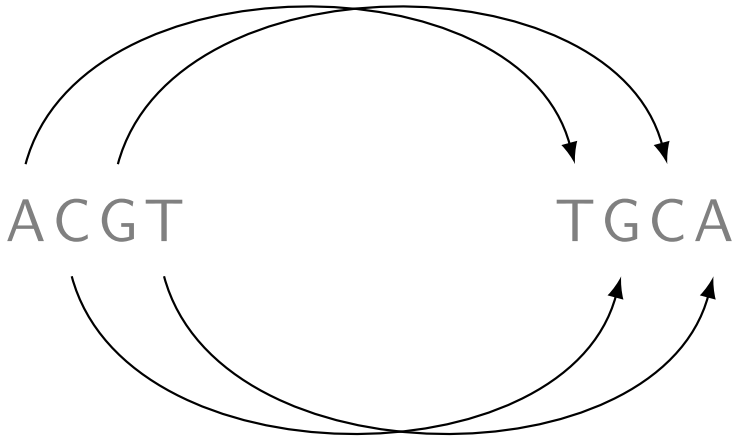
Recapitulate the algorithmic idea

- apply the substitution step by step
- look at each base one at a time, make the change to the complement
- then look at the next base in the DNA

- `maketrans/translate`-methods are exactly suited for this task:
  - `str.maketrans('ACGT','TGCA')` creates translation table specifying correspondence between characters in `ACGT` and `TCGA`
  - `translate()` applies table to `revcom`

- each character in the first string is translated into the character at the same position in the second string
- to handle upper case and lower case character we use

```
str.maketrans('ACGTacgt','TGCAtgca')
```



Calculating the reverse complement (5/5)

string.translate(table) ⇒ new string

return a copy of the string in which each character has been mapped through the given translation table, created e.g. by `str.maketrans`
Example:

```
'hello'.translate(str.maketrans('aeiou', '*****'))   ⇒   'h*ll*'
'hello'.translate(str.maketrans('el', 'ip'))         ⇒   'hippo'
```

str.maketrans(x,y) ⇒ dictionary

Return a translation table usable for `str.translate()`. Example:

```
maketrans('abcde','BCDEF')   ⇒   {97: 66, 98: 67, 99: 68, 100: 69, 101: 70}
```

- a dictionary is a data structure which allows to store key/value pairs
- in our case the keys are the 5 lower case letters and the values are the 5 upper case letters, all represented by their ASCII-number
- we will later go into detail about dictionaries, see frame 153

Reading proteins in files (1/9)

- previous examples: sequences were hard coded in the script
- but usually sequences are stored in a separate file
- for example, the file `NM_021964fragment.pep` stores

  ```
  MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD
  SVLQDRSMPHQEILAADEVLQESEMRQQDMISHDELMVHEETVKNDEEQMETHERLPQ
  GLQYALNVPISVKQEITFTDVSEQLMRDKKQIR
  ```

- a file is a stream of characters usually stored on a disk
- the stream usually contains separators such as \n to simplify readability

Reading proteins in files (2/9)

Example

The file which appears in the editor as

```
MNIDDKL
SVLQ
GLQYA
```

is stored as a sequence of numbers (i.e. ASCII codes) 77, 78, 73, 68, 68, 75, 76, 10, 83, 86, 76, 81, 10, 71, 76, 81, 89, 65, 10 representing the string `MNIDDKL\nSVLQ\nGLQYA\n`

- on the level of the operating system, the contents of a file can only be read sequentially, character by character (i.e. ASCII code by ASCII code)
- fortunately, for convenience, Python provides methods to sequentially read a file in units of lines

Reading proteins in files (3/9)

- before we can access a file, we need to open it

Names of files

- filenames can be arbitrary, but ideally they should somehow reflect the file contents
- e.g. file above is from the record in the Genbank database with ID `NM_021964`
- sequence is a fragment of a protein sequence translated from the DNA stored in this record
- remark: do not use filenames including spacers, as many Linux-tools processing lists of filenames (such as `find`) do not work as supposed

Reading proteins in files (4/9)

```
# filename of the file containing the protein sequence data
proteinfilename = 'NM_021964fragment.pep'

# first create a new stream for reading, named protein_stream
protein_stream = open(proteinfilename, 'r')

# read protein sequence data from stream by calling readline method
protein = protein_stream.readline()

# now that we've got our data, we can close the stream
protein_stream.close()

# display protein sequence
print('Here is the protein:\n{}'.format(protein),end='')
```

```
Here is the protein:
MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD
```

Reading proteins in files (5/9)

- `open` method opens the file for reading and delivers a `stream` object named `proteinfile`
- all interactions with the file are done via stream object, which allows sequential access to the file
- interactions: reading, writing, searching, erasing the file content
- `readline` method reads a single line of the file (namely the first line)
- this line is stored in the string object `protein` which is displayed using the `print`-function
- last argument `end=''` of `print`-function prevents display of trailing `\n`
- this is necessary as `readline` delivers the line including a trailing `\n`
- omitting `end=''` would lead to display of two `\n` and thus an empty line following the protein sequence

Reading proteins in files (6/9)

```
# filename of file containing the protein sequence data
proteinfilename = 'NM_021964fragment.pep'

protein_stream = open(proteinfilename, 'r')

# Since the file has three lines, and since readline only returns
# one line, we'll read a line and print it, three times.

protein = protein_stream.readline()
print('First line of protein file:\n{}'.format(protein),end='')

protein = protein_stream.readline()
print('Second line of protein file:\n{}'.format(protein),end='')

protein = protein_stream.readline()
print('Third line of protein file:\n{}'.format(protein), end='')

# Now that we've got our data, we can close the file.
protein_stream.close()
```

Reading proteins in files (7/9)

```
First line of protein file:
MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD
Second line of protein file:
SVLQDRSMPHQEILAADEVLQESEMRQQDMISHDELMVHEETVKNDEEQMETHERLPQ
Third line of protein file:
GLQYALNVPISVKQEITFTDVSEQLMRDKKQIR
```

- program reads in sequence line by line
- every time the contents of the current line is bound to variable `protein`
- the stream `protein_stream` remembers where the previous read ended and continues from here in the next call of `readline`
- in `print`-statements we omit a second `\n`, as one is already present at the end of the read sequence
- drawback of above program: each line in the file requires extra code

Reading proteins in files (8/9)

- we now consider a more complete solution which works for a file with an arbitrary number of lines

```
filename = 'NM_021964fragment.pep'
print('Try to open "{}"'.format(filename))
stream = open(filename,'r')


# read lines delivered by stream and print them
for line in stream:
  print('next line is {}'.format(line), end='')
stream.close()
```

- the `for`-loop iterates over the lines delivered by the stream
- in each iteration the variable `line` contains the current line (including the `\n`)
- line is displayed using `print` with `end=''`, to prevent output of extra `\n`
- after the `for`-loop, `stream` is closed again

Reading proteins in files (9/9)

```
Try to open 'NM_021964fragment.pep'
next line is MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD
next line is SVLQDRSMPHQEILAADEVLQESEMRQQDMISHDELMVHEETVKNDEEQMETHERLPQ
next line is GLQYALNVPISVKQEITFTDVSEQLMRDKKQIR
```

- next we consider how to store the lines that we have read
- for this we need to introduce lists

Lists (1/11)

- lists allow to store a sequence of values
- in Python the values do not necessarily need to be of the same type (as in most other programming languages)
- Python also provides arrays (via the module `array`) which are less flexible (all elements have same type), but more space efficient

```
# filename of file containing the protein sequence data
proteinfilename = 'NM_021964fragment.pep'

# create a stream, without exception handling to keep it simple
protein_stream = open(proteinfilename, 'r')

# read protein sequence data from file, and store it in list
proteins = protein_stream.readlines()

# iterate over the elements in the list and generate their index
for idx, protein in enumerate(proteins):
  print('{}: {}'.format(idx, protein), end='')

protein_stream.close()
```

Lists (2/11)

```
0: MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD
1: SVLQDRSMPHQEILAADEVLQESEMRQQDMISHDELMVHEETVKNDEEQMETHERLPQ
2: GLQYALNVPISVKQEITFTDVSEQLMRDKKQIR
```

- advantage: only one read statement

  `proteins = protein_stream.readlines()`

- each element in list and its corresponding index (beginning with 0) is generated by `enumerate`-method
- we also could have generated the index value and use them to access the list, but the above solution is the preferred
- let us now consider the most important operations on lists

Lists (3/11)

```
bases = ['A', 'C', 'G', 'T']
print('list elements: {}'.format(bases))
print('first element: {}'.format(bases[0]))
print('second element: {}'.format(bases[1]))
print('third element: {}'.format(bases[2]))
print('fourth element: {}'.format(bases[3]))
```

```
list elements: ['A', 'C', 'G', 'T']
first element: A
second element: C
third element: G
fourth element: T
```

- list elements are specified as comma separated list of elements
- each element is a so called string-literal, which must be quoted using single or double quotes

Lists (4/11)

– variation: print the elements one after each other separated by /:

```
bases = ['A', 'C', 'G', 'T']
print('list elements: {}'.format('/'.join(bases)))
```

```
list elements: A/C/G/T
```

sep.join(list) ⇒ string

Returns a string which is the concatenation of the strings in the list. The separator between elements is the string *sep*.
Example:

```
''.join(['ab', 'c', 'de'])   ⇒   'abcde'
', '.join(['1', '2', '3'])   ⇒   '1, 2, 3'
```

– note that the list appears as an argument of the method, while most other list methods are applied to a list using the .-operator

Lists (5/11)

- take an element off at the end of the list with `pop`

```
bases = ['A', 'C', 'G', 'T']
base1 = bases.pop()
print('element removed from end: {}'.format(base1))
print('remaining list of bases: {}'.format(bases))
```

```
element removed from end: T
remaining list of bases: ['A', 'C', 'G']
```

Lists (6/11)

- take an element off at the beginning of the list with `pop(0)`

```
bases = ['A', 'C', 'G', 'T']
base2 = bases.pop(0)
print('element removed from beginning: {}'.format(base2))
print('remaining list of bases: {}'.format(bases))
```

```
element removed from beginning: A
remaining list of bases: ['C', 'G', 'T']
```

Lists (7/11)

- put an element at the beginning of the list with `insert`

```
bases = ['A', 'C', 'G', 'T']
base1 = bases.pop()
bases.insert(0, base1)
print('element from end put on beginning: {}'.format(bases))
```

```
element from end put on beginning: ['T', 'A', 'C', 'G']
```

- `insert` is a general method which puts an element after the position given as first argument

Lists (8/11)

- put an element on the end of the list with `append`

```
bases = ['A', 'C', 'G', 'T']
base2 = bases.pop(0)
bases.append(base2)
print('element from beginning put on end: {}'.format(bases))
```

```
element from beginning put on end: ['C', 'G', 'T', 'A']
```

Lists (9/11)

- get the length of a list with `len`:

```
weekdays = ['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun']
print('days in week: {}'.format(len(weekdays)))
```

```
days in week: 7
```

Lists (10/11)

- return a new list, which is the slice of another list, specified by its first index and last index (excluded):
- if we leave out the last index or the last index is larger than the index of the last list element, then the slice ends with the last list element

```
workdays = weekdays[0:5]
weekend = weekdays[5:]
print('workdays: {}'.format(workdays))
print('weekend: {}'.format(weekend))
```

```
workdays: ['mon', 'tue', 'wed', 'thu', 'fri']
weekend: ['sat', 'sun']
```

Lists (11/11)

– insert an element at an arbitrary place in a list with `insert`:

```
atoms = ['Hydrogen', 'Helium', 'Beryllium', 'Boron']
atoms.insert(2,'Lithium')
print('list with element inserted after 2nd elem:\n{}'
       .format(atoms))
```

```
list with element inserted after 2nd elem:
['Hydrogen', 'Helium', 'Lithium', 'Beryllium', 'Boron']
```

synopsis: methods on lists (ordered by importance)

`l = list()`	create a new empty list
`len(l)`	deliver length of list `l`
`sep.join(slist)`	concat. list `slist` of strings with separator `sep` to one string
`l.append(e)`	add element `e` to end of list `l`
`l.pop()`	delete elem. at end of list `l` and return deleted elem.
`l[i:j]`	get slice of list `l` from index `i` to index `j-1`
`enumerate(l)`	enumerate the indexes and elements of list `l`
`l.insert(0,e)`	put element `e` at beginning of list `l`
`l.pop(0)`	delete elem. at beginning of list `l` and return deleted elem.
`l.insert(i,e)`	insert (in-place) element `e` after position `i` in list `l`

Command line arguments and lists (self study) (1/2)

- earlier we have seen how we can access the contents of a file
- this way of providing relevant information (e.g. data) to a program is very common
- but there are other kinds of information you want to provide a program with, such as:
  - input files or output files
  - options which trigger specific behavior in a program (e.g. to use a specific of several possible algorithms to solve a problem)
  - an option which shows a help line
  - an option to produce verbose output.
- it would be inconvenient to specify these in files
- it is common to provide such information on the command line following the name of the program (as almost every Unix-command does)
- the access to the command line is via the list `sys.argv` of strings (requires `import sys`)

Command line arguments and lists (self study) (2/2)

- suppose the following Python code is in a file `argv_example.py`, which has been made executable

```
#!/usr/bin/env python3
import sys
for idx, arg in enumerate(sys.argv):
  print('argv[{}]="{}"'.format(idx,arg))
```

- execute it on the command line, providing any number of arguments:

```
$ argv_example.py ab 0.5 'string with space'
argv[0]="argv_example.py"
argv[1]="ab"
argv[2]="0.5"
argv[3]="string with space"
```

- arguments are printed line by line with their index in `sys.argv`
- program name is at index 0,
- all elements of `sys.argv` are strings (even the number)

Common student question

- Q: Does python allow lists of elements of different kinds?
- A: Yes, we can e.g. have a list `['a',1,0.5,[1,'3']]`

Some of the *common students questions* are from the textbook *Learn Python: the hard way*, 3rd edition, Zed A. Shaw, Addison Wesley, Upper Saddle River, NJ

Key points on lists

- `[value1, value2, value3]` creates a list of length 3

```
l = list()
l.append(1)
l.append(4)
```

creates a list of length 2.

- Lists are indexed and the range of indexes is from 0 to $n-1$ for a list of length n
- Lists are mutable (i.e., their values can be changed in place).
- there are some built-in lists (like `sys.argv`) and many methods to conveniently create and modify lists, see frame 62

Formatted printing (1/2)

- in many situations one needs to specify features of a value output
  - precision of floating point number
  - width of the field in which a value is shown
  - left or right adjustment in that field
- this can be done by inserting format characters using the :-notation inside a pair of curly braces

```
f = 2323.14159265
i = 76
s = 'hello world'

print('float   "{:>10.4f}"'.format(f))
print('integer "{:<5d}\"'.format(i))
print('string  "{:>12s}"'.format(s))
```

output:

```
float   ' 2323.1416'
integer '76   '
string  ' hello world'
```

format char	kind of value
`f`	floating point
`d`	integer
`s`	string

format char.	adjustment
`<`	left
`>`	right

format char.	semantic
`.4`	precision 4
`10`	field of width 10

Formatted printing (2/2)

- the curly braces encloses optional format characters to specify
  - minimum field width
  - precision (for floats)
  - length modifier
  - alignment modifier
- `{:>10.4f}` means that floating point is printed right adjusted with minimum width 10 characters (padded with spaces if necessary) and at most 4 positions for the decimal part
- `{:<5d}` means that integer is printed left adjusted in a field of 5 characters
- `{:>12s}` means that string is printed right adjusted in a field of 12 characters

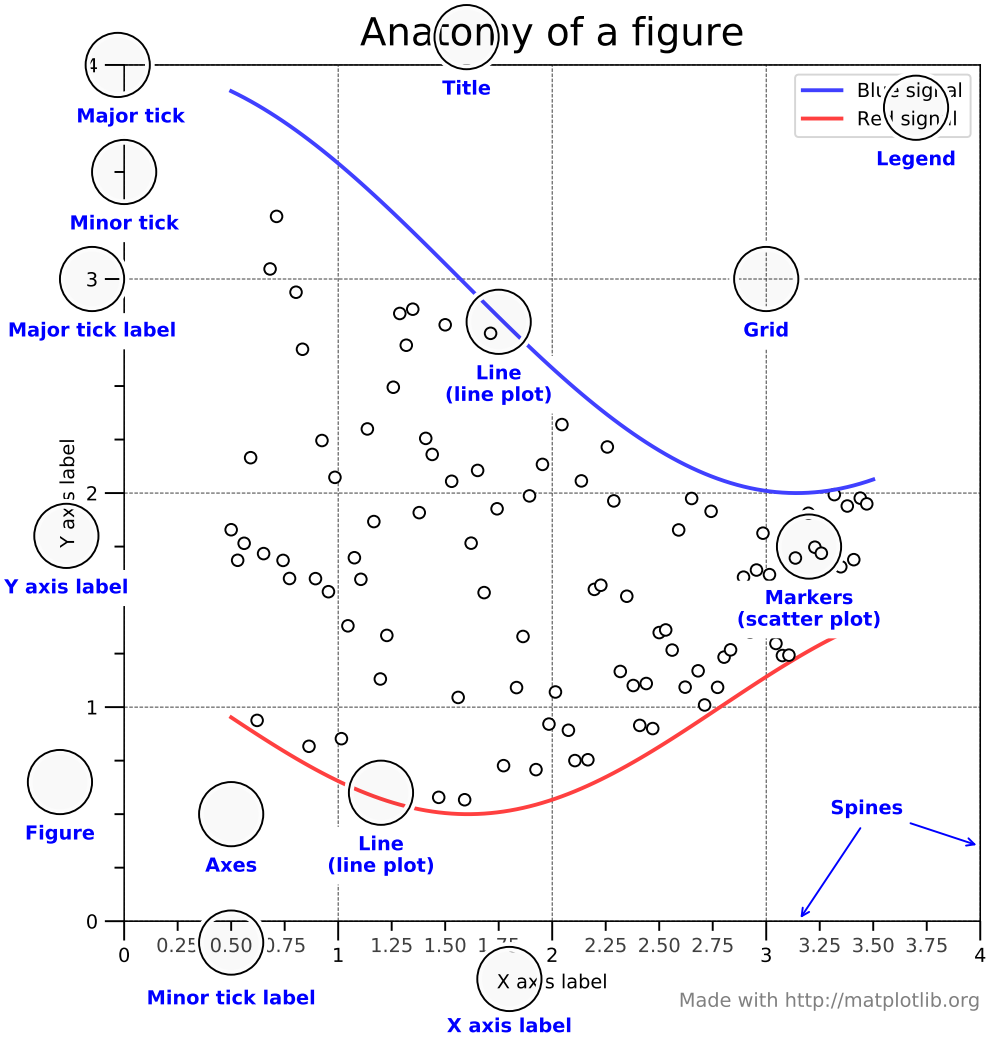
Plotting data (1/7)

- Python3 provides many different ways of plotting data
- here we focus on matplotlib, the most widely used 2-D plotting library for Python3
- matplotlib emulates Matlab like graphs and visualizations (hence its name)
- the following figure explains the terminology of matplotlib and shows the anatomy of a figure
- the corresponding PDF of the figure was generated by a python script available at `https://matplotlib.org/gallery/showcase/anatomy.html`

- this section was originally planned to be presented later in this course, but as plotting is required for Physik 1, it is already presented here (hopefully early enough)

this presentation follows: `https://towardsdatascience.com/data-visualization-using-matplotlib-16f1aae5ce70`

Plotting data (2/7)



- a figure represents the overall window in which one or more plots appear
- figure contains ≥ 1 axes in which actual graphs are plotted
- the different attributes of an axes can be set by corresponding methods, like `ax.set_title('title')` for an axes-object `ax`

Plotting data (3/7)

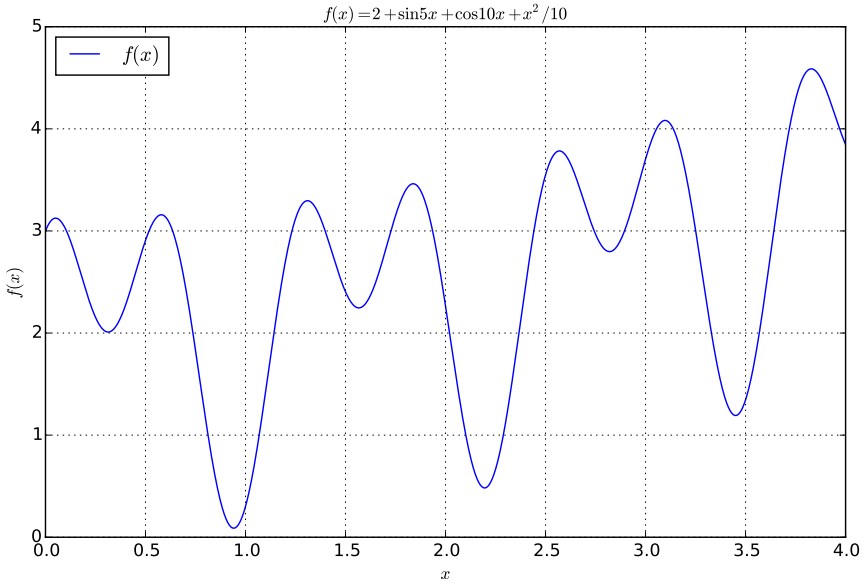
- the axes of a figure can be put above or beside each other, like in this figure
- every axes has an x-axis and y-axis for plotting
- as attributes of axes one specifies titles, labels, spines (i.e. boundaries)
- with the x-axis and y-axis of an axes one may specify ticks and ticks labels
- in matplotlib, `pyplot.subplots` is used to create figures and axes and to specify the characteristics of figures (like their size)
- there are many different forms of plots, like line plot, scatter plot, bar plots (histograms), boxplot, violin plots, stack plots
- we start with plotting a mathematical function as a line plot
- later we consider plotting meta data about Genbank, finally time series data
- we even learn how to create interactive plots, using a different library

Plotting data (4/7)

– consider $f(x) = 2 + \sin 5x + \cos 10x + \frac{x^2}{10}$ implemented as follows:

```
def curvedM(x):
  return 2.0 + math.sin(5.0 * x) + math.cos(10.0 * x) + 0.1 * x * x
```

– the name `curvedM` is motivated by its shape:



Plotting data (5/7)

- such a plot can easily be generated using Python's `matplotlib.pylib`-module
- the main task is to supply the plotting function with two lists of the same length, one for the values on the X-axes and one for the corresponding values on the Y-axes
- these can e.g. be experimental measurements or (as in our case),
  - `numpoints` values of x, $x_{\text{min}} \leq x \leq x_{max}$ for some user defined real valued boundaries x_{min}, x_{max} and positive integer `numpoints`, and
  - corresponding function values $f(x)$, $x_{\text{min}} \leq x \leq x_{max}$
- the next two frames present the code for creating the previously shown plot

Plotting data (6/7)

```
x_min = 0.0
x_max = 4.0
numpoints = 10000
stepwidth = \
 (x_max - x_min)/numpoints
x_list = list()
y_list = list()
for p in range(numpoints+1):
  x = x_min + p * stepwidth
  x_list.append(x)
  y_list.append(curvedM(x))
y_min = min(y_list)
y_max = max(y_list)
```

- in the first part we fix the min/max. X-value and the number of points
- the requested number of points on the X-axes are evenly distributed on the range from x_{min} to x_{max}, according to the value of `stepwidth`
- these points are stored in `x_list` and the corresponding function values in `y_list`
- finally we determine the minimum and maximum `y_min` and `y_max` of `y_list`, respectively
- there are many other possible ways to prepare the two lists
- one may use more efficient libraries like `numpy` to create them
- or one may use a different function or read the x/y-pairs from a file

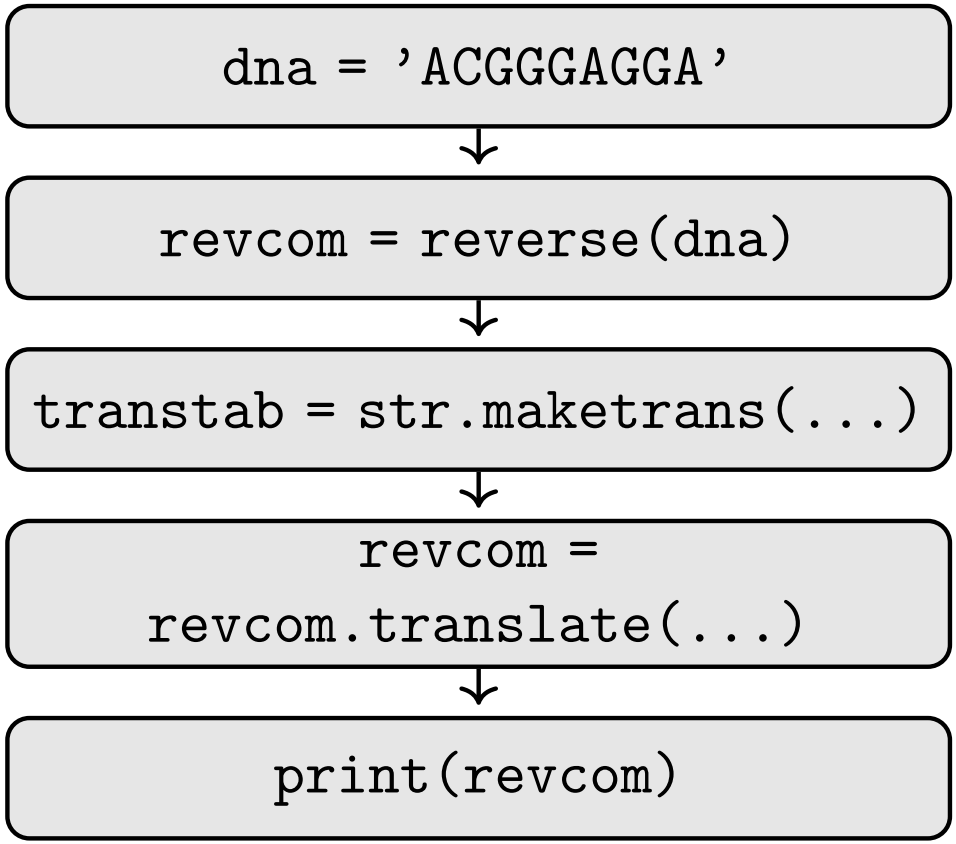
Plotting data (7/7)

- the second part takes the prepared data and prints it, using appropriate methods from the `plt`-module
- module name is an abbreviation introduced with `import`-statement
- the figure (saved as pdf-file) includes a legend and axes annotations using LaTeX-notation enclosed in $'s, see plot on frame 68

```
import matplotlib.pyplot as plt
plt.switch_backend('agg') # to allow remote use

fig, ax = plt.subplots(figsize=(10, 6.18)) # golden section
ax.set_xlabel('$x$')
ax.set_ylabel('$f(x)$')
ax.grid(True)
ax.set_xlim(x_min,x_max)
ax.set_ylim(min(0.0,y_min),math.floor(y_max+1.0))
ax.set_title('$f(x)=2+\sin 5x+\cos 10x+x^{2}/10$',  # LaTeX
             fontsize=12, color='black')
ax.plot(x_list, y_list, color='blue',label='$f(x)$')
ax.legend(loc='upper left')
fig.savefig('curvedMplot.pdf')
```

Conditional Statements (1/8)

- up until now we only have seen scripts without any control structures (only exception was the `for`-loop to enumerate lines)
- scripts are executed step by step in sequential order, as in the following case



- two ways to organize the execution in other ways: conditional statements (for branching) and loops (for iteration)

Conditional Statements (2/8)

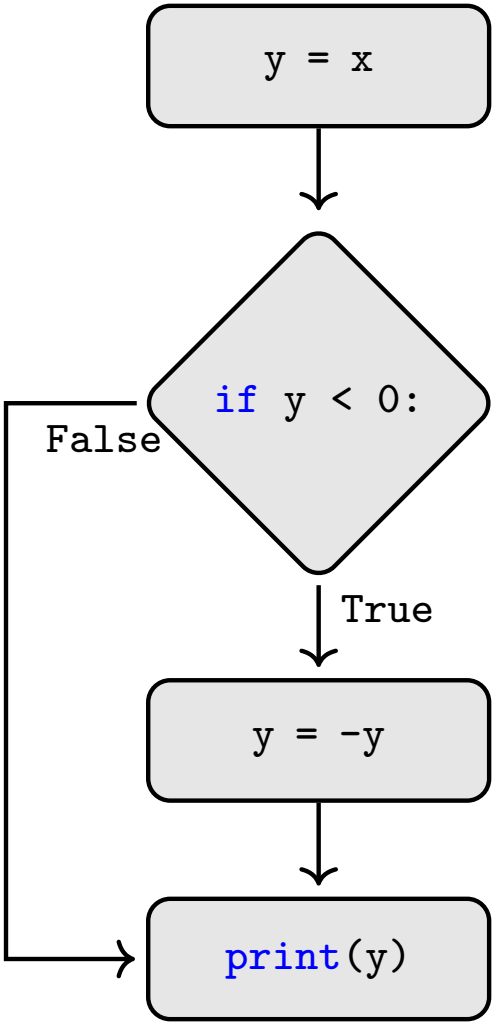
- we often have to execute statements only if certain conditions hold

Example

- Suppose we want to determine the absolute value of some integer variable `x` and print it
- we first assign `x` to `y` (in case we need the original value of `x` later)
- if `y` is negative, we make it positive
- otherwise we are done and in both cases we print `y`

```
y = x
if y < 0:
  y = -y      # indent relative to if
print(y)
```

control flow

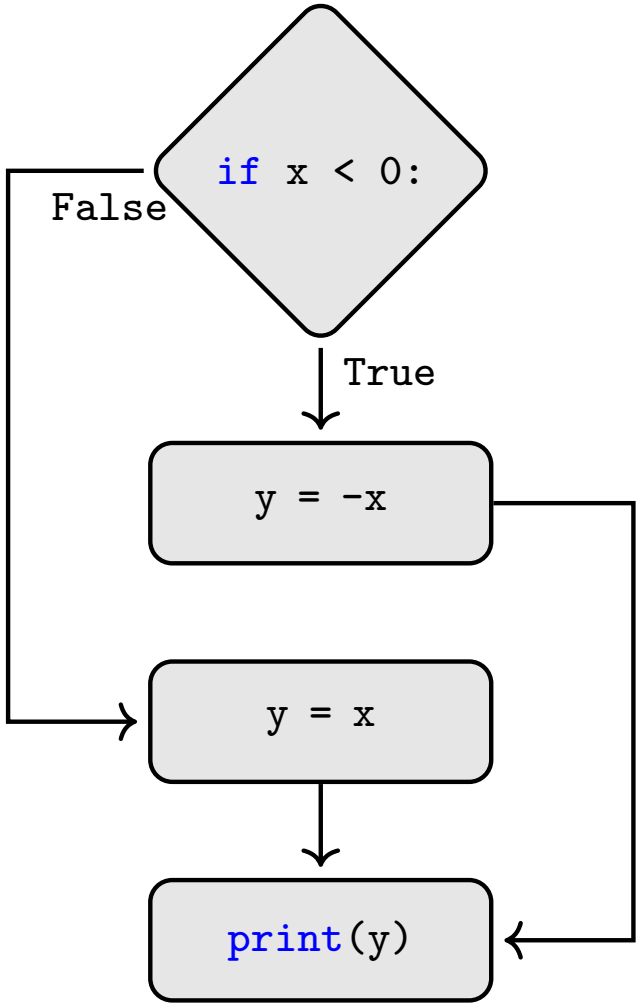


Conditional Statements (3/8)

Example (continued)

- instead of first assigning `x` to `y` we can also directly compute the absolute value in `y` using an `if`/`else`-statement
- the `if`-case handles negative values
- the `else`-case handles positive values

```
if x < 0:
  y = -x    # indent relative to if
else:
  y = x     # indent relative to else
print(y)
```



Conditional Statements (4/8)

- more examples on conditional statements:

```
if 1 == 1:
  print('1 equals 1')
```

output:

```
1 equals 1
```

- test for equality is written with the operator `==` (a single equation symbol is used for assignments)
- condition must be followed by `:` and statements depending on this (called block) must be indented (we use 2 blanks)
- as test for equality evaluates to `True`, the block is executed
- no parentheses around boolean expression necessary (as in many other languages, like e.g. C)
- `1` (as any number different from 0) evaluates to `True`

```
if 1:
  print('1 evaluates to True')
```

Conditional Statements (5/8)

- `if` can optionally be followed by an `else`

```
if 1 == 0:
  print('1 equals 0')
else:
  print('1 does not equal 0')
```

- `not` can be used for negation

```
if not 1 == 0:
  print('1 does not equal 0')
```

- which can also be expressed using the inequality operator `!=`

```
if 1 != 0:
  print('1 does not equal 0')
```

Conditional Statements (6/8)

- conditional statements (as any control construct), can be nested
- result of one condition possibly triggers next conditional statement

Example

- let `score` be variable storing number of points achieved in exam
- we want to convert the score into a grade

```
if score >= 85:
  print('sehr gut')
else:
  if score >= 70:
    print('gut')
  else:
    if score >= 55:
      print('befriedigend')
    else:
      if score >= 40:
        print('ausreichend')
      else:
        print('nicht bestanden')
```

```
if score >= 85:
  print('sehr gut')
elif score >= 70:
  print('gut')
elif score >= 55:
  print('befriedigend')
elif score >= 40:
  print('ausreichend')
else:
  print('nicht bestanden')
```

- `elif`-version uses less syntax and should be preferred

Conditional Statements (7/8)

- instead of conditions on numbers, we can have conditions on strings

```
seq = 'ACGT'
if seq == 'AAAAAA':
  print('may be Poly-A tail')
elif seq == 'TATAAT':
  print('may be a Pribnow Box')
elif seq == 'GGCCAATCT':
  print('may be a CCAAT Box')
else:
  print('Cannot decide if "{}" is regulatory element'.format(seq))
```

```
Cannot decide if "ACGT" is regulatory element
```

- note the use of `"` : it prints the doubleqote inside a single quoted string
- to print `"` inside a doublequoted string, one would have to escape it using \
- note that `elif` is used instead of `else if`

Conditional Statements (8/8)

Common student question

- Q: What happens if multiple `if`/`elif`-conditions are `True`
- A: The evaluation starts at the top and as soon as a condition evaluates to `True`, the corresponding block is executed

Boolean Expressions (1/7)

- now let us focus on the syntax of the conditions following the keywords `if` and `elif`
- conditions can be arbitrary boolean expressions, that is, expressions which evaluate to either `True` or `False`
- here is a list of operators which can be used in boolean expressions

arithmetic operators	relational operators	logical operators
`+, -, *, /, %, **, //`	`==, !=, <, >, <=, >=`	`not, and, or`

- `%` is the modulus operator, so `5 % 2` evaluates to 1
- `**` is the exponentiation operation, so `2**8` evaluates to 256.
- `//` is the integer division operator, so `9//2` evaluates to 4, while `9/2` evaluates to 4.5.

Boolean Expressions (2/7)

- the logical operators `not`, `and`, `or` deliver values according to the following following table

x	y	not x	x and y	x or y
False	False	True	False	False
False	True	True	False	True
True	False	False	False	True
True	True	False	True	True

in boolean expressions

- `0` evaluates to `False`
- any value $\neq 0$ evaluates to `True`

Boolean Expressions (3/7)

Example

- Count the number of nucleotides in a DNA sequence which are purine (i.e. A or G) and which are pyrimidine (i.e. C or T)

```
dna = 'acgactactcgaccatcatcagcca'
count_purine = count_pyrimidine = 0
for base in dna:
  if base == 'a' or base == 'g':
    count_purine += 1
  elif base == 'c' or base == 't':
    count_pyrimidine += 1
```

- note the use of the increment operator `+=` which combine assignment and arithmetic, e.g. `x += 2` is equivalent to `x = x + 2`

assignment operators

`=, +=, -=, *=, /=`

Boolean Expressions (4/7)

parentheses

- the operator have precedences e.g.
  - `*` is evaluated before `+`
  - `and` is evaluated before `or`

  (Punkt- vor Strichrechnung)
- if you are not sure about the precedences, use `( )` around subexpressions to clarify in which order the evaluation should be performed, as in

```
if ((3 <= x) and (x <= 10)) \
    or ((y % 2) != 0):
  if 9 * (x + 2) >= 99:
    print('success')
```

- the \ is used to split a long expression over two lines (must be last character in line)

- the parentheses in the condition could be omitted, as
  - `<=` has higher precedence than `and`
  - `and` has higher precedence than `or`
  - the modulus operator `%` has higher precedence than `!=`
  - `!=` has higher precedence than `or`

Boolean Expressions (5/7)

Common student question

- Q: Why does `'x' and 1` evaluate to `'x'` and not to `True`
- A: Python prefers to return one of the operands as value rather than just `True` or `False`. The correctness of the result is guaranteed:

```
False and 1   ⇒   False
False or 0    ⇒   0
True and 1    ⇒   1
True or 0     ⇒   True
'' and True   ⇒   ''
```

Boolean Expressions (6/7)

Common student question

- Q: Can I use `<>` instead of `!=` as in some other programming languages?
- A: No. This operator is not valid in Python3, but was valid in earlier versions of Python.

Boolean Expressions (7/7)

Common student question

- Q: I get an error like `SyntaxError: invalid syntax` with reference to a specific line of my Python script. What should I do about it?
- A: Start at the mentioned line and check if it is correct.
  - Check if each opening parentheses has a closing one.
  - Check if each opening quote ' or " has a closing one.
  - Check if each `if`, `else`, `elif`-line has a colon at the end.
  - Check if a block is correctly indented relative to the statement it depends on.
  - Check if all keywords like `if`, `else`, `elif`, `print` are correctly spelled.
  - Check if all variables are correctly spelled.

Examples for syntax errors (1/6)

```
if a == 5
  print('Hello world')
```

```
 File "test.py", line 1
   if a == 5 print('Hello world')
                  ^
SyntaxError: invalid syntax
```

```
If a == 3:
  print('hello world')
```

```
File "test.py", line 1
    If a == 3:
             ^
SyntaxError: invalid syntax
```

Examples for syntax errors (2/6)

```
if a == 2:
  print('hello world')
```

```
File "test.py", line 2
    print('hello world")
                        ^
SyntaxError: EOL while scanning string literal
```

```
i = 0
if 2 * (i+1) / (3 * i < 5:
  print('i={}'.format(i))
```

```
File "test.py", line 2
    if 2 * (i+1) / (3 * i < 5:
                             ^
SyntaxError: invalid syntax
```

Examples for syntax errors (3/6)

```
if a == 4:
  print 'hello world'
```

```
File "test.py", line 3
    print 'hello world'
                       ^
SyntaxError: Missing parentheses in call to 'print'
```

```
arr = [1, 2, 3 4]
```

```
File "test.py", line 1
    arr = [1, 2, 3 4]
                   ^
SyntaxError: invalid syntax
```

Examples for syntax errors (4/6)

```
s = 'abc'
print('Here is a string {}'format(s))
```

```
File "test.py", line 2
    print("Here is a string {}"format(s))
                                      ^
SyntaxError: invalid syntax
```

```
print('Here is a string {}'.format(s)
```

```
File "test.py", line 3
                                       ^
SyntaxError: unexpected EOF while parsing
```

Examples for syntax errors (5/6)

```
i = 5
if i > 10:
print(i)
```

```
File "test.py", line 3
    print(i)
        ^
IndentationError: expected an indented block
```

```
countr = 0
if a < 5:
  counter = counter + 1
```

Examples for syntax errors (6/6)

```
Traceback (most recent call last):
  File "test.py", line 1, in <module>
    if counter < 5:
NameError: name 'counter' is not defined
```

- please report more errors, which non-trivially differ from these
- include minimal context of lines for which the error is reported
- include the error message by python3
- from these errors we can all learn a lot about the language

Key points on conditional statements

- use `if` to start a conditional statement
- use `elif` for providing additional tests
- use `else` to provide a default.
- the blocks depending on conditional statements must be indented relative to these
- use `==` to test for equality
- `x and y` is only `True` if both `x` and `y` are `True`
- `x or y` is `True` if either `x`, or `y`, or both, are `True`
- `0`, the empty string, and the empty list evaluate to `False`; all other numbers, strings, and lists evaluate to `True`

Key Points taken from `https://swcarpentry.github.io/python-novice-inflammation/05-cond/`

Being productive when developing software

- you basically need two terminals, side by side and non-overlapping
- use the left for editing the code (needs to be exactly 80 characters wide) and the other for running it (maybe < 80 characters wide)
- have PDF-viewer loaded with lecture notes available
- do not always Google for a solution before you think about it yourself

Basic — 80×25

```
filename = 'NM_021964fragment.pep'
print(Try to open \"{}\"".format(filename))
stream = openr(filename,"r")

for line in stream:
  print("next line is {}".format(line), end='')
stream.close()
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
                                        6,7          Bot
```

Basic — 80×25

```
-rwxr-----   1 stefan  staff   112 Nov  6 20:43 argv-example.py*
-rwxr-x---   1 stefan  staff  4420 Nov  7 20:39 simpleexpr.py*
-rwxr-x---   1 stefan  staff  1841 Nov  7 20:39 loopexamples.py*
-rwxr-x---   1 stefan  staff   772 Nov  7 20:39 example4-6.py*
-rwxr-x---   1 stefan  staff  1515 Nov  7 20:39 example4-4.py*
-rwxr-x---   1 stefan  staff   807 Nov  7 20:39 example4-2.py*
-rw-r-----   1 stefan  staff   779 Nov  7 23:04 errors.py
-rwxr-x---   1 stefan  staff   249 Nov  7 23:16 readprot-plain.py*
limosen [504] readprot-plain.py
  File "./readprot-plain.py", line 9
    for line in stream
                      ^
SyntaxError: invalid syntax
limosen [505] readprot-plain.py
Try to open "NM_021964fragment.pep"
Traceback (most recent call last):
  File "./readprot-plain.py", line 7, in <module>
    stream = openr(filename,"r")
NameError: name 'openr' is not defined
limosen [505] readprot-plain.py
  File "./readprot-plain.py", line 6
    print(Try to open \"{}\"".format(filename))
                ^
SyntaxError: invalid syntax
limosen [505]
```

Loops (1/8)

- loops are used for iterations
- Python provides two kinds of loops: very powerful `for`-loops and standard `while`-loops
- as the first kind is more common, we start with it
- `for`-loops allow to iterate over some items, such as lines of an open file, characters in a string or elements of a list
- such items are called *enumerable* or *iterable*
- here is an example of iterating over the elements of a list

```
languages = ['C', 'C++', 'Python']
for lang in languages:
  print(lang)
```

```
C
C++
Python
```

- as with all variables, we are free to choose the name of the variable which stores the current value in each iteration
- but it is good practice to use a name which somehow reminds of what it stores, the name of a language in our case

Loops (2/8)

- it is common to apply a method to the elements iterated over:

```
for word in ['dog','cat','mouse']:
  print(word,len(word))
```

```
dog 3
cat 3
mouse 5
```

- or we can just extract the minimum of the numbers in a list

```
minval_so_far = None  # represent the absence of a value
for i in [3,5,2,1,9,5,-1,4,-2,0]:
  if minval_so_far is None or i < minval_so_far:
    minval_so_far = i
print('minimum is {}'.format(minval_so_far))
```

```
minimum is -2
```

Loops (3/8)

– or determine the average value of a list of floating point numbers

```
count = total = 0
for f in [3.8,5.1,2.3,1.9,9.1,5.2,11.0,4.1]:
  total += f
  count += 1
print('average is {:.2f}'.format(total/count))
```

```
average is 5.31
```

– or collect the characters occurring in a sentence

```
poem = 'Almost nothing was more annoying than having our wasted\
        time wasted on something not worth wasting it on'
char_list = list()
for cc in poem:
  if not (cc in char_list):
    char_list.append(cc)
print('"{}"'.format(''.join(char_list)))
```

```
"Almost nhigwareyvud"
```

Poem is from *Then We Came to the End: A Novel* by Joshua Ferris

Loops (4/8)

- to iterate over a range of numbers, use the `range()`-function:

```
for i in range(5):
  print('{} '.format(i),end='')
print('')
```

```
0 1 2 3 4
```

- `range(n)` delivers an iterator to process the integers starting with 0 and ending with $(n-1) \Rightarrow n$ is excluded from the iteration
- one can also specify the lower bound of the range (0 by default)
- this is used in the following example which counts the sum of numbers between $m = 50$ and $n = 100$

```
m = 50
n = 100
sum = 0
for i in range(m,n+1):
  sum = sum + i
print(sum)
```

```
3825
```

iterator

- ... is an object that can be iterated upon, i.e. you can traverse through all the values.
- in Python, an iterator is an object which implements the iterator protocol, which consist of the methods `__iter__()` and `__next__()`.

Loops (5/8)

- here is an example of a for-loop which enumerates the values in reverse order, beginning with 6 and ending with 1
- for this we use `range` with three parameters:
  - start value 4
  - last value 0 (excluding this)
  - step -1

```
for counter in range (4,0,-1):
  print('counter has value {} and is '.format(counter),end='')
  if counter % 2 == 0:
    print('even')
  else:
    print('odd')
```

```
counter has value 4 and is even
counter has value 3 and is odd
counter has value 2 and is even
counter has value 1 and is odd
```

Loops (6/8)

- in some cases one needs to have loops inside of loops
- this is so in the following example, in which all triples i, j, k of numbers are output, such that $0 \leq i \leq j \leq n$ (for some given n) and $i^2 + j^2 = k^2$
- such triples are called pythagorean numbers
- as we need to compute $\sqrt{i^2 + j^2}$, we import the corresponding function `sqrt` from the math module

- approach: generate all (i, j)-pairs, compute $k = \sqrt{i^2 + j^2}$ and verify that $k^2 = i^2 + j^2$

```
from math import sqrt
for i in range(1,n+1):
  for j in range(i,n+1):
    square_sum = i**2 + j**2
    k = int(sqrt(square_sum))
    if square_sum == k**2:
      print(i, j, k)
```

```
3 4 5
5 12 13
6 8 10
8 15 17
9 12 15
12 16 20
...
```

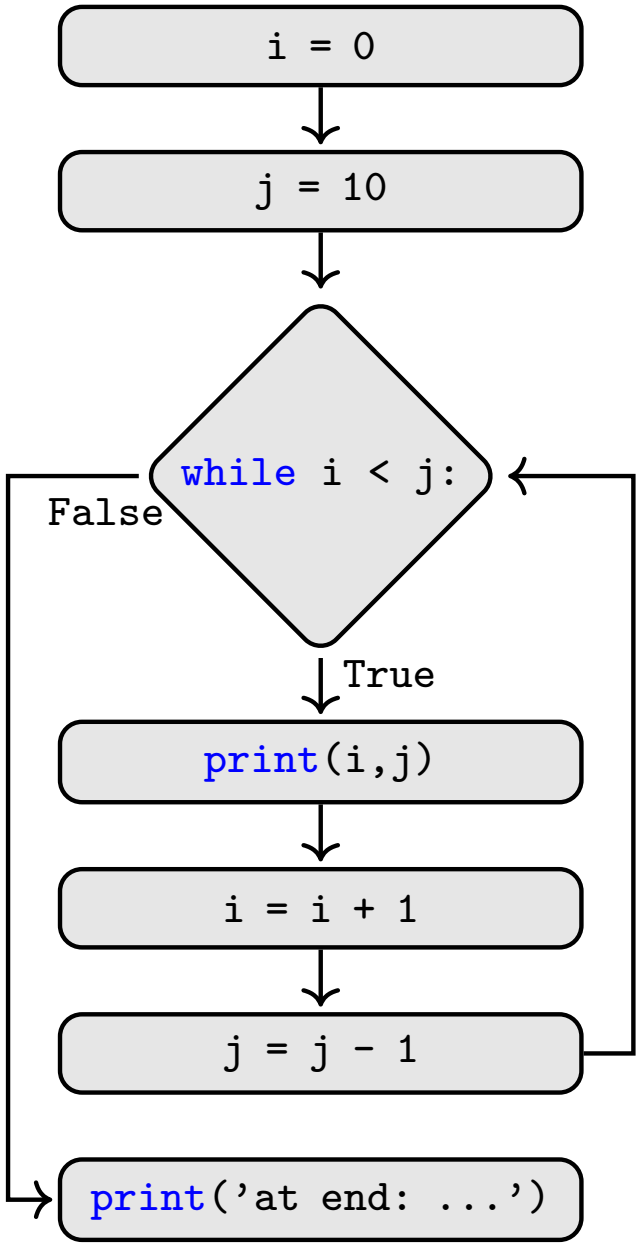
op	meaning
`**`	exponentiation
`int(f)`	integer conversion of floating point number `f`
`sqrt()`	computes $\sqrt{}$

Loops (7/8)

- another kind of loop is the `while`-loop which iterates as long as a given condition is `True`
- the following loop enumerates all pairs $(i, 10 - i)$, $0 \leq i \leq 4$

```
i = 0
j = 10
while i < j:
  print(i,j)
  i = i + 1
  j = j - 1
print('at end: i={},j={}'
       .format(i,j))
```

```
0 10
1 9
2 8
3 7
4 6
at end: i=5,j=5
```



Loops (8/8)

- if a condition can only be tested after some other computations inside the loop, then the `break`-statement to leave the loop is useful

```
import sys
while True:
  c = sys.stdin.read(1)
  if c == '\n':
    break
  print('found {}'.format(c))
```

when typing `abcd` after starting the script, it reports

```
found a
found b
found c
found d
```

- stop the script by typing Ctrl+d

Exception handling (1/5)

- we have already seen how to open files and how to read them

```
stream = open(filename,'r')
```

- we have ignored the very common case that the file cannot be opened
- this may be due to the fact that the filename does not exist
- in this case we get the following error message:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'DNAs.fn'
```

- Or the file exists, but we have no read permission, in which case we get the following error message:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
PermissionError: [Errno 13] Permission denied: 'DNAs.fna'
```

Exception handling (2/5)

- in any case, `stream` is invalid and we need to handle this exception
- at first, we want to provide the user with a less cryptic error message
- second, we have to react to the exception, e.g. by
  - stopping the program with an error exit code (usually done in command line tools)
  - asking the user to again type in the filename (usually done in graphical user interfaces)
- there are other exceptions which may occur when running a program, like
  - a full disk,
  - not enough memory,
  - external hardware not responding,
  - illegal input formats
- in general we want to write robust programs that handle exceptions

Exception handling (3/5)

Syntax of exception handling

- in Python exception handling is specified using the keywords `try` and `except`
- following `try` one specifies code that may lead to an exception
- following `except` one specifies
  - what kind of exception is handled, e.g. `IOError`
  - the name of a variable in which the Python-interpreter will store a possible error message
  - some lines of code reacting on the error
- only when statements following `try` raise an exception of the specified kind, the control is transferred to the error handling code
- if no exception occurs, the error handling code is not executed

Exception handling (4/5)

- Example: open a file and handle the possible exception:

```
try:
  stream = open(fname,'r')  # this statement may throw an exception
except IOError as err:  # exception stores error message in err
  # now comes the code reacting on the exception
  sys.stderr.write('{}: {}\n'.format(sys.argv[0],err))
  exit(1)
```

- call to the method `open` is a system call: Python must communicate to the operating system (for example Linux) to open the file
- important to check for success or failure of anything that can go wrong in a system call
- if something went wrong the operating system will communicate this and in the running python program an exception will be raised
- `except` handles the exception and outputs a hint to the user what went wrong including the message stored in string `err`

Exception handling (5/5)

- note that we do not use `print` for output of the error message, as this writes to the standard output (i.e. the terminal)
- to not mix the error message with valid results (which may have been output to stdout), we use the error-output via `sys.stderr` and the method `write` applied to this
- this also sends output to the terminal, but when (in the shell) we redirect output to a file, we still see an error message on the terminal
- so in case of an exception, our program `readmyfile.py` shows a more readable error message

```
$ readmyfile.rb > out.txt
Can't open file DNAs.fn: [Errno 2] No such file or directory: 'DNAs.fn'
```

- if exception would not be handled, later in the program it would be detected that `stream` is invalid
- the program would terminate and a generic (often difficult to understand) error message would be generated (see above)

Finding motifs (1/7)

- we now have introduced all techniques required for our initial motivating case study: finding regulatory elements
- in bioinformatics regulatory elements are often expressed as motifs
- finding motifs is one of the most common things done in bioinformatics
- motif is a short segment of DNA or protein of special interest, e.g. regulatory elements
- motifs are usually not simple strings: some positions are unspecific $\Rightarrow$ does not matter what base or residue is there
- regular expressions (REs, for short) are convenient to express motifs
- we later provide more details on REs
- we explain the solution for the case study by a Python script which
  - reads sequence data from file
  - concatenates the sequence data into one string to simplify search
  - looks for motifs, the user types in at the keyboard

Finding motifs (2/7)

```
import sys  # for error message
import re   # for regular expressions

# ask the user for the filename of the file containing
# the sequence data, and collect it from the keyboard
filename = input('type filename of the sequence: ')

# remove trailing newline from the read filename
filename = filename.rstrip()

# open the file, or exit: sys.argv[0] is name of program
try:
  stream = open(filename,'r')
except IOError as err:
  sys.stderr.write('{}: {}\n'.format(sys.argv[0], err))
  exit (1)

# read the sequence data from file, store it in sequence_lines
sequence_lines = stream.readlines()

stream.close()
```

Finding motifs (3/7)

```
# concatenate lines into a single string, as it's easier to
# search for a motif in a string than in a list of lines
sequence = ''.join(sequence_lines)

# Remove whitespace from sequence
sequence = re.sub(r'\s','',sequence)

# In a loop, ask user for motif, search for motif, and report
# if it was found. break out of loop if no motif is entered.
while True:
  motif = input('enter motif to search (return => quit): ')
  motif = motif.rstrip()
  if motif == '':
    break
  print('searching motif "{}"'.format(motif))
  m = re.search(r'{}'.format(motif), sequence)
  if m:
    print('I found it!')
  else:
    print('I couldn\'t find it!')
```

Finding motifs (4/7)

- here is an example of the output:

```
type filename of the sequence: NM_021964fragment.pep
enter a motif to search: SVLQ
searching motif "SVLQ"
I found it!
enter a motif to search: jkl
searching motif "jkl"
I couldn't find it.
enter a motif to search:
```

- consider the following lines of the previous program

```
filename = input('type filename of the sequence: ')
filename = filename.rstrip()
```

Finding motifs (5/7)

- `input()` is a method which reads from standard input, e.g. from the keyboard
- in this case: read the file name
- optional argument of `input()`: string describing the kind of information requested from the user
- user types filename and an implicit newline (enter/return key)
- newline is part of the string read from the keyboard
- formatting newline characters must be removed from the input (in this case filename), before processing it
- Python function `rstrip()` removes newlines from the end of a string
- same approach is used in `motif = motif.rstrip()`
- use method `join` to combine all lines into a single string:

```
sequence = ''.join(sequence_lines)
```

Finding motifs (6/7)

- instead of using simple strings like `SVLQ` as motifs we could also use REs specifying more than one sequence
- syntax: `r'<`*regexp*`>'` where *regexp* is some RE
- match one or more strings using special wildcard characters
- above program already uses REs to discard formatting characters from the input file
- for example `\s` matches any whitespace (space, tab, newline, carriage return, formfeed) ⇒ `\s` stands for

  `[ \t\n\r\f]`

  ```
  sequence = re.sub(r'\s','',sequence)
  ```

- replaces any whitespace occurring in `sequence` by the empty string ⇒ these characters are effectively deleted

Finding motifs (7/7)

- now consider the line:

```
m = re.search(r'{}'.format(motif), sequence)
```

- this replaces `{}` by the value of the variable `motif` and thus effectively generates a RE from this value: this is called interpolation
- `search` is the method from the `re`-module to search a RE in the string given as second argument

re.search(pattern,string) ⇒ match object

Scan through string looking for a match to the pattern, returning a match object, or `None` if no match was found.

- we will later see how to access a match object
- note that `re.search(r'motif',sequence)` would just search for the string `motif`, but not for the value of the variable `motif`

Overview of Python-methods used until now (1/2)

method	meaning
`print`	print values to terminal with trailing \n
`write`	print error messages without trailing \n
`+`	concatenation of strings
`+`	addition of integers
`sub`	substitution of strings matching a RE
`reversed`	reverse order of characters in string
`translate`	translate characters according to dictionary
`maketrans`	create dictionary specifying translation
`open`	open file, return stream
`close`	close file, delete stream
`readline`	read next line
`readlines`	read all lines, return list

Overview of Python-methods used until now (2/2)

method	meaning
`join`	concatenate strings with separator
`pop`	remove element from end of list
`pop(0)`	remove element from beginning of list
`append`	add to end of list
`[i:j]`	get slice of list from index `i` to index `j-1`
`insert(i,e)`	add `e` after `i`th element of list
`len`	determine length of string or list
`rstrip`	remove trailing newlines
`search`	search for RE

Regular expressions: syntax and semantics (1/6)

- we have seen that REs can be used to specify what formatting characters are removed from the input
- we have also seen that REs can be used to specify motifs (although we have not done it yet)
- we now go into detail about REs
- simplest RE: just a string of characters, e.g. `AQQK`
- RE to search for an `A` followed by `D` or `S`, followed by `V`: `A[DS]V`
- RE to search for `K`, followed by `N`, followed by zero or more `D`'s, and two or more `E`'s: `KND*E{2,}`
- RE to search for two `E`'s followed by anything, followed by another two `E`'s: `EE.*EE`
- RE to search for a word, i.e. consecutive sequence of characters excluding separators or white spaces: `\w+`
- RE to search for three characters in a row: `...`

Regular expressions: syntax and semantics (2/6)

`[abc]`	a single character of `a`, `b`, or `c`	`a|b`	`a` or `b`
`[^abc]`	a single character except `a`, `b`, or `c`	`a?`	zero or one of `a`
`[a-z]`	any single character in the range `a-z`	`a*`	zero or more of `a`
`[a-zA-Z]`	any single character in the range `a-z` or `A-Z`	`a+`	one or more of `a`
`^`	start of line, after newline	`a{3}`	exactly 3 of `a`
`$`	end of line, before newline	`a{3,}`	3 or more of `a`
`\A`	start of string	`a{,6}`	up to 6 of `a`
`\z`	end of string	`a{3,6}`	between 3 and 6 of `a`
`.`	any single character except for `\n`	`\(`	the symbol `(`
`\w`	any character of a word (letter, digit, `_`)	`\n`	a newline
`\W`	any non-word character	`\t`	a tabulator
`\s`	any whitespace character	`\$`	the character `$`
`\S`	any non-whitespace character	`\^`	the character `^`
`\d`	any digit	`\*`	the character `*`
`\D`	any non-digit	`(...)`	capture everything enclosed, define group
`\b`	any word boundary		

- `https://pythex.org/` provides a RE editor
- this tries to match a RE in a given string as you write the RE

Regular expressions: syntax and semantics (3/6)

task	RE	string	matches
match a codon	`[acgt]{3}`	`aaggacta`	aag gac
match a word	`\w+`	`a long tale`	a long tale
match `ac` at start of string	`^ac`	`acgaccac`	ac gaccac
match `ac` at end of string	`ac$`	`acgaccac`	acgacc ac
match `Car` or `car`	`[Cc]ar`	`care about a Car`	car e about a Car
match 2-digit number not in context of letter or digit	`\b\d{2}\b`	`won: 74, lost: 235`	won: 74 , lost: 235
match 3 or 4-letter words beginning with a and ending with b	`\ba\w\w?b\b`	`accb agb taab`	accb agb taab

Regular expressions: syntax and semantics (4/6)

- we now give another example on applications of REs and show how specify and access groups in them

```
for date in ['2014-12-13','2016-01-02']:
  # extract numbers from a date string YYYY-MM-DD
  m = re.search(r'(\d{4})-(\d{2})-(\d{2})', date)
  if m:
    year = m.group(1)
    month = m.group(2)
    day = m.group(3)
    print('{}.{}.{}'.format(day, month, year))
```

```
$ ./datematch.py
13.12.2014
02.01.2016
```

- the above RE uses parentheses enclosing parts of the RE
- each pair of parentheses denotes a group and if the RE matches in a string, we can access the substrings matching to partial RE
- access is via the method `group` applied to the match object `m` delivered by the `search`-method
- argument of `group` is number of group in order from left to right

Regular expressions: syntax and semantics (5/6)

- here is another example which extracts parts of a name from a string and prints first name before last name

```
if len(sys.argv) == 2:
  s = sys.argv[1]
else:
  sys.stderr.write('Usage: {} <string>\n'.format(sys.argv[0]))
  exit(1)

m = re.search(r'name is (\w+), (\w+)', s)
if m:
  print('{} {}'.format(m.group(2), m.group(1)))
```

```
$ ./first-last-name.py 'name is Doe, John'
John Doe
```

Regular expressions: syntax and semantics (6/6)

- the following uses a more complicated RE to match an atom name, the name of the group it belongs to, and the corresponding atomic weight, all embedded in a string with blanks
- `search`-method delivers match-object and each string matching the *i*th-group is accessed by `group(i)`
- result is printed with with columns aligned

```
atoms = ['Hydrogen gas 1.00794',
         'Lithium  alkaline metal 6.941',
         'Beryllium   alkaline earth metal9.012182']
for atom in atoms:
  m = re.search(r'(\w+)\s+([\w\s]+)\s?(\d+\.\d+)', atom)
  if m:
    print('{:<9s} {:<20s} {:>.3f}'
            .format(m.group(1),m.group(2),float(m.group(3))))
```

```
Hydrogen  gas                  1.008
Lithium   alkaline metal       6.941
Beryllium alkaline earth metal 9.012
```

methods which use REs

`sub` `search`

REs for parsing complex numbers (1/4)

- in $\mathbb{R}$, there is no solution to the equation $x^2 = -1$
- for this reason one uses complex numbers, based on the imaginary unit i standing for $\sqrt{-1}$
- by taking multiples of this imaginary unit, we can create infinitely many more new numbers, such as $3i$ and $-12.9i$.
- their general form is bi for a real number b
- adding real numbers to these pure imaginary numbers creates even more numbers like $7.2i + 2$ or $3 - 5.1i$
- while these are not pure imaginary numbers, they are not real numbers either.
- instead, they belong to a set of numbers called complex numbers.

intro to complex numbers adapted from `https://www.khanacademy.org/math`

REs for parsing complex numbers (2/4)

- a complex number is any number that can be written as $a + bi$ where i is the imaginary unit and a and b are real numbers

$$\underset{\substack{\uparrow \\ \text{real} \\ \text{part}}}{a} + \underset{\substack{\uparrow \\ \text{imaginary} \\ \text{part}}}{b}\, i$$

- sometimes the notation for complex numbers is not unified, so that $-2 + 7i$ is written as $7i - 2$ or $4 + (-3)i$ is written as $4 - 3i$
- our task is to parse from a string with a complex number the real part a and the imaginary part b and to print the number in unified format $a + bi$.
- to simplify the parsing we restrict to the case that a and b are integers

REs for parsing complex numbers (3/4)

- the idea of our approach is to first identify in the given string the imaginary part, which comes before an occurrence of the letter *i*
- here is an appropriate RE: `(-?\s*\d+)\s*i`
- this matches (in the given order)
  - an optional minus sign
  - any number of white spaces
  - one or more digits
  - any number of white spaces
  - the letter *i*
- the brackets are used for defining a group consisting of the imaginary part
- if the match to the RE is successful, we can extract the imaginary part as integer, enclose it in brackets if its negative and then delete the imaginary part from the original string
- the remaining string must then be the real part
- here is the complete program with some examples input strings

REs for parsing complex numbers (4/4)

```
im_re = '(-?\s*\d+)\s*i'
for cs in ['7i-2','4-3i','9 i','-2',\
           '21-14i','-17 i+1']:
  m = re.search(r'{}'.format(im_re),cs)
  if not m:
    b = 0
    a = int(cs)
  else:
    b = int(m.group(1))
    if b < 0:
      b = '({})'.format(b)
    rest = re.sub(r'{}'.format(im_re),'',cs)
    if rest == '':
      a = 0
    else:
      a = int(rest)
  print('{:10s} {}+{}i'.format(cs,a,b))
```

```
7i-2       -2+7i
4-3i       4+(-3)i
9 i        0+9i
-2         -2+0i
21-14i     21+(-14)i
-17 i+1    1+(-17)i
```

- as we use the same regular expression in two contexts (search, sub), we store it in a string variable `im_re`

- first `if`-statement handles case that imaginary part is 0 as in −2
- imaginary part not negative ⇒ `b` is integer variable
- imaginary part is negative `b` is string variable with brackets

What is wrong here?

```
import re

re_list = ['[ab][cd]','[01][23]']
for re in re_list:
  for s in ['ac','bd','02','12']:
    m = re.search(r'{}'.format(re),s)
    if not (m is None):
      print('{} matches {}'.format(re,s))
```

```
Traceback (most recent call last):
  File "./parse-strings.py", line 8, in <module>
    m = re.search(r'{}'.format(re),s)
AttributeError: 'str' object has no attribute 'search'
```

What is wrong here?

```
import re

re_list = ['[ab][cd]','[01][23]']
for re in re_list:
  for s in ['ac','bd','02','12']:
    m = re.search(r'{}'.format(re),s)
    if not (m is None):
      print('{} matches {}'.format(re,s))
```

```
Traceback (most recent call last):
  File "./parse-strings.py", line 8, in <module>
    m = re.search(r'{}'.format(re),s)
AttributeError: 'str' object has no attribute 'search'
```

⇒ be careful to not introduce identifiers that are identical with existing module/class names

Different ways of splitting a complex number (1/3)

- we continue with parsing complex numbers
- this time we suppose that $a + bi$ is written as a string (a, b), possibly with spaces before or after the brackets
- of course, we can perform the parsing using a regular expression, in which we capture the real and imaginary part using brackets
- the real part is everything after and before any number of white spaces, but not a comma
- the imaginary part follows a comma, and any number of white spaces, and does not contain a closing bracket
- the brackets which are part of the notation have to be preceded by \ to state that we do not mean the pair ( ) of meta symbols

```
cnp = ' (-3.1, 4) ' # complex number as pair
m = re.search(r'\(\s*([^,]+)\s*,\s*([^\)]+)\s*\)', cnp)
if m:
  a, b = float(m.group(1)), float(m.group(2))
  print('{}+{}i'.format(a,b))
```

Different ways of splitting a complex number (2/3)

- as the notation has a unique separator (the comma) between the real and imaginary part we can also use the method `split`
- this splits a string at a given separator and delivers a list of strings
- the length of the list is 1 longer than the number of separators

```
m = cnp.strip()[1:-1].split(',') # m is list of length 2
a, b = float(m[0]), float(m[1])
print('{}+{}i'.format(a,b))
```

- as the string to parse may be enclosed in white spaces, we first apply the method `strip()` to remove trailing and leading white spaces
- after applying `strip()` we obtain a string in which the first character is `'('` and the last character is `')'`.
- to get rid of these we use the slice operator for strings: for a string `s`, `s[i:j]` is a new string, the slice from index `i` to index `j-1` in `s`
- we use `i=1` (to start with the second character) and `j=-1` (to end with the last but one character)
- index `-1` is convenient notation for the last index of string

Different ways of splitting a complex number (3/3)

- the simplest extraction method is based on the fact that the string we parse is a valid expression for a pair in Python-syntax
- we can exploit this using the parser of the Python-interpreter
- this is done by applying the method `eval` to the given string
- this delivers a pair of two values and we can assign these two values to corresponding variables

```
a, b = eval(cnp)
print('{}+{}i'.format(a,b))
```

- `eval` is a very general method, which could be applied to any string consisting of a valid expression in python syntax

`eval`(source,vars=None) ⇒ value

Evaluate the given source for the optional variable binding, which is given as a dictionary. The source may be a string representing a Python expression.

Overview of method on strings

synopsis: methods on strings	
`len(s)`	deliver number of elements in string `s`
`s[i:j]`	get slice of string `s` from index `i` to index `j-1`
`s.split(sep)`	split string `s` at the given separator into list of strings
`s.rstrip()`	remove trailing white spaces from string
`s.lstrip()`	remove leading white spaces from string
`s.strip()`	remove leading and trailing white spaces from string
`'a{}c'.format(s)`	substitute string `s` for place holder `{}`
`str.maketrans(x,y)`	create translation table mapping elements in `x` to elements in `y`
`s.translate(t)`	apply translation table `t` to string `s`

synopsis: methods on regular (ordered by importance)

`re.sub(r'...',rep,s)`	replace all substrings in `s` matching RE by `rep`
`re.search(r'...',s)`	search for occurrences of RE in string `s`, return match object
`re.findall(r'...',s)`	Return all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order found. If one or more groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group. Empty matches are included in the result.
`re.finditer(r'...',s)`	Return an iterator yielding match objects over all non-overlapping matches for the RE in string. The string is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result.

Histograms: counting occurrences of values (1/5)

- it is a common task to count the number of occurrences of elements in some collection of data, e.g. the number of occurrences of
  - characters in a sequence
  - words in a text
  - integers in a table
- in the simplest case we have to count just one character in a string and for this we can use the method `count`

`s.count(sub)` $\Rightarrow$ int

Return the number of non-overlapping occurrences of substring `sub` in string `s`

Example:

```
'abracadabra'.count('a')    ⇒   5
'abracadabra'.count('ab')   ⇒   2
```

Histograms: counting occurrences of values (2/5)

- the following program reads a DNA string from the command line and counts the number of occurrences of base g

```
if len(sys.argv) != 2:
  sys.stderr.write(
        'Usage: {} DNA\n'
        .format(sys.argv[0]))
  exit(1)

s = sys.argv[1].lower()
print('{} contains {} g\'s'
        .format(s,
                s.count('g')))
```

```
$ ./countg.py
Usage: ./countg.py DNA
$ ./countg.py AGGTGGAA
aggtggaa contains 4 g's
```

- `sys.argv[0]` is the program name
- the first argument is at index 1

⇒ `sys.argv` must be of length 2

- if not, generate error message on `sys.stderr`, the error output stream
- error message is a `Usage`-line telling how to correctly use the program
- such `Usage`-lines are mandatory in all solutions of future exercises
- provides a basic way of documenting the program's interface

- in the general case we have to count more than one element

Histograms: counting occurrences of values (3/5)

- depending on the number of the possible elements and their type (e.g. characters, strings, numbers) we need different ways to store counts
- we start with a simple example counting bases in a DNA sequence
- as we only have four bases and are interested in the number of characters not denoting a base, we only need 5 variables for counting

```
if len(sys.argv) != 2:
  sys.stderr.write('Usage: {} <DNA sequence file>\n'
                   .format(sys.argv[0]))
  exit(1)

fname = sys.argv[1]
try:
  stream = open(fname, 'r')
except IOError as err:
  sys.stderr.write('{}: {}\n'.format(sys.argv[0],err))
  exit(1)

dna = stream.read()  # return string of chars from stream
stream.close()
dna = re.sub(r'\s', '', dna)    # remove whitespace
```

Histograms: counting occurrences of values (4/5)

```
countA = countC = countG = countT = errs = 0

# look at each base in turn, and increment
# appropriate count.
for base in dna:
  if base == 'A':
    countA += 1
  elif base == 'C':
    countC += 1
  elif base == 'G':
    countG += 1
  elif base == 'T':
    countT += 1
  else:
    sys.stderr.write('unknown base {}\n'
                     .format(base))
    errs += 1

print('A\t{}\nC\t{}\nG\t{}\nT\t{}\nerrs\t{}'
      .format(countA,countC,countG,countT,
              errs))
```

```
$ cat DNAfile
ATTCGCGCTCTCC
TCGCGCTCTCCTT
CGGGGGCTCTC
$ ./distrib.py DNAfile
A       1
C       16
G       9
T       11
errs    0
```

Histograms: counting occurrences of values (5/5)

- the counter values are usually only the first step
- one often wants to see relative frequencies and the G/C-content
- these values are output by the following code

```
count_all = countA + countC + countG + countT
print('[ACGT]\t{}'.format(count_all))
print('relative frequencies')
print('A\t{:>6.2f}%'.format(100.0 * countA/count_all))
print('C\t{:>6.2f}%'.format(100.0 * countC/count_all))
print('G\t{:>6.2f}%'.format(100.0 * countG/count_all))
print('T\t{:>6.2f}%'.format(100.0 * countT/count_all))
print('G/C\t{:>6.2f}%'
       .format(100.0 * (countG+countC)/count_all))
```

output for previous file named DNAfile with counts 1, 16, 9, 11 for A, C, G, T

```
[ACGT]    37
relative frequencies
A         2.70%
C        43.24%
G        24.32%
T        29.73%
G/C      67.57%
```

Counting bases using REs (1/3)

- REs and the method `findall` provide another convenient approach for counting bases
- again we read the file content into a string
- not necessary to remove `\n`, since we are looking for single character patterns, this time accepting notation in lower and upper case

```
stream = open('DNAfile', 'r')
dna = stream.read()
stream.close()

countA = len(re.findall(r'a|A', dna)) # len for number of matches
countC = len(re.findall(r'c|C', dna))
countG = len(re.findall(r'g|G', dna))
countT = len(re.findall(r't|T', dna))
errs = len(re.findall(r'[^acgt|ACGT]',dna)) # all which is not base

print('A\t{}\nC\t{}\nG\t{}\nT\t{}\nerrs\t{}'
      .format(countA,countC,countG,countT,
              errs))
```

Counting bases using REs (2/3)

– for each base and the non-base characters there is a separate iteration performed by `findall` to count these

re.findall(pattern,string) ⇒ list

Return a list of all non-overlapping matches of the pattern in the string. If one or more capturing groups, enclosed in `()` are present in pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group.

Example:

```
s = 'cruel world'
re.findall(r'\w+',s)        ⇒   ['cruel', 'world']
re.findall(r'...',s)        ⇒   ['cru', 'el ', 'wor']
re.findall(r'(..)(..)',s)   ⇒   [['cr', 'ue'], ['l ', 'wo']]
```

Counting bases using REs (3/3)

- until now we have always generated output on stdout (the terminal)
- we now want to directly store the output in a file

```
outputfilename = 'countbase'

# write results to file, s_out is the output stream
try:
  s_out = open(outputfilename,'w')
except IOError as err:
  sys.stderr.write('{}: {}\n'.format(sys.argv[0],err))
  exit(1)

s_out.write('A={} C={} G={} T={}\n'
            .format(countA, countC, countG, countT))
s_out.close()
```

- to open a file for writing, we use the second argument `'w'` in the `open`-method
- instead of `sys.stderr` for error messages (as before) we use the output stream `s_out` to which the `write`-method writes its output

Counting many characters: the ASCII-table (1/1)

dec.	char	dec.	char	dec.	char	dec.	char
0	NUL	32	SP	64	@	96	‘
1	SOH	33	!	**65**	**A**	**97**	**a**
2	STX	34	”	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	**$**	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	’	71	G	103	g
8	BS	40	(	72	H	104	h
9	TAB	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	**55**	**7**	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	**90**	**Z**	**122**	**z**
27	ESC	59	;	91	[	123	{
28	FS	60	¡	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	¿	94	^	126	”
31	US	63	?	95	_	127	DEL

- `ord` for conversion of character to integer:

  `ord('a')` ⇒ 97

  `ord('A')` ⇒ 65

  `ord('0')` ⇒ 48

- `ord('a')-ord('A')` ⇒ 32 is the distance between lower and upper characters

- `chr` for conversion of integer to character:

  `chr(97)` ⇒ `'a'`

  `chr(65)` ⇒ `'A'`

  `chr(48)` ⇒ `'0'`

Counting characters (1/6)

- once the number of items to count becomes larger (not just 4 as above), we need a more flexible way of introducing and using counters
- suppose we want to determine the number of characters in an arbitrary file, including text and binary files
- the possible number of characters is $256 = 2^8$, but we do not want to introduce this many different variable names
- use a list `counters` of length 256, initialized to 0: this is introduced by the statement `counters = [0] * 256`
- here the operator `*` serves to express building a list of 256 copies of an element 0
- each entry corresponds to one of the possible 256 different characters we need to count
- *i*th entry counts occurrences of character `x` satisfying `ord(x)==i`

Counting characters (2/6)

```
try:
  stream = open(__file__, 'r') # open this source file
except IOError as err:
  sys.stderr.write('{}: {}\n'.format(sys.argv[0],err))
  exit(1)

counters = [0] * 256  # list with 256 entries, init with 0
for line in stream:    # iterate over the lines in stream
  for c in line:       # iterate over the characters in line
    c_code = ord(c)          # convert character to code
    counters[c_code] += 1    # use code as index and increment
stream.close()
for c_code, count in enumerate(counters):  # iterate over counters
  if count > 0:                            # output positive counts
    print('{}\t{}'.format(chr(c_code),count)) # convert code->char
```

– show 4 most occurring characters

```
$ ./selfcountchar.py | sort -k 2 -n | tail -n 4
```

i	90
r	95
e	108
t	124

Counting characters (3/6)

Structure of previous program: three phases

1. open input stream
2. read line by line and in each line character by character, accumulating character counts in list `counters`
3. output characters and their counts in tab-separated output lines

- step 2 could be simplified by using `stream.read()` to read all characters from stream at once
- but this may require space on the order of the size of the file (which can be very large)
- so the line by line method is more space efficient and thus the preferred one

Counting characters (4/6)

Choice of data structure:

- above scheme is used in a similar way by many programs
- but of course there are differences about what to accumulate
- a central decision concerns the kind of data structure in which to accumulate the counts
- in our case: a list of counters initialized to 0
- as a consequence, we had to convert a character to a list index and vice versa, using the functions `ord` and `chr`
- these functions are actually not necessary if we replace the list by another data structure, a dictionary
- a dictionary has an internal conversion method from keys to locations in memory, where a key/value pair is stored
- this is illustrated in the following program

Counting characters (5/6)

```
line = 'abracadabra'
count_dict = dict()     # empty dictionary
for c in line:          # iterate over charactres in string
  if not (c in count_dict): # check if c is not already in dict.
    count_dict[c] = 0   # for first occurence init count to 0
  count_dict[c] += 1    # increment entry for character c in dict
for c, count in count_dict.items():# iterate over key/value pairs
  print('{}\t{}'.format(c,count))
```

- we simplified the input to keep it short
- result is as before, except for the order of the counts output

Live programming

- this program will be the first we study in a live programming session
- use `www.pythontutor.com/live.html`
- open `https://goo.gl/XgJJYn` to repeat this live session without typing the code again

Counting characters (6/6)

Live Programming Mode - Pyt

Not Secure | pythontutor.com/live.html#mode=edit

This is the live programming mode (video intro), which continually runs and visualizes your code as you type. It's **highly experimental** and does not yet support all languages and features of the regular Python Tutor visualizer.

Support our research and keep this tool free by **filling out this short user survey**.

Get live help! (NEW!)

Start private chat session

How do I use this?

These Python Tutor users are asking for help right now. Please volunteer to help!

- user_49a from Recife, Brazil needs help with Python3 - 2 people chatting - **click to help** (active a few seconds ago, requested 3 hours ago)
- user_72a from Hanau, Germany needs help with Python3 - click to help (IDLE: last active 3 minutes ago, requested 2 hours ago)

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1  count_dict = dict()
2  line = "aasfhsdfasf"
3  for c in line:
4      if not (c in count_dict):
5          count_dict[c] = 0
6      count_dict[c] += 1
7  for c, count in count_dict.items():
8      print('{}\t{}'.format(c,count))
9
```

Print output (drag lower right corner to resize)

```
a       3
s       3
f       3
h       1
d       1
```

Frames

Global frame

count_dict	
line	"aasfhsdfasf"
c	"d"
count	1

Objects

dict

"a"	3
"s"	3
"f"	3
"h"	1
"d"	1

line that has just executed

next line to execute

<< First | < Back | Step 52 of 52 | Forward > | Last >>

Dictionaries (1/2)

- we have seen how to use a dictionary to count characters
- we now go into more detail about dictionaries before we consider an application which counts *words* using dictionaries
- dictionaries provide a data structures for associating a value with a key
- one can easily add a new key/value pair and update or lookup the value for a given key
- one can even delete key/values pairs from a dictionary
- dictionaries are similar to lists of key/value-pairs (and often visualized in this way, see live session)
- but in dictionaries operations on key/value pairs are much faster (i.e. require amortized constant time) compared to the corresponding operations on lists (which require linear time)
- in dictionaries, keys must be hashable objects

Dictionaries (2/2)

- an object is hashable[1] if it has a hash value which
  - never changes during its lifetime (it needs a `__hash__()` method), and
  - can be compared to other objects (it needs an `__eq__()` method)
- hashable objects which compare equal must have the same hash value
- this implies that the hash value only depends on the object but not its context
- examples of types with hashable objects:
  - immutable built-in types, such as strings, numbers and tuples (pairs, triples, ...)
  - functions
  - used defined classes
- lists are not hashable

[1] `https://docs.python.org/3/glossary.html`

Dictionaries

- in a dictionary, introduced by `english2german = dict()`, we can add a key/value pair by:

```
english2german['Iron'] = 'Eisen'
```

- lookup is done as follows:

```
germanword = english2german['Iron']
```

- string `Iron` is key; returned value is associated with this key
- a dictionary always introduces a finite mapping from keys to values (where we use a shorthand notation for key/value pairs):

```
english2german = {
  'Hydrogen' : 'Wasserstoff',
  'Carbon'   : 'Kohlenstoff',
  'Sulfur'   : 'Schwefel'
  }
```

- key lists and value lists are extracted with corresponding methods:

```
translatedwords = list(english2german.keys())
translations = list(english2german.values())
```

Using dictionaries for counting words

- in the 'counting words problem' we use the method `findall` with RE `\w+` to extract the words from the given text (a string)
- each word adds 1 to the corresponding entry in the initially empty dictionary `countwords`
- output of all dict-entries is tab-separated

```
text = ('now is the time for all good men '
        'to come to the aid of their party')

countwords = dict() # emtpy dictionary
for w in re.findall(r'\w+',text):
  if not (w in countwords): # new word?
    countwords[w] = 0  # first initialize
  countwords[w] += 1   # increment count

for key, value in countwords.items():
  print('{}\t{}'.format(key,value))
```

- `items()`-method gives access to key/value pairs

for	1
now	1
is	1
all	1
good	1
of	1
men	1
their	1
come	1
the	2
aid	1
party	1
time	1
to	2

order of values in lines is implementation dependent

https://goo.gl/Uk9Bui

Joining list of atom names (1/4)

- as a further application of dictionaries, consider the problem of joining two files with abbreviations of atoms and their full names
- one file contains the full names in english and one contains the full names in german:

```
$ head -n 5  elementlist*.tsv
==> elementlist_de.tsv <==
H       Wasserstoff
He      Helium
Li      Lithium
Be      Beryllium
B       Bor
==> elementlist.tsv <==
He      Helium
Be      Beryllium
H       Hydrogen
B       Boron
Li      Lithium
```

- the two columns of both files are separated by a tabulator, as suggested by the suffix `.tsv`
- the lines are not necessarily ordered
- the goal is to write a Python-script `atom_join.py` to merge these two files and output three columns:

```
$ atom_join.py elementlist*.tsv
Ac      Actinium        Actinium
Ag      Silver  Silber
Al      Aluminum        Aluminium
Am      Americium       Americium
```

Joining list of atom names (2/4)

- our approach is to read the two files, parse them line by line and create a dictionary with the atom abbreviation as key and the full name as value
- so we first open the streams and as this will be done for the first and second command line parameter we wrap it into a loop

```
if len(sys.argv) != 3:
  sys.stderr.write('Usage: {} <atomfile1> <atomfile2>\n'
                   .format(sys.argv[0]))
  exit(1)

stream_list = list()
for arg in sys.argv[1:]:  # iterate over all elements except 1st
  try:
    stream = open(arg)
  except IOError as err:
    sys.stderr.write('{}: cannot open file {}\n'
                     .format(sys.argv[0],arg))
    exit(1)
  stream_list.append(stream)
```

Joining list of atom names (3/4)

- we now have the list of streams and iterate over each of the streams, line by line
- each line (after removing trailing white spaces) is split on `\t`
- check if the resulting list has exactly two elements
- if so, then use the first as key and the second as value of a dictionary

```
dict_list = list()
for stream in stream_list:
  atom2name = dict()  # key is abbrev, value is name
  for line in stream:
    abbrev_name = line.rstrip().split('\t')
    if len(abbrev_name) != 2:
      sys.stderr.write('{}: expect two columns in line {}'
                       .format(sys.argv[0],line.rstrip()))
      exit(1)
    atom2name[abbrev_name[0]] = abbrev_name[1]
  stream.close()
  dict_list.append(atom2name)
```

- result is list of two dictionaries, corresponding to the two files read

Joining list of atom names (4/4)

- in the final loop we iterate over the key/value pairs in the first dictionary
- each such key/value-pair is an abbreviation associated with a name
- we check if the key is present in the second dictionary, and if so, output the three columns

```
atom2name0 = dict_list[0]
atom2name1 = dict_list[1]
for abbrev, name in atom2name0.items():
  if abbrev in atom2name1:
    print('{}\t{}\t{}'.format(abbrev,name,atom2name1[abbrev]))
```

- this gives output, like the one shown on frame 155
- as mentioned previously, there is no specific order of lines output
- this is because, the order depends on where in memory the dictionary stores the key/value-pairs and this is implementation dependent

Key points on dictionaries

- a dictionary is used for associating keys with values
- in a given dictionary, any key appearing is associated with a unique value, so `{'a' : 1, 'a': 3}` is not a valid dictionary
- the same value can be associated with different keys, like in `{'a' : 1, 'b': 1}`
- a dictionary `d` is introduced by `d = {}` or (preferred) `d = dict()`
- we can add a pair `(k,v)` of key `k` and value `v` by `d[k] = v`, where `d` was introduced as described above
- `list(d.keys())` is the list of keys of dictionary `d`
- `list(d.values())` is the list of values of dictionary `d`
- `list(d.items())` is the list of key/value pairs of dictionary `d`
- the elements in these lists are in no defined order
- in a context where we want to iterate over the list of keys, or the list of values, or the list of key/value-pairs, we omit `list()`
- `d[k]` is the value associated with key `k` in dictionary `d`
- if there is no key `k` in `d`, `d[k]` raises an exception of type `KeyError`
- to check if there is a key `k` in dictionary `d`, use the expression `k in d`

What you have learned so far (hopefully) (1/1)

- you know how to write down values of the most important build-in classes: integers, floats, strings, lists, dictionaries
- you know several methods to manipulate objects of these classes
- you know how to open files and read them all at once or line by line
- you know how to control the flow of a program by using control structures: `if`, `while`, `for`
- you know how to search for patterns in strings using regular expressions

Important features of Python still missing

- define your own functions
- define your own classes

- will be considered in this and the following lecture

Function definitions (1/14)

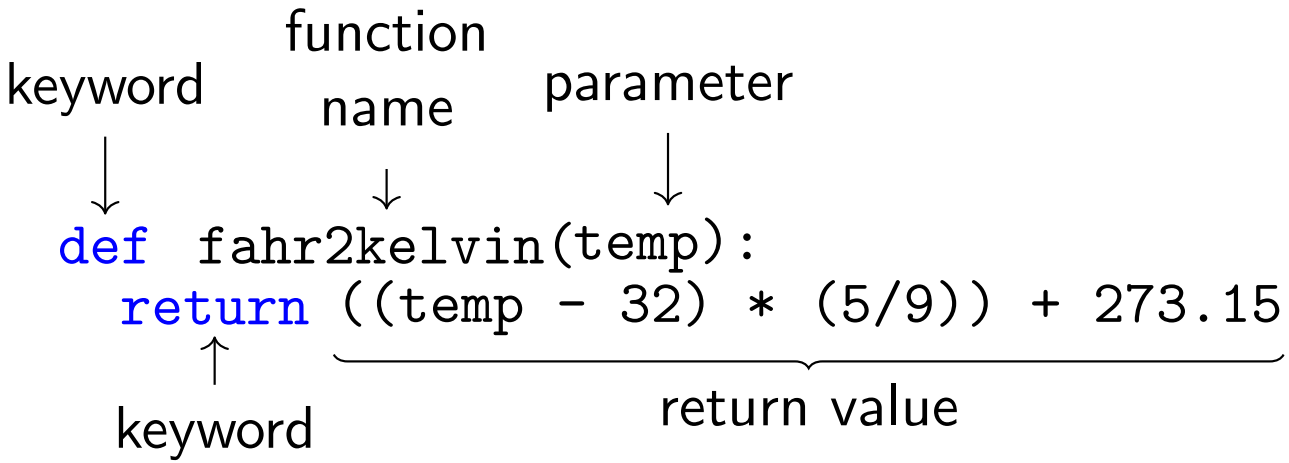
- functions (also called methods in Python) are an important part of almost all Python scripts
- a function allows to group a sequence of statements into one unit, give this a name and reuse the statements by calling the function
- functions can be applied to object of specific classes (via dot-operator)
- functions can take parameters which are substituted when calling the function

⇒ provides a very powerful mechanism for abstraction: combine sequence of statements with a similar effect into a single function

- advantage of using functions:
  - part of code becomes reusable (no paste and copy): faster to write and more reliable code
  - easier to test: functions can be tested separately
  - helps to organize and to abstract your ideas
  - improves readability

Function definitions (2/14)

- we already have seen several functions/methods and their application
- all these functions are applied to different kinds of values, like numbers and strings
- many of them have different kinds of arguments
- function declaration starts with keyword `def` followed by the function name and an optional comma separated list of parameters
- function may return values in a `return` statement



- inside the function, we can use parameters like we use variables

Function definitions (3/14)

- of course, we can call our own function like any other function:

```
print('freezing point of water in K: {}'.format(fahr2kelv(32)))
print('boiling point of water in K: {}'.format(fahr2kelv(212)))
```

```
freezing point of water in K: 273.15
boiling point of water in K: 373.15
```

- we now want to convert Kelvin to Celsius

```
def kelv2cels(temp):
  return temp - 273.15

print('absolute zero in C: {}'.format(kelv2cels(0.0)))
```

```
absolute zero in Celsius: -273.15
```

- to convert Fahrenheit to Celsius, we can compose the two previous functions and derive a new function

Function definitions (4/14)

```
def fahr2cels(temp_f):
  temp_k = fahr2kelv(temp_f)
  return kelv2cels(temp_k)

print('freezing point of water in C: {}'.format(fahr2cels(32.0)))
```

```
freezing point of water in Celsius: 0.0
```

example from http://swcarpentry.github.io/python-novice-inflammation/06-func/

Function definitions (5/14)

- here is another example which wraps the code on word counting (see frame 155) into two functions, called directly after the definitions (result is as before)

```
def distribution_dict_new(t):
  countwords = dict()
  for w in re.findall(r'\w+',t):
    if not (w in countwords):
      countwords[w] = 0
    countwords[w] += 1
  return countwords
```

```
def distribution_dict_show(dist):
  for key, value in dist.items():
    print('{}\t{}'
          .format(key,value))

# now the functions declared
# above are executed
cw = distribution_dict_new(text)
distribution_dict_show(cw)
```

Function definitions (6/14)

- note that we have applied some renaming, like `t` instead of `text` and `dist` instead of `countwords`
- this is no problem as we are free in choosing the names of parameters
- we also use the notion `distribution` as part of the function names to reflect that they operate on distributions in some mathematical sense, associating counts with keys (words in our case)
- first function creates the distribution, hence we use the suffix `new`
- second function shows the distribution, hence we use the suffix `show`
- we intentionally split the previous code into two functions to reflect the two phases of the program
- this also improves reusability of the code, as we could show the distribution created by the first function in a different way as before, say as frequency distribution
- note that the declaration of the function only defines what the function could do; only when it is called (last two lines), it is actually executed

Function definitions (7/14)

```
def distribution_dict_freq_show(dist):
  sum_count = 0   # sum of counts
  for key, count in dist.items():
    sum_count += count  # add count
  for key, count in dist.items():
    print('{}\t{:.2f}%'
          .format(key,100.0 * count/sum_count))
```

- as we want the relative counts, we first determine the sum of all counts in an iteration over the items in the dictionary
- in a second iteration we print relative frequencies with precision 2 by dividing each count by the sum
- multiplication by 100.0 turns the ratio into a percentage value
- while the function is used for output of distribution for words, it can be used to output any kind of distribution stored in a dictionary

the 5 lines of output with max. percentage, if the input is the source file:

```
t          2.80%
for        3.27%
key        3.27%
in         4.21%
value      4.21%
w          4.67%
countwords 5.14%
```

Function definitions (8/14)

- wrap the code on character distributions (see frame 147) into reusable functions, now only showing the function displaying rel. frequencies

```
def distribution_list_new(s):
  counters = [0] * 256
  for line in s:
    for c in line:
      counters[ord(c)] += 1
  return counters
```

- main difference to previous functions is how counts are accumulated
- `show` functions only differ in how keys with their counts are extracted from dictionary/list

```
def distribution_list_freq_show(dist):
  sum_count = 0
  for count in dist:
    sum_count += count
  for c_code, count in enumerate(dist):
    if count > 0:
      print('{}\t{}'.format(chr(c_code),100.0 * count/sum_count))

dist = distribution_list_new(stream)
distribution_list_freq_show(dist)
```

Function definitions (9/14)

- let us now wrap previous code involving REs (see frame 123) into own functions
- let us start with the conversion of date formats
- the conversion function returns a dictionary with the keys `year`, `month` and `day` as keys and the corresponding values

```
def date_parse(date):
  m = re.search(r'(\d{4})-(\d{2})-(\d{2})', date)
  if not m:
    return None
  d = dict()
  d['y'] = int(m.group(1))
  d['m'] = int(m.group(2))
  d['d'] = int(m.group(3))
  return d
```

if input string does not conform to the appropriate format `yyyy-mm-dd`, return value is `None`

- `print`-function can directly output all key/value pairs of dictionary

```
for date in ['2014-12-13','2016-01-02']:
  d = date_parse(date)
  if d is not None:
    print(d)
```

```
{'m': 12, 'd': 13, 'y': 2014}
{'m': 1, 'd': 2, 'y': 2016}
```

Function definitions (10/14)

- the previous code for printing pythagorean numbers (see frame 103) is wrapped into a function which takes the parameter *n* defining the upper bound for the values to generate
- the function returns the list of pythagorean triples, each of which is represented by a list of length 3

```
def pythagorean(n):
  triples = list()
  for i in range(1,n+1):
    for j in range(i,n):
      square_sum = i**2 + j**2
      k = int(sqrt(square_sum))
      if square_sum == k**2:
        triples.append([i, j, k])
  return triples
```

- to output a triple with single spaces between the numbers, we want to use the `join`-method

Function definitions (11/14)

- but then we need to convert the list `pt` of 3 numbers into a list of 3 strings, by applying the `str`-method to each number
- e.g. `str(1)` delivers the string `'1'`
- the application of `str` to each element in `pt` can be done conveniently using `map`

```
for pt in pythagorean(20):
  print('\t'.join(map(str,pt)))
```

- the output of this code is identical to the one shown on frame 103

Function definitions (12/14)

`map(func, iterable)` $\Rightarrow$ map object

return an iterator that computes the function using arguments from the iterable.

```
def increment(x):
  return x+1

for i in map(increment,[1,2,3]):
  print(i)
```

```
2
3
4
```

Function definitions (13/14)

- as we will later need to construct the reverse complement of a DNA several times, we wrap the code into a function
- for this we need to implement the `reverse`-function we already used on frame 38
- this uses a method `reversed` which returns an iterator that delivers the characters of the string in reverse order
- we just have to join them using the `join`-method

```
def reverse(seq):
  return ''.join(reversed(seq))

def reverse_complement(seq):
  revcom = reverse(seq)
  transtab = str.maketrans('ACGTacgt','TGCAtgca')
  return revcom.translate(transtab)
```

Function definitions (14/14)

- here is a final example for wrapping previous code (see frame 49) into a function
- the second function takes one argument, namely the name of an input file, opens it for reading, embedded in a `try`/`except`-statement to handle exceptions
- in case of success, the content of the file is read, white spaces are deleted and the final string is returned

```
def myopen(filename,mode='r'):
  try:
    stream = open(filename,mode)
  except IOError as err:
    sys.stderr.write('{}: {}\n',sys.argv[0],err)
    exit(1)
  return stream

def extract_sequence_data(filename):
  stream = myopen(filename)
  return re.sub('\s','',stream.read())
```

- `myopen` uses a parameter `mode` with a default value `'r'`
- so, when opening a file for reading, we can omit second argument

Key Points on Functions

- declare a function using the header `def name(...params...)`
- `params` is a comma separated list of identifiers, the parameters defining the interface to the function
- one often uses the notion of *formal parameters*
- inside the function declaration we can access the parameters as any other variable
- the block of statements belonging to a function must be indented relative to the function header
- call a function using `name(...arguments...)` where the number of arguments must correspond to the number of parameters
- a function must be declared before its first call
- the names of variables used as arguments in a function call do not need to be identical; but they may be identical
- functions may return a value, so a function call can be part of an expression, as in `x = f(y)+ g(x)`

Newton's method to compute $\sqrt{r}$ (1/3)

- As a further example involving a function definition, we implement a method to compute $\sqrt{r}$ for some non-negative real valued number r
- of course, we could use the corresponding function of Python's Math-library
- but here we want to develop our own Python function based on Newton's Method
- the method consists of computing a sequence of floating point values

$$x_0, x_1, x_2, \ldots$$

where $x_0 = \frac{1}{2}r$ and

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{r}{x_i}\right) \tag{1}$$

We have $\sqrt{r} = \lim_{i\to\infty} x_i$, i.e. the series of x_i approximates $\sqrt{n}$ with arbitrary precision.

Newton's method to compute $\sqrt{r}$ (2/3)

- now let us develop Python code implementing this method
- at first note that x_i is only used for computing $x_{i+1} = \frac{1}{2}\left(x_i + \frac{r}{x_i}\right)$
- so we can use a single variable x which is overwritten in each iteration
- the parameter to the function `newton_sqrt` is of course the value r for which we want to compute $\sqrt{r}$
- additionally we allow to specify the number of iterations, with a default value of 20

```
def newton_sqrt(r,its = 20): # number of iterations
  x = r/2   # x_{0}: initial guess of sqrt(r)
  for i in range(its): # i = 0, .., its - 1
    x = 0.5 * (x + r / x)
  return x
```

- we now want to look at a few examples and verify that the method well approximates the square root

Newton's method to compute $\sqrt{r}$ (3/3)

- to this end, we output the result of the method above of the result of the function `sqrt` from the Python Math-library.

```
from math import sqrt

for r in [12345.0,83235.9,93483.2]:
  print('sqrt({}): newton={:.14f}\n{}python={:.14f}'
        .format(r,newton_sqrt(r),' ' * 15,sqrt(r)))
```

```
sqrt(12345.0): newton=111.10805551354051
               python=111.10805551354051
sqrt(83235.9): newton=288.50632575387317
               python=288.50632575387317
sqrt(93483.2): newton=305.75022485682655
               python=305.75022485682655
```

- for the 3 numbers, the computed values are exact up to a precision of 14 digits.

Recursive definitions (1/1)

- recursion is often used in defining notions via a self-reference, like in:

Definition of $\mathbb{N}$, the set of natural numbers

1. 1 is in $\mathbb{N}$
2. if n is in $\mathbb{N}$, then so is $n+1$
3. $\mathbb{N}$ is the smallest set satisfying 1 and 2

- here natural numbers are constructed via a base case, an if-then rule and a rule to exclude extra elements
- basic arithmetic operations can be defined recursively[2]

Addition

$$0 + a = a$$
$$(1+n) + a = 1 + (n+a)$$

Multiplication

$$0 \cdot a = 0$$
$$(1+n) \cdot a = a + n \cdot a$$

Exponentiation

$$a^0 = 1$$
$$a^{1+n} = a \cdot a^n$$

[2] https://en.wikipedia.org/wiki/Recursive_definition

Recursive functions (1/2)

- now apply principle of recursion in declarations of Python-functions
- a function is recursive, if it calls itself
  - directly or
  - indirectly via other functions
- recursive functions often lead to elegant solutions of problems that may otherwise be very difficult to solve
- to illustrate this, lets start with a recursive definition of the faculty and a corresponding python function `fac`

Recursive functions (2/2)

definition of faculty

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

postfix operator ! $\Rightarrow$ function name `fac`

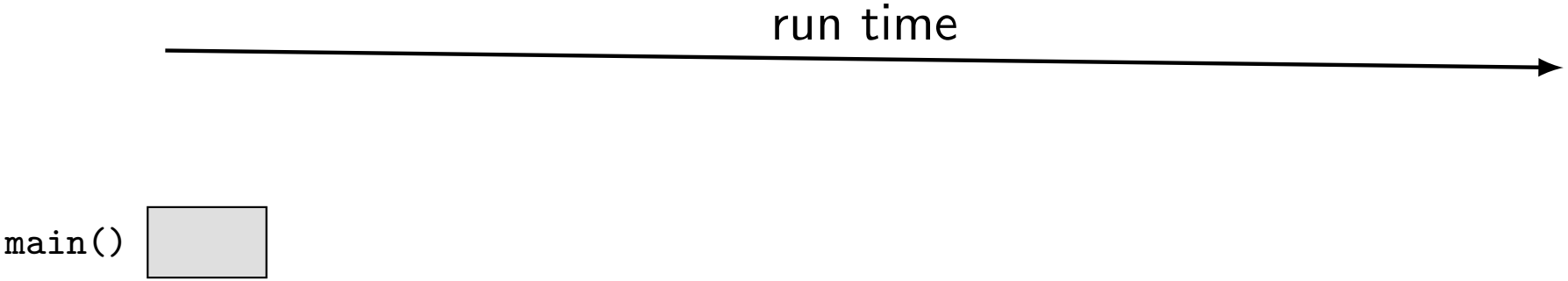
```
def fac(n):
  if n == 0:
    return 1
  return n * fac(n-1)
```

```
def main():
  return fac(3)
```

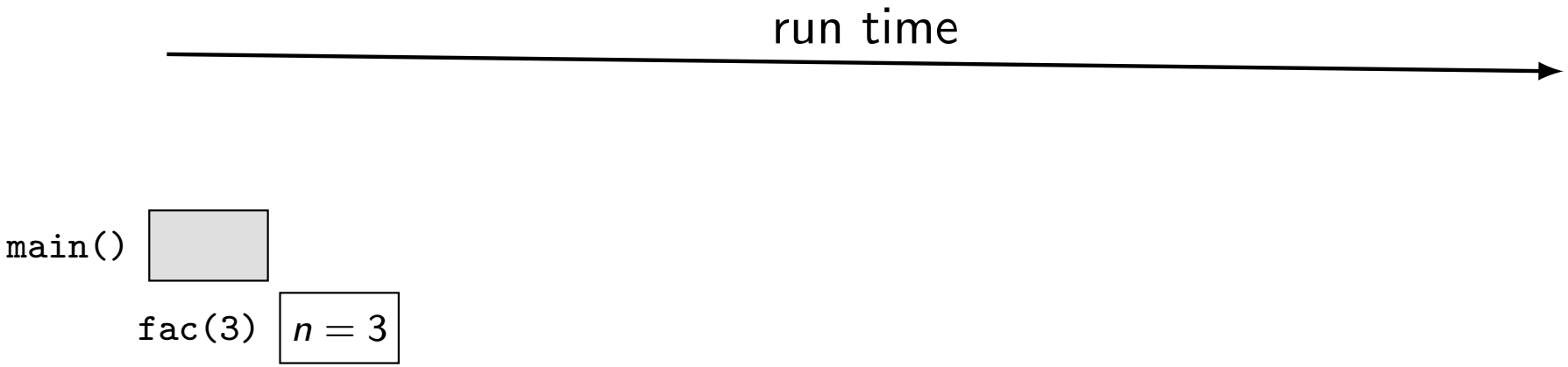
Placement of operators

- postfix operator: after its arguments
- prefix operator: before its arguments (e.g. `not`)
- infix operator: between arguments (e.g. `+`)

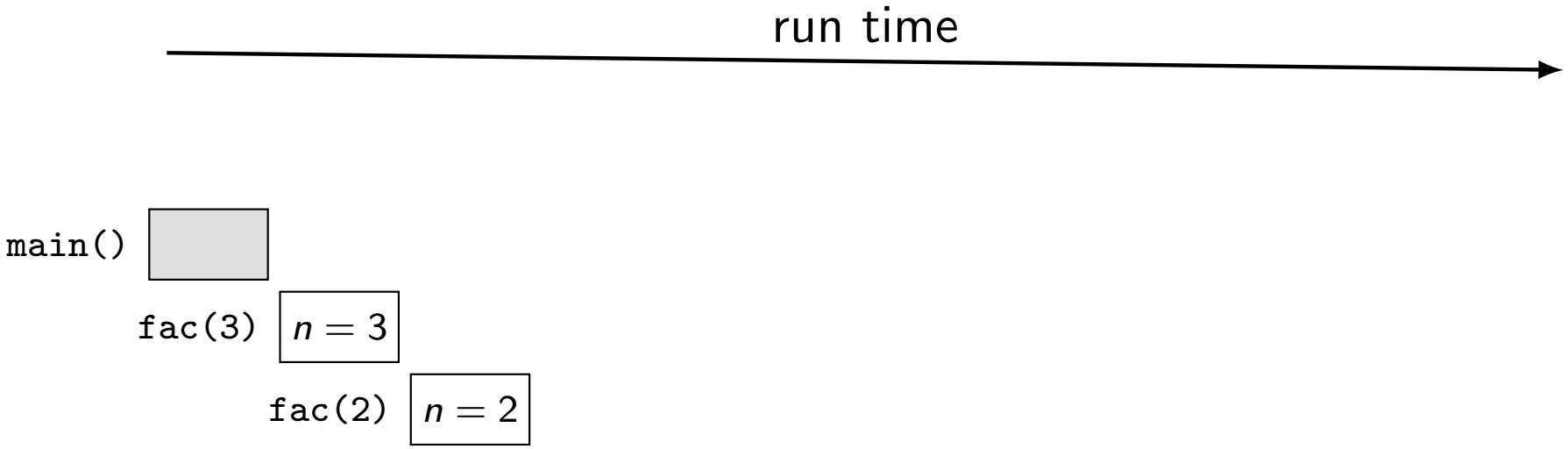
```
def fac(n):
  if n == 0:
    return 1
  return n * fac(n-1)
```



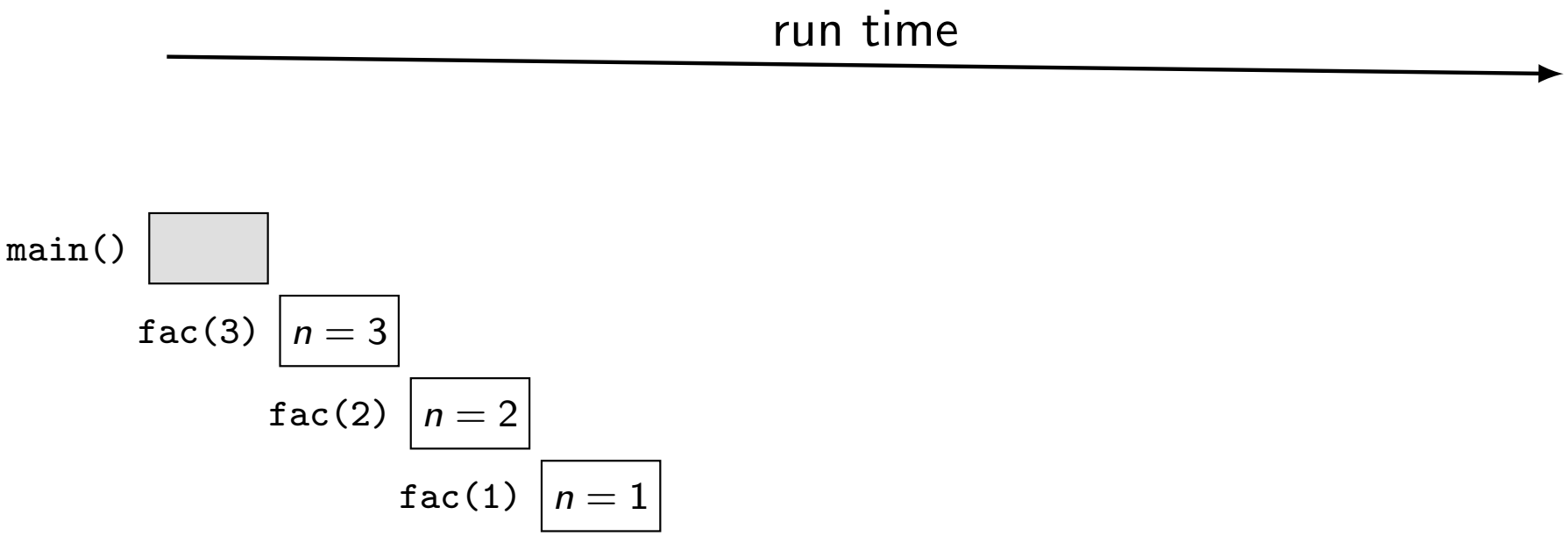
```
def fac(n):
  if n == 0:
    return 1
  return n * fac(n-1)
```



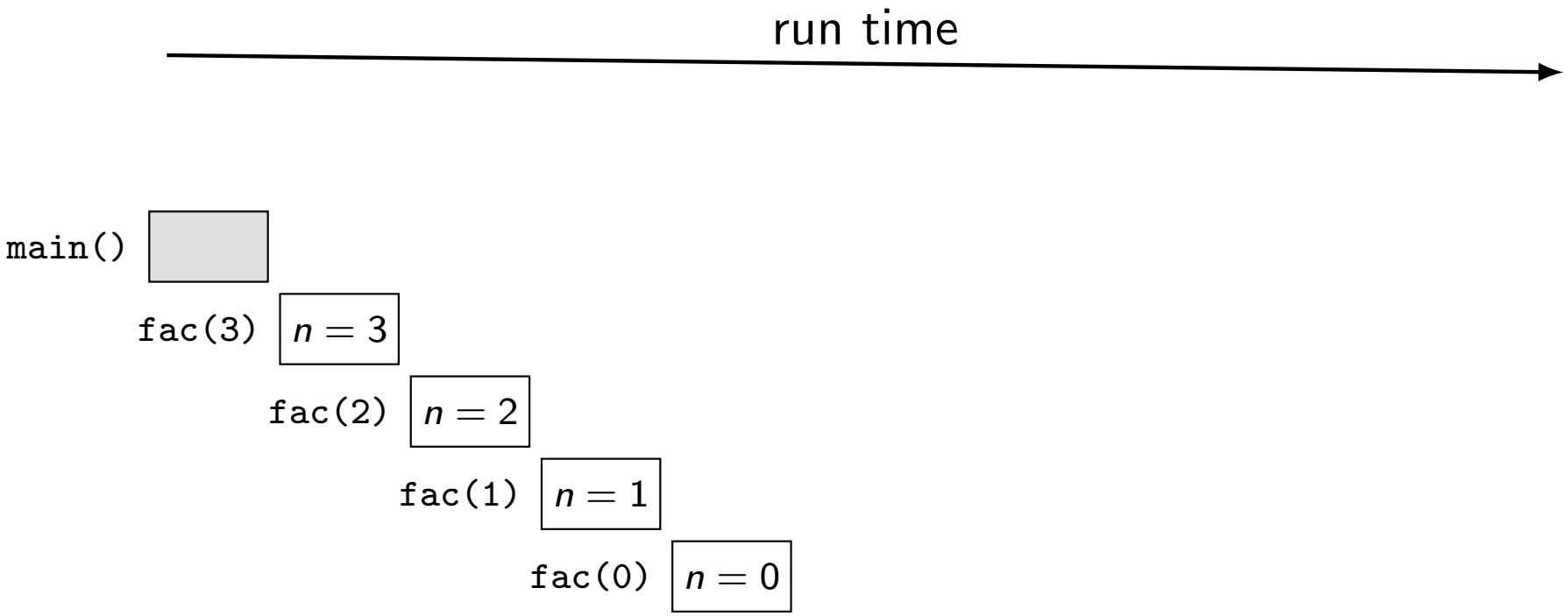
```
def fac(n):
  if n == 0:
    return 1
  return n * fac(n-1)
```



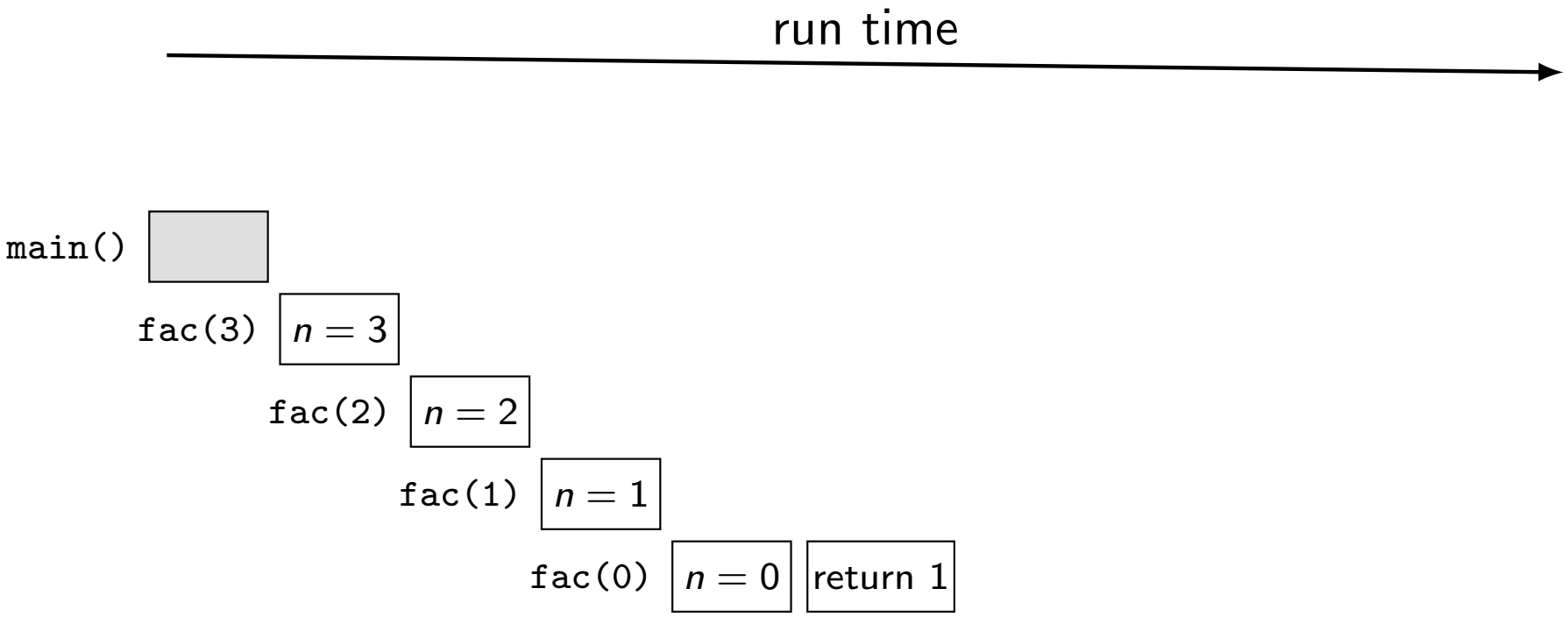
```
def fac(n):
  if n == 0:
    return 1
  return n * fac(n-1)
```



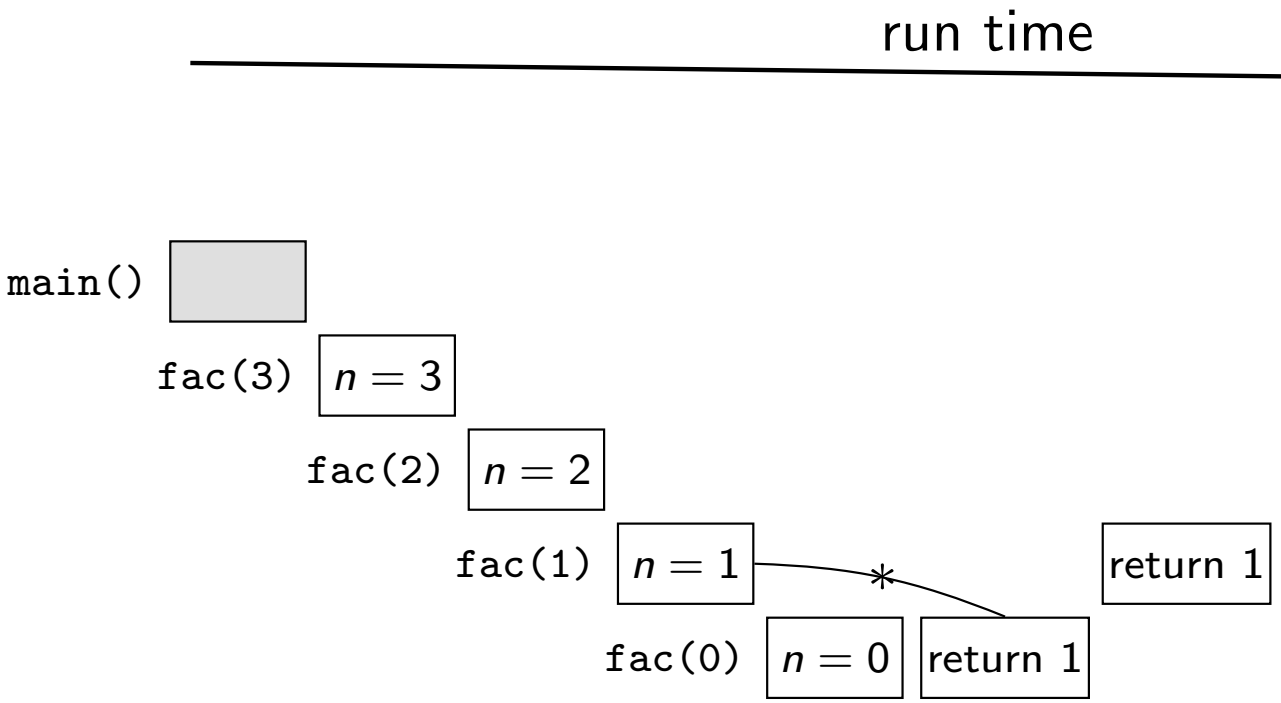
```python
def fac(n):
  if n == 0:
    return 1
  return n * fac(n-1)
```



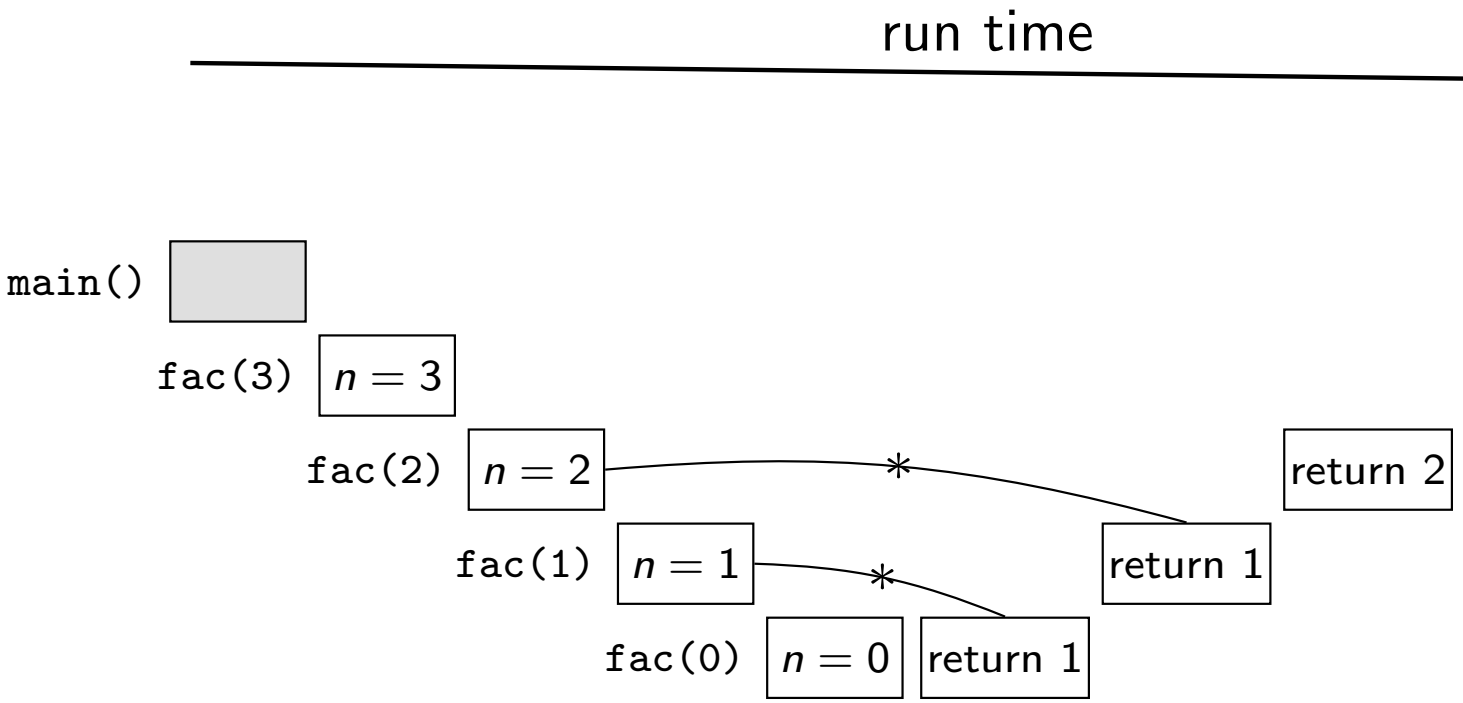
```
def fac(n):
  if n == 0:
    return 1
  return n * fac(n-1)
```



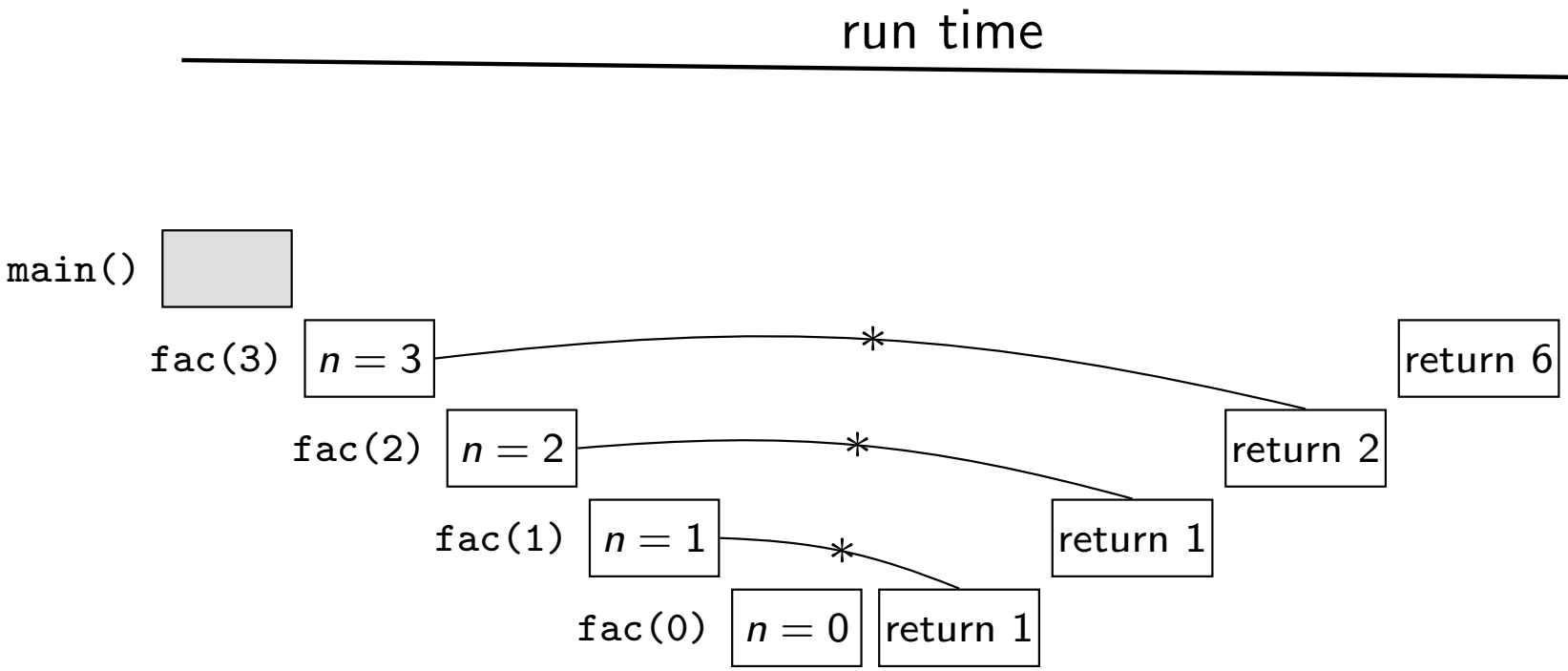
```
def fac(n):
  if n == 0:
    return 1
  return n * fac(n-1)
```



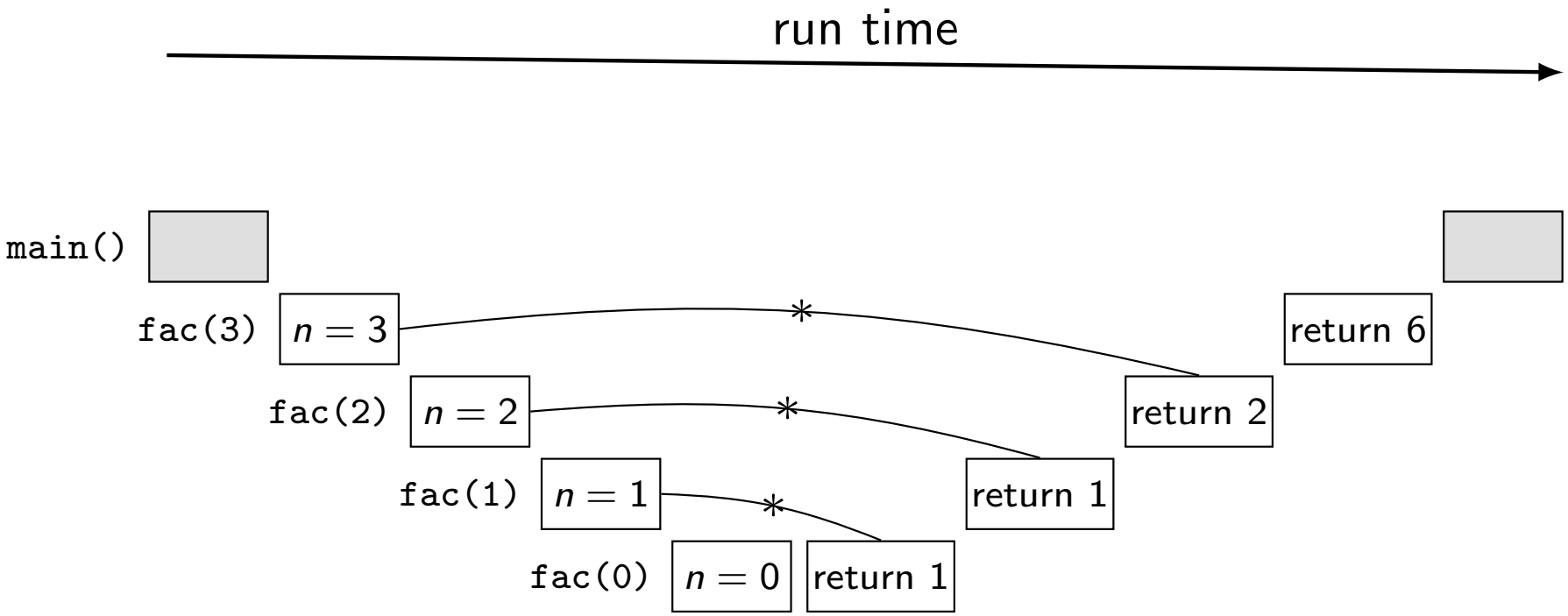
```
def fac(n):
  if n == 0:
    return 1
  return n * fac(n-1)
```



```
def fac(n):
  if n == 0:
    return 1
  return n * fac(n-1)
```



```
def fac(n):
  if n == 0:
    return 1
  return n * fac(n-1)
```



Laws of recursion[3]

All recursive functions must obey three important laws

1. a recursive function must have at least one base case
2. a recursive function must change its state and move towards one of the base cases
3. a recursive function must call itself (directly or indirectly).

The laws obviously hold for `fac`:

```
def fac(n):
  if n == 0:
    return 1
  return n * fac(n-1)
```

1. base case: `if n == 0: return 1`
2. move toward base case: `return ...fac(n-1)`
3. call itself recursively: `fac(n-1)`

[3] http://interactivepython.org/runestone/static/pythonds/Recursion/TheThreeLawsofRecursion.html

More examples of recursive functions (1/4)

```
def ls(l):
  if len(l) == 0:
    return 0
  return l[0] + ls(l[1:])

l = [1,2,3,4]
print('ls({})={}'.format(l,ls(l)))
```

```
ls([1, 2, 3, 4])=10
```

- `ls(l)` computes the sum of the elements in the list

More examples of recursive functions (2/4)

```
def myr(s):
  if len(s) == 0:
    return s
  return myr(s[1:]) + s[0]

s = 'abcd'
print('myr({})={}'.format(s,myr(s)))
```

```
myr(abcd)=dcba
```

- `myr(s)` computes the reverse of the string `s`

More examples of recursive functions (3/4)

```
def palindrome(s):
  if len(s) == 0 or len(s) == 1:
    return True
  return s[0] == s[-1] and palindrome(s[1:-1])

for w in ['kayak','radar','wassamassaw','madam','level','noon']:
  assert palindrome(w)

for w in ['abca','ab','abc','abab','abba']:
  assert not palindrome(w)
```

```
Traceback (most recent call last):
  File "Math/recursion.py", line 66, in <module>
    assert not palindrome(w)
AssertionError
```

- for a string `s`, `palindrome(s)` returns `True` iff `s` is identical to its reverse
- after the keyword `assert` we specify conditions which are supposed to hold; if not, then the program outputs an appropriate error message and exits with an error code

More examples of recursive functions (4/4)

```
def toStr(n,base):
  characters = '0123456789ABCDEF'
  if n < base:
    return characters[n]
  return toStr(n//base,base) + \
         characters[n%base]

for n in [17,1234,837373]:
  for b in [2,8,16]:
    print('{}\t{}'.format(n,toStr(n,b)))
```

```
17      10001
17      21
17      11
1234    10011010010
1234    2322
1234    4D2
837373  11001100011011111101
837373  3143375
837373  C6FD
```

- `toStr(n,base)` converts the integer to its string representation with respect to the given base

Function calls and stack frames

- each function call in Python creates a stack frame which stores the local variables and parameters of the function
- when the function returns, the return value is left on top of the stack for the calling function to access
- example call is `toStr(5,2)`
- top frame will be leave `'1'` on stack after call
- middle frame will leave `'10'` on stack after call
- bottom frame will be leave `'101'` on stack after call

```
def toStr(n,base):
  characters = '0123456789ABCDEF'
  if n < base:
    return characters[n]
  return toStr(n//base,base) + \
         characters[n%base]
```

`toStr(1,2)`
$n = 1$
$b = 2$
`characters[1]`

`toStr(2,2)`
$n = 2$
$b = 2$
`toStr(2//2,2)+characters[2%2]`

`toStr(5,2)`
$n = 5$
$b = 2$
`toStr(5//2,2)+characters[5%2]`

Passing data to functions (1/8)

- in each previous example showing the declaration of a function, the functions have parameters: `inputfile`, `n`, `date`, `dist`, `s`, `t`, `r`
- these abstract from the concrete values of the computation
- only when the function is called, the parameters are substituted by concrete values
- so by calling the functions with different values, we can reuse the code
- the values computed by the functions (i.e. those which become available) outside of the function are specified in `return`-statements
- up until, we did not assign a value to the parameters
- this is consistent with the way we use *mathematical* functions
- however, in Python, we can also modify function parameters
- this issue is discussed next
- consider the following function which takes three arguments, an integer argument, a string argument and a list argument

Passing data to functions (2/8)

- it adds something to these, using the +-operator, which has a different meaning for each of these values

```
def addsomething(n,s,l):
  print('parameters: n={},s={},l={}'
        .format(n,s,l))
  n += 1
  s += 'C'
  l += [3]
  print(' after inc: n={},s={},l={}'
        .format(n,s,l))

n = 5
s = 'A'
l = [1]
addsomething(n,s,l)
print('after call: n={},s={},l={}'
      .format(n,s,l))
```

```
parameters: n=5,s=A,l=[1]
 after inc: n=6,s=AC,l=[1, 3]
after call: n=5,s=A,l=[1, 3]
```

- so the modifications inside the function have no effect on the number and the string, but only on the list
- this is due to the immutability of numbers/strings and mutability of lists

Passing data to functions (3/8)

- the immutability/mutability can be traced by adding the following statement after each call of `print` in the previous program

```
print('ids={},{},{}'.format(id(n),id(s),id(l)))
```

- `id`, when applied to a variable returns its *identity*, i.e. a unique integer
- two variables with identical identities refer to same location in memory
- here is output of the modified program (with shortened ids):

```
parameters: n=5,s=A,l=[1]     ids=8656,2856,5832
 after inc: n=6,s=AC,l=[1, 3] ids=8688,5624,5832
after call: n=5,s=A,l=[1, 3]  ids=8656,2856,5832
```

- so `l` has the same identity in all its occurrences: the operation `+=` affects the original list
- for `n` and `s`: before the assignment they have the same identity as outside of the function, but after the assignment the identity changes, because for immutable object, a copy with different identity is created

Passing data to functions (4/8)

- mutability of lists however only allows to add something to the list or modify its contents, but one cannot overwrite the list itself, as illustrated in the following example:

```
def overwrite_list(l):
  print('param:        l={},id={}'.format(l,id(l)))
  l = [3]
  print('after write: l={},id={}'.format(l,id(l)))

l = [1]
overwrite_list(l)
print('after call:  l={},id={}'.format(l,id(l)))
```

```
param:        l=[1],id=1832
after write: l=[3],id=8680
after call:  l=[1],id=1832
```

- so before the complete list is overwritten in the function, a copy of `l` with different identity is created and the original list is unmodified

Passing data to functions (5/8)

- Python passes references of parameters to a function
- for immutable objects, such as numbers and strings:
  - whenever a statement changing the value of the variable, a copy of the contents of the variable with a different identity is created and modified ⇒ no effect on original values
- for mutable objects, such as lists and dictionaries which are modified, but not overwritten, the modification is on the referenced lists/dictionaries and have an effect on the original values in the list/dictionary
- the following example shows this for modifications by an update and the method `pop`

Passing data to functions (6/8)

```
def updatepop(a,b):
  print('in func:      a = {}'.format(a))
  print('in func:      b = {}'.format(b))
  a[0] = '2'
  b.pop(0)

a = ['1','3']
b = ['a','b']

print('before call: a = {}'.format(a))
print('before call: b = {}'.format(b))

updatepop(a,b)

print('after call:  a = {}'.format(a))
print('after call:  b = {}'.format(b))
```

- a and b are mutable, i.e. references to lists
- the referenced lists are modified

⇒ update and pop both have an effect on the original lists:

```
before call: a = ['1', '3']
before call: b = ['a', 'b']
in func:     a = ['1', '3']
in func:     b = ['a', 'b']
after call:  a = ['2', '3']
after call:  b = ['b']
```

- we now consider some convenient notations for parameter lists, i.e. default parameters and variable lists of arguments

Passing data to functions (7/8)

- in the standard case, the number of parameters in a function declaration is the same as the number of arguments in its call
- note that `__name__` is the attribute of the function storing its name
- after the declaration we always show an example call and the output following ⇒

```
def func1(a, b, c):
  print('{}: {} {} {}'.format(func1.__name__, a, b, c))
```

`func1(1,2,3)` ⇒ `func1: 1 2 3`

- in `func2`, the third parameter has a default, so it can be omitted when the default value is used

```
def func2(a, b, c=3):
  print('{}: {} {} {}'.format(func2.__name__, a, b, c))
```

`func2(1,2)` ⇒ `func2: 1 2 3`

`func2(1,2,5)` ⇒ `func2: 1 2 5`

Passing data to functions (8/8)

- for variable argument lists one uses the *-operator in front of the last parameter
- all extra arguments are available in this parameter, which is a list

```
def func3(a, b, *otherargs):
  str_otherargs = ' '.join(map(str,otherargs))
  print('{}: {} {} {}'.format(func3.__name__, a, b, str_otherargs))
```

`func3(1, 2, 3, 4, 5)` ⇒ `func3: 1 2 3 4 5`

`func3(1, 2)` ⇒ `func3: 1 2`

- in the first call to `func3`, we have `a=1`, `b=2` and `otherargs=[3,4,5]`.
- in the second call to `func3`, we have `a=1`, `b=2` and `otherargs=[]`.

Importing functions (1/1)

- to build reusable software it is necessary to distribute program code over modules and libraries
- put your function into a separate file, for example `mylib.py`
- suppose `mylib.py` contains the declaration of a function `func1`
- with a statement `import mylib` in your main program you can use the functions there, but you have to use `mylib.func1` for every call of `func1`
- so it is better use use the statement `from mylib import func1`
- then you can use `func1` without again referring to `mylib`
- if you have `mylib.py` in a different directory, then you should extend the environment variable `PYTHONPATH` like this:

  ```
  export PYTHONPATH="${PYTHONPATH}":${HOME}/pbi/python/lib"
  ```

- the python-interpreter looks for modules in the colon-separated list of directories specified by `PYTHONPATH`
- so `PYTHONPATH` plays the same role for Python as `PATH` plays for Linux/macOS

Reading and representing data matrices (1/9)

- in this section we consider how to read a file with data represented as matrix, in which
  - the columns have attributes (shown in first line of matrix) and
  - each line contains the values for these attributes including a specific column which serves as a key
- here is an example of such a matrix involving attributes of elements:

```
atomicNumber symbol  name      meltingPoint
1            H       Hydrogen  14
2            He      Helium
3            Li      Lithium   454
```

- missing values may occur, such as the melting point for Helium
- the values in the `symbol` column serve as keys, but we could use the atomic number of the name as well
- the matrices we consider here are stored as text files and we want to extract the data from such a text file

Reading and representing data matrices (2/9)

- for the exercises you will use a matrix with 119 rows and 20 columns, in which the columns are separated by tabs[1]
- a natural ways to represent such a matrix in Python is to use a dictionary, say `atom_matrix`, whose values are dictionaries
- for ease of notation we want to use
  - the element symbols as keys for the rows and
  - the attributes as keys for the colums,

  such that we can write `atom_matrix["Li"]["meltingPoint"]` to obtain the melting point of Lithium.
- this means that `atom_matrix` as well as `atom_matrix["Li"]` must be a dictionary, as we use strings as keys (which we cannot use for lists)
- so `atom_matrix` must be a dictionary
  - whose keys are the element-symbols and
  - whose values are dictionaries with element attributes as keys

1: the original file is from `https://github.com/andrejewski/periodic-table.git` and was comma separated

Reading and representing data matrices (3/9)

- so the previous matrix

```
atomicNumber symbol  name      meltingPoint
1            H       Hydrogen  14
2            He      Helium
3            Li      Lithium   454
```

- would be represented as follows:

```
{'He': {'atomicNumber': '2', 'symbol': 'He', 'meltingPoint': '',
        'name': 'Helium'},
 'Li': {'atomicNumber': '3', 'symbol': 'Li', 'meltingPoint': '454',
        'name': 'Lithium'},
 'H': {'atomicNumber': '1', 'symbol': 'H', 'meltingPoint': '14',
       'name': 'Hydrogen'}}
```

- here the curly brackets enclose the key/value pairs of a dictionary and a colon `:` is used to separate a key from the corresponding value

Reading and representing data matrices (4/9)

```
def data_matrix_new(lines,key_col = 1,sep = '\t'):
  matrix = dict()
  attribute_list = None
  for line in lines:
    ls = line.rstrip('\n').split(sep)
    if attribute_list is None: # in first line
      attribute_list = ls
    else:  # not in first line: values
      if len(ls) != len(attribute_list):
        sys.stderr.write('line has {} columns, but {} expected\n'
                         .format(len(ls),len(attribute_list)))
        exit(1)
      line_dict = dict()
      for attr, value in zip(attribute_list,ls):
        line_dict[attr] = value
      k = ls[key_col]
      if k in matrix:
        sys.stderr.write('key {} in line {} is not unique\n'
                         .format(k,2+len(matrix)))
        exit(1)
      matrix[k] = line_dict
  return attribute_list, matrix
```

Reading and representing data matrices (5/9)

- the function shown above has 3 parameters:
  - an object `lines` over which we can iterate to obtain the lines of the matrix (`lines` will be a stream in our application, but it can also be an array of lines)
  - a column index for the column from which we obtain the keys (1 by default)
  - a separator on which we split the lines (a tabulator by default)
- the variables for the list of attributes and the data matrix are first initialized and will be returned at the end
- in an iteration over the lines we first strip trailing newlines and then split the lines on the separator ⇒ obtain a list `ls`
- we cannot strip white spaces in general since this would delete trailing tabulators which would be a problem for the Helium line (which ends with a tab)
- when reading the first line, `attribute_list` is not defined, so we assign the list `ls` of values to `attribute_list`

Reading and representing data matrices (6/9)

- for all other lines, we first check, if we have the same number of values as in `attribute_list`; if not, generate an error message and exit
- otherwise, we want to construct a new dictionary `line_dict` for this line
- using a method `zip` we simultaneously iterate over `attribute_list` and `ls` to obtain pairs of attributes and corresponding values which we use as key/value pairs for the initially empty dictionary `line_dict`

`zip(iter1, iter2 [,...])` ⇒ zip object

Return a zip object whose `.__next__()` method returns a tuple where the i-th element comes from the i-th iterable argument. The `.__next__()` method continues until the shortest iterable in the argument sequence is exhausted and then it raises `StopIteration`.

```
attribute_list = ['atomicNumber','symbol',
                  'name', 'meltingPoint']
ls = ['1','H','Hydrogen','14']
for attr, value in zip(attribute_list,ls):
  print('{} {}'.format(attr,value))
```

```
atomicNumber 1
symbol H
name Hydrogen
meltingPoint 14
```

Reading and representing data matrices (7/9)

- then we extract the key `k` from the current line at the given index in `ls` and check if it already occurs in the matrix; if so, we generate an error message and exit
- otherwise we store the dictionary `line_dict` in `matrix[k]` and can finish with a return statement
- in many cases we want to output the matrix or parts of it
- the first output function has parameters for
  - the matrix itself,
  - the separator to show,
  - the attributes which we want to show and
  - the keys for which we want to show the attributes

```
def data_matrix_show(matrix,sep,attributes,keys):
  for key in keys:
    for a in attributes:
      if matrix[key][a] != '':
        print('{}{}{}{}{}'.format(key,sep,a,sep,matrix[key][a]))
```

Reading and representing data matrices (8/9)

- here is an example of an application of the previous show-function
- it also shows that `data_matrix_new` allows to extract the matrix from a list of strings in the appropriate format (i.e. tab separated)

```
from data_matrix import *

lines = ['atomicNumber  symbol  name  meltingPoint',
         '1 H Hydrogen  14',
         '2 He  Helium  ',
         '3 Li  Lithium 454']

attribute_list, atom_matrix = data_matrix_new(lines)
my_attributes = ['name','meltingPoint']
my_elements = ['H','Li']
data_matrix_show(atom_matrix,' ',my_attributes,my_elements)
```

```
H name Hydrogen
H meltingPoint 14
Li name Lithium
Li meltingPoint 454
```

Reading and representing data matrices (9/9)

- when extracting information from a text file it is always good practice to check that no information is lost or accidentally modified
- the simplest way to test this is to write a function which outputs the internal representation (i.e. the data matrix) in the same format as the input
- this is what the following function does
- it uses the same four arguments as the previous function
- in contrast to this, it first prints the header line with the attributes and then for each key the line of values for this key, all separated by the string `sep`

```
def data_matrix_show_orig(matrix,sep,attributes,keys):
  print(sep.join(attributes))
  for key in keys:
    line_elems = list()
    for a in attributes:
      line_elems.append(matrix[key][a])
    print(sep.join(line_elems))
```

Flexible use of data matrices (1/9)

- we now want to use the three functions for creating and showing data matrices in a flexible way from the command line
- suppose our program is `data_matrix_main.py`
- we want to be able to call it as follows (and of course in many other ways, without changing the code)

```
data_matrix_main.py -a atomicNumber symbol name meltingPoint
                    -k H He Li -o atom-data.tsv
```

- as in many Linux programs, we use the letter - as prefix to denote an option, optionally followed by some strings (not prefixed with -)
- in the above call, we have specified
  - option `-a`, followed by the attributes we want to see
  - option `-k` followed by the keys for which to output the attribute values
  - `-o` specifies that we want the output in the same format as the input
  - the input file `atom-data.tsv` containing the data matrix in `.tsv` format (which stands for tab separated values)

Flexible use of data matrices (2/9)

– to implement a program with such a set of options, we do not have to start from scratch, but can make use of a module `argparse`:

```
import sys, argparse

def parse_command_line():
  p = argparse.ArgumentParser()
  p.add_argument('-k','--keys',nargs='+',default=None,
                 help='specify keys for which values are output')
  p.add_argument('-a','--attributes',nargs='+',default=None,
                 help='specify attributes output')
  p.add_argument('-o','--orig',action='store_true',default=False,
                 help='output key/value pairs in original format')
  p.add_argument('-s','--sep',type=str,default='\t',
                 help='specify column separator, default is Tab')
  p.add_argument('inputfile',type=str,
                 help='specify inputfile (mandatory argument)')
  return p.parse_args()
```

Flexible use of data matrices (3/9)

- with the assignment `args = parse_command_line()` after the method declarations, the argument parser is called
- if we use the implicit option `-h` (for help), the program reports the following formatted description of the possible options and exits

```
$ data_matrix_main.py -h
usage: data_matrix_main.py [-h] [-k KEYS [KEYS ...]]
                           [-a ATTRIBUTES [ATTRIBUTES ...]] [-o] [-s SEP]
                           inputfile
positional arguments:
  inputfile             specify inputfile (mandatory argument)
optional arguments:
  -h, --help            show this help message and exit
  -k KEYS [KEYS ...], --keys KEYS [KEYS ...]
                        specify keys for which values are output
  -a ATTRIBUTES [ATTRIBUTES ..], --attributes ATTRIBUTES [ATTRIBUTES ..]
                        specify attributes output
  -o, --orig            output key/value pairs in original matrix format
  -s SEP, --sep SEP     specify column separator, default is Tab
```

Flexible use of data matrices (4/9)

- now let us decipher the parts of the specification of the option parser
- we need to specify the module `argparse` in the import list
- this module implements a class `ArgumentParser` so that `p = argparse.ArgumentParser()` creates a new instance of the class, i.e. an object `p` with methods for specifying and running an option parser
- the most important method is `add_argument` which has several parameters, some of which are named, like `default`, `help`, `type`
- this method adds an argument to the parser
- there are basically two kinds of arguments, an option beginning with `-` and any non-optional argument
- in our case we use `inputfile` as non-optional argument:

```
p.add_argument('inputfile',type=str,
               help='specify inputfile (mandatory argument)')
```

- `type=str` specified that we require a string argument

Flexible use of data matrices (5/9)

- we have three kinds of options: `-k` and `-a` require a non empty (`nargs='+'`) list of white space separated arguments

```
p.add_argument('-k','--keys',nargs='+',default=None,
                help='specify keys for which values are output')
```

- if `-k` is not used, then the option has value `None`
- to please users who prefer long options, we have added a synonym option `--keys`
- the second kind of option is `-o` (synonym `--orig`) which is a boolean option, i.e. it has the value `True` or `False`

```
p.add_argument('-o','--orig',action='store_true',default=False,
                help='output key/value pairs in original format')
```

- if the option `-o` is used, the value of the option is `True`, otherwise it gets the default value `False`

Flexible use of data matrices (6/9)

- the third kind of option is `-s` (synonym `--sep`), which has exactly one string argument
- if this option is not used, it has the default value `\t`

```
p.add_argument('-s','--sep',type=str,default='\t',
                help='specify column separator, default is Tab')
```

- with `args = parse_command_line()`, the parser is called in line `p.parse_args()` which implicitly takes its input from the list `sys.argv` and looks for the different options in this list, thereby considering all details the programmer has specified
- the return value is an object `args` which stores the values of options in corresponding variables whose names where implicitly specified in the parser:

Flexible use of data matrices (7/9)

argument/option	variable name	type
inputfile	args.inputfile	str
-k, --keys	args.keys	list(str)
-a, --attributes	args.attributes	list(str)
-o, --orig	args.orig	bool
-s, --sep	args.sep	str

– these variables are used in the rest of the program:

```
args = parse_command_line()

try:
  stream = open(args.inputfile)
except IOError as err:
  sys.stderr.write('{}: {}\n'.format(sys.argv[0],err))
  exit(1)
```

Flexible use of data matrices (8/9)

```
from data_matrix import *

attribute_list, matrix = data_matrix_new(stream)
stream.close()
if args.attributes:  # option -a was used
  attributes = args.attributes
else:
  attributes = attribute_list # use all attributes
if args.keys:  # option -k was used
  keys = args.keys
else:
  keys = matrix.keys()   # use all keys
if args.orig:
  data_matrix_show_orig(matrix,args.sep,attributes,keys)
else:
  data_matrix_show(matrix,args.sep,attributes,keys)
```

- note how the case that option `-k` was not used is handled
- in this case, `args.keys` has the value `None` (the default value of the option)

Flexible use of data matrices (9/9)

- and so we use the complete key list
- the option `-a` is handled in the same way
- as a consequence, the call `data_matrix_main.py -o atom-data.tsv` in the terminal delivers the original matrix in tsv-format, except that the lines are in different order
- this can be verified by the following commands on the Terminal:

```
$ data_matrix_main.py -o atom-data.tsv | sort > tmp
$ sort atom-data.tsv | diff - tmp
```

Recall methods on dictionaries

synopsis: methods on dictionaries	
`d = dict()`	create a new empty dictionary
`d[k]`	lookup key `k` in dictionary `d`
`if k in d:`	test if key `k` is in dictionary `d`
`d[k] = v`	add key value pair `k`/`v` to dictionary `d`; if value for key `k` already exists, then overwrite current value
`list(d.keys())`	return list of keys of dictionary `d` in arbitrary order
`list(d.values())`	return list of values of dictionary `d` in arbitrary order
`list(d.items())`	return list of key/value pairs of dictionary `d` in arbitrary order

Abstract data types (1/2)

Algorithm development and programming process . . .

- can be very complex
- use abstractions to allow focus on the "big picture" without getting lost in the details
- create models of the problem domain
- models allow us to simplify the description of the data with respect to the considered problem
- the models we consider here are abstract data types and they describe
  - the data
  - and the allowed operations on the data

  both without regard to how they will be implemented

⇒ we are concerned only with what the data is representing and not with how it will eventually be constructed

⇒ encapsulation around the data

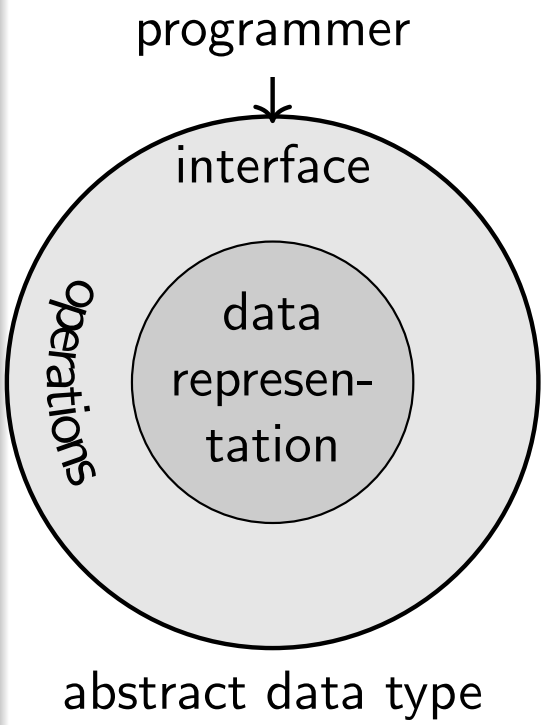
Abstract data types (2/2)

Encapsulation

- by encapsulating the details of the implementation, we are hiding them from the user's view.

⇒ information hiding

- user interacts with interface, using the available operations
- implementation is hidden one level deeper.
- user is not concerned with the details of the implementation
- implementation-independent view of the data
- user can remain focused on the problem-solving process using the abstract data type (ADT)



A simple class for fractions (1/7)

- up until now we already had such an abstract view on different data types, such as list, string, dictionary
- we were only concerned about the interface, i.e. the methods provided
- and did not care about how these data types were implemented
- here we will consider how to implement our own abstract data type in form of a class in Python3

A simple class for fractions (2/7)

- this will be exemplified by implementing a first very simple class `FractionSimple` to store fractions, like $\frac{1}{2}$ or $\frac{3}{13}$ specified by numerator (Zähler) and denominator (Nenner)
- the numerator can be any integer, the denominator any positive integer
- negative fractions have a negative numerator
- a simple representation of a fraction would be a floating point number
- but for some fractions, like $\frac{1}{3}$, this would only be an approximation
- therefore we choose a representation which keeps both, the numerator and denominator

```
class FractionSimple:
  def __init__(self,top,bottom):
    self.num = top
    self.den = bottom
```

A simple class for fractions (3/7)

- the definition of a class starts with the keyword `class` followed by the name of the class, `FractionSimple` in our case, followed by a colon
- this line is the header of the class
- the class header is followed by declarations of one or more methods, with indentation relative to the class header
- there it at least one method, named `__init__` which is called immediately after the instance of the class, in this case, `FractionSimple`, is created
- `__init__` has three parameters
  - a handle named `self` to the created object (this is always the first argument of the class methods in their declaration)
  - the parameters `top` and `bottom` by which we pass the numerator and denominator, respectively, of the rational number we want to create
- `__init__` creates two variables `self.num` and `self.den`, to which we assign the values of the parameters `top` and `bottom`, respectively

A simple class for fractions (4/7)

- the notation with the initial keyword `self` makes these new variables instance variables: they represent the data of an instantiated object of the specified class
- such an instantiated object (or instance) is created by using the class name with two parameters

```
frac1 = FractionSimple(13,30)
frac2 = FractionSimple(1,15)
```

- each such expression beginning with the class name and the appropriate number of arguments (one less than the number of parameters for `__init__`), creates an object of the corresponding class and then calls `__init__` with a reference `self` to that instance
- important feature of the class: the data is not accessible from outside $\Rightarrow$ class is actually an implementation of an abstract data type

A simple class for fractions (5/7)

- we now only know how to construct new `FractionSimple`-objects, but have no other method to manipulate them
- the first method we introduce is used for adding two fractions
- to add two fractions, say $\frac{a}{b}$ and $\frac{c}{d}$ for integers a, b, c, d with $b, d > 0$ we apply the following equality

$$\frac{a}{b} + \frac{c}{d} = \frac{a}{b} \cdot 1 + \frac{c}{d} \cdot 1 = \frac{a}{b} \cdot \frac{d}{d} + \frac{c}{d} \cdot \frac{b}{b} = \frac{ad}{bd} + \frac{bc}{bd} = \frac{ad + bc}{bd} \quad (2)$$

- this is translated into the method `add` which has two fractions `f1` and `f2` as parameters and returns a new `FractionSimple`-object

```
def add(f1,f2):
  newnum = f1.num * f2.den + f1.den * f2.num # numerator of (2)
  newden = f1.den * f2.den                   # denominator of (2)
  return FractionSimple(newnum,newden)
```

- this comes directly after `__init__` at the same indentation level

A simple class for fractions (6/7)

- the method `add`
  - first computes the new numerator and the new denominator for the sum of the fractions, according to (2)
  - then stores these in local variables,
  - creates a new `FractionSimple`-object and
  - finally returns this
- now let us apply `add` (code appears at indentation level 0, after the declaration of the class)

```
frac3 = FractionSimple.add(frac1,frac2)  # frac1=13/30, frac2=1/15
```

- to obtain a string representation using the symbol / as separator, we add a method `tostring` (same indentation level as `__init__`)

```
def tostring(frac):
  return '{}/{}'.format(frac.num,frac.den)
```

- now let us apply `tostring` to `frac1`, `frac2` and their sum `frac3`

A simple class for fractions (7/7)

```
print('{} + {} = {}'
      .format(FractionSimple.tostring(frac1),
              FractionSimple.tostring(frac2),
              FractionSimple.tostring(frac3)))
```

- this delivers the following output

```
13/30 + 1/15 = 225/450
```

- the result is correct, but of course we would like to see it in the most common form, i.e. it should be displayed as 1/2
- we also would like to use standard operators like + for adding fractions and `str` for converting a fraction to a string.
- with the improved implementation, described below, we will obtain these features

An improved class for fractions (1/10)

- to prevent results like 225/450, in the improved class, we always represent the numerator and denominator in a unified form, i.e. as the smallest possible pair of numerator and denominator
- this is achieved by dividing the numerator and denominator by their greatest common divisor
- the greatest common divisor of two integer values x and y is computed by the algorithm of Euclid
- this generates a sequence of pairs $(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots$ where

$$x_i = \begin{cases} x & \text{if } i = 0 \\ y_{i-1} & \text{if } i > 0 \end{cases} \qquad y_i = \begin{cases} y & \text{if } i = 0 \\ x_{i-1} \bmod y_{i-1} & \text{if } i > 0 \end{cases}$$

  and $a \mod b$ is the remainder of the integer division $\frac{a}{b}$
- as soon as $x_i \bmod y_i = 0$ for some i, the algorithm stops and returns y_i as the greatest common divisor

An improved class for fractions (2/10)

- let us now turn this algorithmic description into Python code
- obviously, for all $i > 0$, x_i and y_i depends only on x_{i-1} and y_{i-1}
- so, as in similar previous cases, we do not need to store all x_i and y_i
- at any time only two consecutive pairs of values suffice
- we therefore use two variables `x` and `y` and store the values of the previous iteration in some temporary variables `previous_x` and `previous_y`
- then we can safely compute x_i and y_i in `x` and `y`
- this gives a first version of a function `gcd_simple` to compute the greatest common divisor

```
def gcd_simple(x,y):
  while x % y != 0:
    previous_x = x
    previous_y = y
    x = previous_y
    y = previous_x % previous_y   # % is modulus operator
  return y
```

An improved class for fractions (3/10)

- as we assign a value to `y` only in the last statement, we can replace `previous_y` by `y` in the last two statements, to obtain

```
def gcd(x,y):
  while x % y != 0:
    previous_x = x
    x = y
    y = previous_x % y
  return y
```

- it is good practice to verify the result of a computation
- of course we do not want to do this manually, but let the computer do the work
- to do so, we implement some conditions, that must hold for given x, y:
  - $gcd(x, y)$ must divide x and y without remainder
  - $gcd(x, y) = gcd(y, x)$
  - $gcd\left(\frac{x}{d}, \frac{y}{d}\right) = 1$ where $d = gcd(x, y)$

An improved class for fractions (4/10)

```
def expect_gcd(x,y,d):
  if x % d != 0:
    sys.stderr.write('expect x={} % d={} == 0, but it is {}\n'
                     .format(x,d,x % d))
    exit(1)
  if y % d != 0:
    sys.stderr.write('expect y={} % d={} == 0, but it is {}\n'
                     .format(y,d,y % d))
    exit(1)
  if d != gcd(y,x):
    sys.stderr.write('expect d={} == gcd({},{})={}\n'
                     .format(d,y,x,gcd(y,x)))
    exit(1)
  if gcd(x//d,y//d) != 1:   # // is the integer division
    sys.stderr.write('expect gcd(x/d={},y/d={}) == 1, it is {}\n'
                     .format(x//d,y//d,gcd(x//d,y//d)))
    exit(1)
```

- the actual test is implemented by a function which generate pairs of random numbers between 1 and 1 000 and verifies that the expectations hold

An improved class for fractions (5/10)

```
import random
def run_test(trials):
  for i in range(trials):
    x = random.randint(1,1000)
    y = random.randint(1,1000)
    d = gcd(x,y)
    expect_gcd(x,y,d)
```

- the code related to the computation of the gcd is stored in a file `gcd.py` which will be imported in other files by the statement

  ```
  from gcd import gcd
  ```

- in that case, the name of the executed script, stored in the variable `__name__`, is not `gcd.py` and we do not want to run the test
- otherwise, when `gcd.py` itself is the running script, then `__name__` is set equal to `'__main__'` and we want to run the test
- the case distinction is implemented at the end of `gcd.py`:

```
if __name__ == '__main__':
  run_test(100000)
```

An improved class for fractions (6/10)

- for the improved class `Fraction`, we use `gcd`
- before we store the numerator and denominator, we unify them by dividing by their gcd
- as we know that the gcd divides without remainder, we use integer division expressed by the binary operator `//`

```
class Fraction:
  def __init__(self,top,bottom):
    common = gcd(top,bottom)
    self.num = top//common  # integer division
    self.den = bottom//common

  def __str__(self):    # overload str
    if self.den == 1:
      return '{}'.format(self.num)
    else:
      return '{}/{}'.format(self.num,self.den)
```

- we also implement a pretty printing function `__str__` to convert a fraction into a string

An improved class for fractions (7/10)

- whenever the function `str` is applied to a fraction, the class-method `__str__` is called
- this concept is called *overloading*: the same name `str` is used for different functions
- as Python knows to which object `str` is applied, it can figure out the class and from this it knows which `__str__`-method to actually call
- note that each application of `.format(..)` to non-string objects also implicitly calls `str` for conversion
- we also overload the binary operator `+` by implementing a method `__add__` in the Fraction class
- note that the first argument of `__add__` is `self` (the fraction to which we want to add something) but otherwise the code is the same as in the method `add` above

An improved class for fractions (8/10)

```
def __add__(self,o_frac):    # overload +
  newnum = self.num * o_frac.den + self.den * o_frac.num
  newden = self.den * o_frac.den
  return Fraction(newnum,newden)
```

- the same approach is used to overload the equality operator
- to overload this, we must implement the method `__eq__`
- testing for equality of fractions is simple, as they are represented in a unified way: we only have to check that the numerator and the denominator are the same

```
def __eq__(self, other):   # overload ==
  return self.num == other.num and self.den == other.den
```

- again we implement a simple test which outputs some fraction and verifies that $\frac{13}{30} + \frac{1}{15}$ is equal to $\frac{4}{8}$
- note that equality implicitly introduces the inequality operator `!=`

An improved class for fractions (9/10)

```
def run_test():
  print('8/4={}'.format(Fraction(8,4)))
  frac1 = Fraction(13,30)
  frac2 = Fraction(1,15)
  frac3 = frac1 + frac2
  print('13/30 + 1/15={}'.format(frac3))
  frac4 = Fraction(4,8)
  if frac3 != frac4:
    sys.stderr.write('frac3 == {} != {} = frac4 not expected\n'
                     .format(frac3,frac4))
    exit(1)
```

- and finally, we run the test, if this module is not used as an imported module

```
if __name__ == '__main__':
  run_test()
```

- we will see several other examples of classes in the lectures and the exercises

An improved class for fractions (10/10)

- overloading will be used several times
- here is a list of operators and the corresponding method names to be used in the definition of the class

Operator	Method
str	__str__
+	__add__
-	__sub__
*	__mult__
**	__pow__
/	__truediv__
//	__floordiv__
%	__mod__
<<	__lshift__
>>	__rshift__

Operator	Method
&	__and__
|	__or__
~	__invert__
^	__xor__
<	__lt__
<=	__le__
==	__eq__
!=	__ne__
>	__gt__
>=	__ge__

Overview of Fraction-Class (1/3)

- idea: represent each fraction in a unique way by dividing numerator and denominator by their greatest common divisor
- so `Fraction(6,26)` is equivalent to `Fraction(3,13)`

```
class Fraction:
  def __init__(self,top,bottom):
    common = gcd(top,bottom)
    self.num = top//common  # integer division
    self.den = bottom//common
```

- create a fraction $\frac{6}{26}$ by `Fraction(6,26)`
- this immediately calls `__init__` (a mandatory method) which initializes the instance variables
- `Fraction(6,26)` is an instance of the class, i.e. an object with concrete values for the instance variables `self.num` and `self.den`

Overview of Fraction-Class (2/3)

- it is supposed that all updates and reads of these instance variables are done inside the class
- unfortunately, this is not enforced in Python
- example: for an object `frac3` of class `Fraction` one can write

```
frac3.num=21
print('frac3.den={}'.format(frac3.den))
```

  outside of the class declaration
- a common rule is to prefix a member variable or method with an underscore to state that it is private and is not supposed to be used outside of the class
- according to this rule we would use `self._num` and `self._den`
- in future class definitions we will apply this rule

Overview of Fraction-Class (3/3)

- method `__str__` is called when creating a string representation of a `Fraction`-object using `str` (overload `str`)

```
def __str__(self):    # overload str
  if self.den == 1:
    return '{}'.format(self.num)
  else:
    return '{}/{}'.format(self.num,self.den)
```

- method `__add__` is called when adding 2 `Fraction`s using `+` (overload `+`)

```
def __add__(self,o_frac):    # overload +
  newnum = self.num * o_frac.den + self.den * o_frac.num
  newden = self.den * o_frac.den
  return Fraction(newnum,newden)
```

- method `__eq__` is called when comparing two `Fraction`-objects using `==` (overload of `==`)

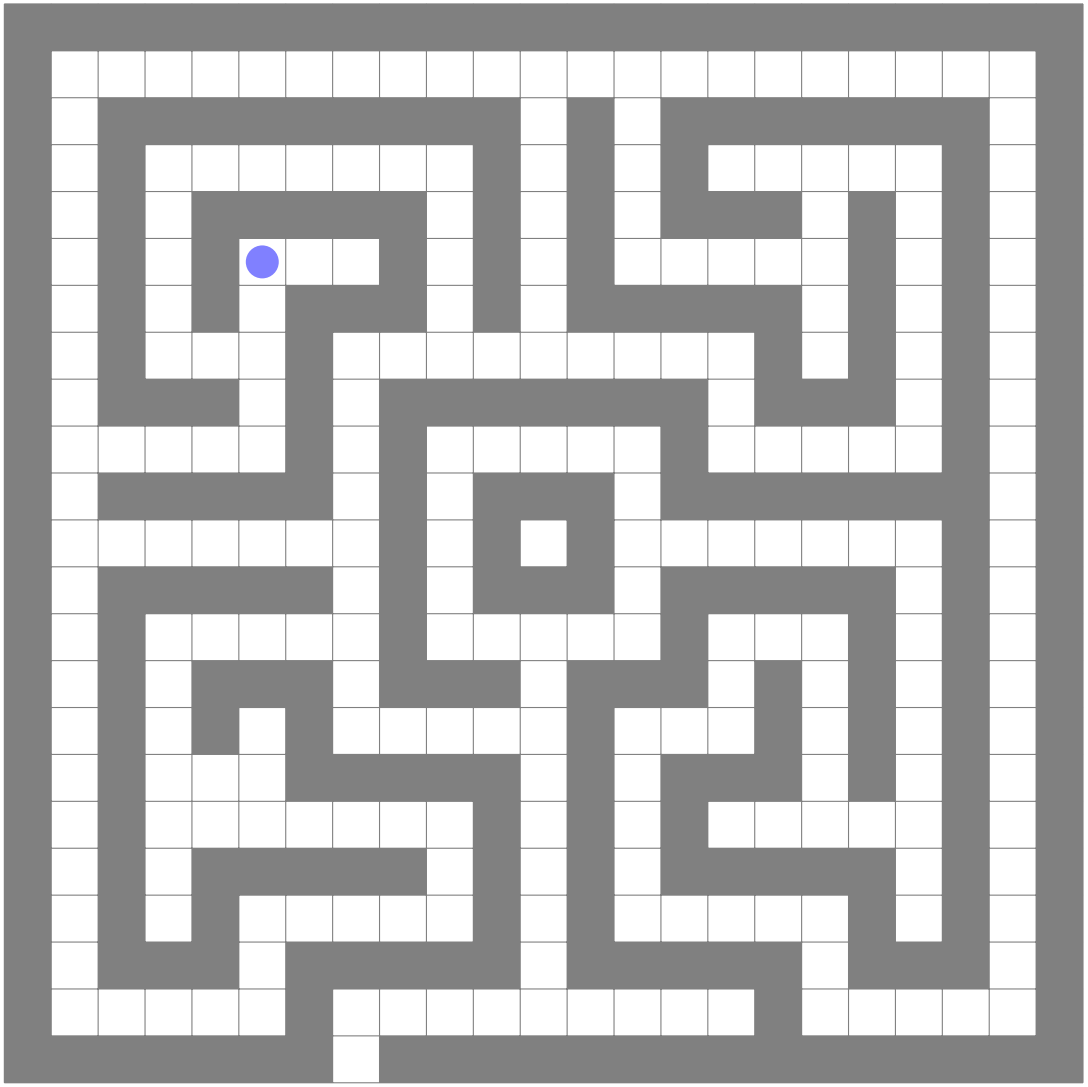
```
def __eq__(self, other):   # overload ==
  return self.num == other.num and self.den == other.den
```

Overview of Object Oriented Programming (OOP) (1/1)

- OOP: everything you manipulate is an object and the results of those manipulations are objects as well
- each object is generated as an instance of a class
- a class consists of a state (the instance variables) and a set of functions (called methods) to manipulate the state
- a class can be seen as a construction plan according to which an object (of this class), called instance, is created
- class definition of the form `class Classname:`
- minimum requirement: method `__init__` called immediately after instance was created
- in most cases: `__str__` for creating string representation (pretty printing of instance variables)
- all class methods have `self` as first parameter by which the instance variables (the state) are accessed

Exploring a maze[4]

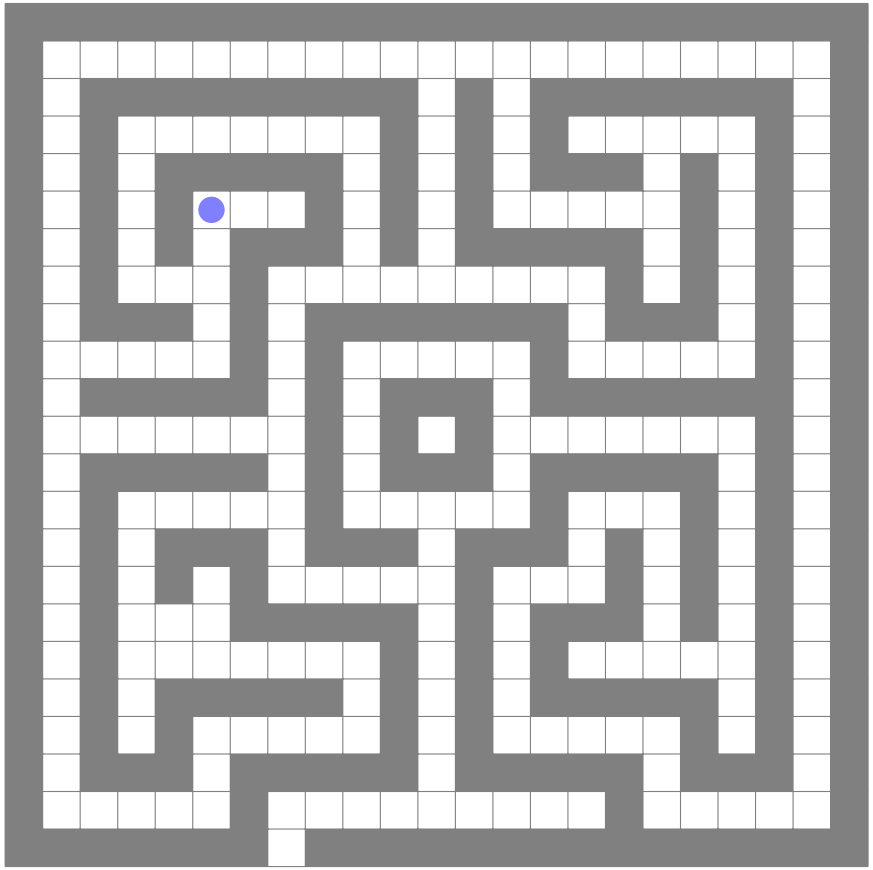
- consider problem of finding a way out of this maze starting from blue dot
- problem has applications in robotics, chemistry, medicine, physics
- closely related to many problems involving the exploration of possible paths to a solution



[4] inspired by `http://interactivepython.org/runestone/static/pythonds/Recursion/ExploringaMaze.html`

Navigation in a maze (1/2)

- the kind of maze we study is divided into squares
- each square of the maze is either
  - open (i.e. white) or
  - occupied by a section of wall (depicted as gray block).
- object in a maze (the blue dot) can move single step to
  - left
  - right
  - up
  - down
- but only if the square the move would lead to is not occupied



- blue dot above could only move down or right

Navigation in a maze (2/2)

Sketch of Algorithm

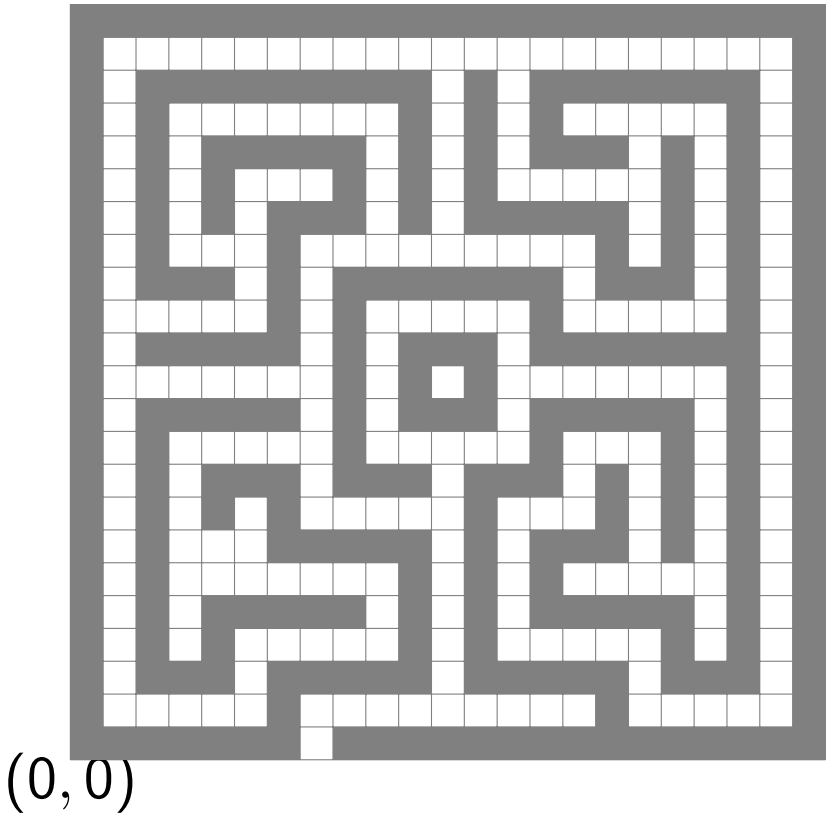
- we try the four possible directions one after the other
- after making the move, recursively continue from square we have reached
- if we have reached the exit with a move (and the corresponding recursive steps), we are done and do not continue with the other directions

- here is some pseudo code in python like syntax

```
def has_path2exit(square):
  if exit_square(square):
    return True
  else:
    return has_path2exit(square + (0,-1)) or    # down
           has_path2exit(square + (0,1)) or     # up
           has_path2exit(square + (-1,0)) or    # left
           has_path2exit(square + (1,0))        # right
```

Representing a maze by a dictionary

- to turn this idea into a working implementation, first consider how to represent a maze
- maze can be specified by a list of rows with an index for each occupied square
- number the rows from bottom to top and the columns from left to right
- origin $(0, 0)$ is at south west corner of maze of 23 rows and 23 columns
- row 0: $[0, 1, \ldots, 6, 8, \ldots, 22]$
- row 1: $[0, 6, 16, 22]$
- row 2: [0,2,3,4,6,...,10,12,...,16,18,19,20,22]



- represent maze by a dictionary `maze_rows` with row numbers as keys and values being lists of indexes of occupied squares

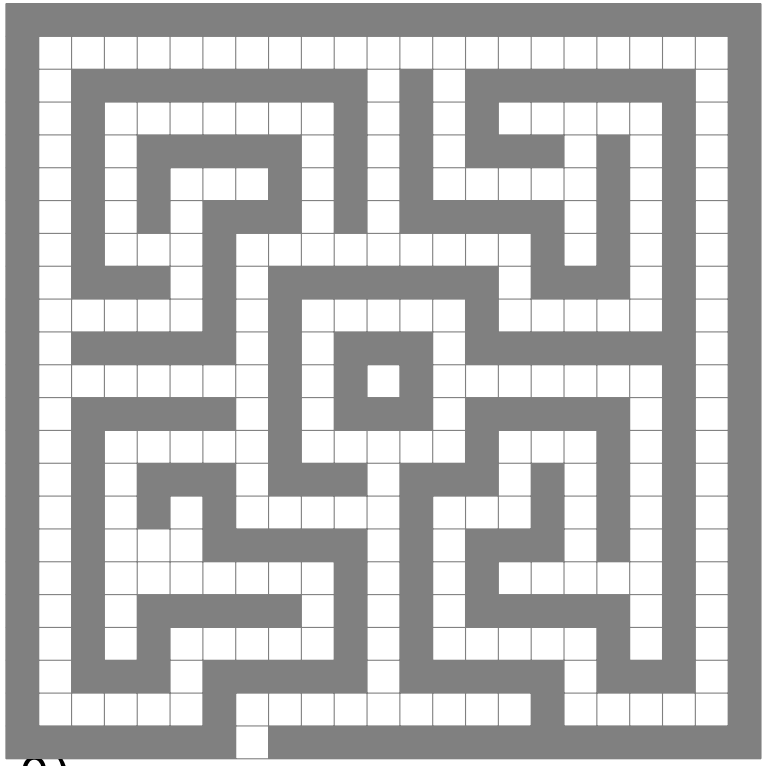
The Maze-class (1/5)

- `__init__` method is supplied with a dictionary `maze_rows` as described above
- it initializes instance variables storing the number of rows and columns of maze
- for finding paths in the maze, it is more convenient to represent the maze as a matrix with:
  - 1 for occupied squares
  - 0 for open squares

```
class Maze:
  def __init__(self,maze_rows):
    max_column = 0
    for rows in maze_rows.values():
      max_column = max(max_column,
                       max(rows))
    self._num_rows = len(maze_rows)
    self._num_cols = max_column + 1
    self._matrix = dict()
    self._rand_dir_ord = False
    for i in range(0,self._num_rows):
      self._matrix[i] = dict()
      for j in range(0,self._num_cols):
        if j in maze_rows[i]:
          self._matrix[i][j] = 1
        else:
          self._matrix[i][j] = 0
```

- see next frame for example of the constructed matrix

The Maze-class (2/5)



(0, 0)

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

(0, 0)

– the following method returns `True` iff the given square, identified by its row i and column-number j is immediately below, above, left or right of the maze

The Maze-class (3/5)

```
def isexit(self,square):
  i, j = square
  if i == -1 or i == self._num_rows or \
     j == -1 or j == self._num_cols:
    return True
  return False
```

The Maze-class (4/5)

- the next method returns the list of neighbor-squares for a given square
- as above, we use row and column-numbers to identify a square

```
def neighbors(self,square):
  i, j = square
  assert self._matrix[i][j] == 0
  neighbors_list = list()
  directions = [(0,-1),(0,1),(-1,0),(1,0)]
  if self._rand_dir_ord:
    rand_perm_fisher_yates(directions)
  for idiff, jdiff in directions:
    n_i = i + idiff
    n_j = j + jdiff
    if self.isexit((n_i,n_j)) or \
       (n_i >= 0 and n_i < self._num_rows and
           \
        n_j >= 0 and n_j < self._num_cols and
            \
        self._matrix[n_i][n_j] == 0):
      neighbors_list.append((n_i,n_j))
  return neighbors_list
```

- neighbor can be an exit square and all returned squares are open
- `rand_dir_ord` ⇒ random order of directions
- `not rand_dir_ord` ⇒ fixed order of directions (down, up, left, right)

The Maze-class (5/5)

- for test purposes it is useful to have the list of all open squares available
- this could be also be retrieved from the original `maze_rows` parameter
- but as we have not stored this, we have to iterate over all matrix entries and collect the open squares

```
def open_squares(self):
  squares = list()
  for i in range(0,self._num_rows):
    for j in range(0,self._num_cols):
      if self._matrix[i][j] == 0:
        squares.append((i,j))
  return squares
```

From the algorithm sketch to a recursive method (1/5)

- in the recursive algorithm sketched on frame 241 we only have considered a single base case, namely reaching the exit
- however, there is another base case:

possible base cases:

- we have found a square that has already been explored. To prevent an infinite loop, introduce a dictionary `mark` in which we mark the square as `EXPLORED` (i.e. by the value -2)
- we have passed an exit and have found the end of a path. So we mark the square as `EXIT` (i.e. by the value -1)

- in all recursive calls, if none of the base cases applies, we store in the dictionary `mark` the square leading to an exit

From the algorithm sketch to a recursive method (2/5)

- this method collects the path from the state dictionary

```
def collect_path(self,mark,startsquare):
  sq = startsquare
  path = [startsquare]
  while True:
    assert (sq in mark) and mark[sq] != EXPLORED
    next_sq = mark[sq]
    if next_sq == EXIT:
      break
    path.append(next_sq)
    sq = next_sq
  return path
```

- we now turn the previous pseudo code into a `Maze`-class-method `path2exit` which has the start square as parameter
- we make use of the fact that Python allows to declare local methods
- the local method `somepath2exit_rec` is visible only within `path2exit` and has access to the local variable `mark` storing the markings

From the algorithm sketch to a recursive method (3/5)

```
def somepath2exit(self,startsquare):
  mark = dict()
  def somepath2exit_rec(sq):
    if self.isexit(sq):
      mark[sq] = EXIT
      return 0
    if (sq in mark) and mark[sq] == EXPLORED:
      return -1
    mark[sq] = EXPLORED
    path_len = -1
    for next_sq in self.neighbors(sq):
      path_len = somepath2exit_rec(next_sq)
      if path_len >= 0:
        mark[sq] = next_sq
        path_len += 1
        break
    return path_len
  path_len = somepath2exit_rec(startsquare)
  if path_len < 0: return None
  return path_len, mark
```

From the algorithm sketch to a recursive method (4/5)

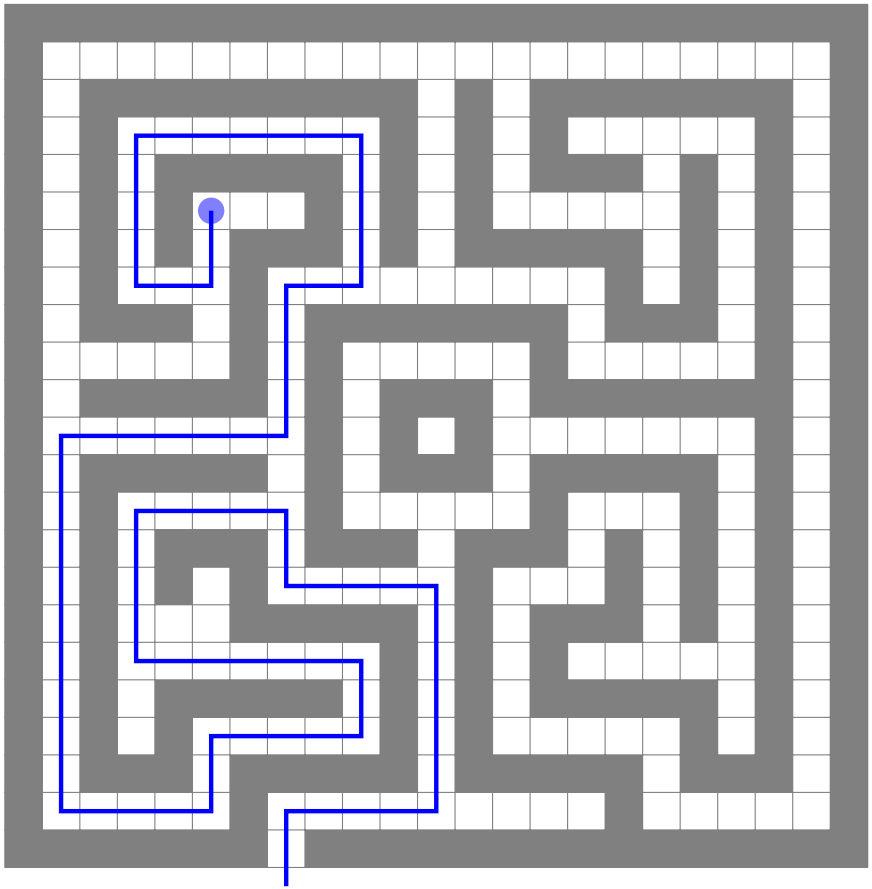
- the recursive method `somepath2exit_rec` returns
  - the length of the shortest path from the given square to an exit, or
  - `-1` if no such path exists.
- in each recursive call it is first checked, whether an exit was reached. If yes, then this is tracked in the `mark`-dictionary and `0` is returned
- if an open square has been explored before, then return `-1`
- otherwise, mark the square as explored and apply the recursive method to the neighbors, named `next_sq`, until a non-negative path length is returned
- for the first non-negative path length, store the value of `next_sq` in the dictionary `mark`, to later reconstruct a path and add 1 to the path length

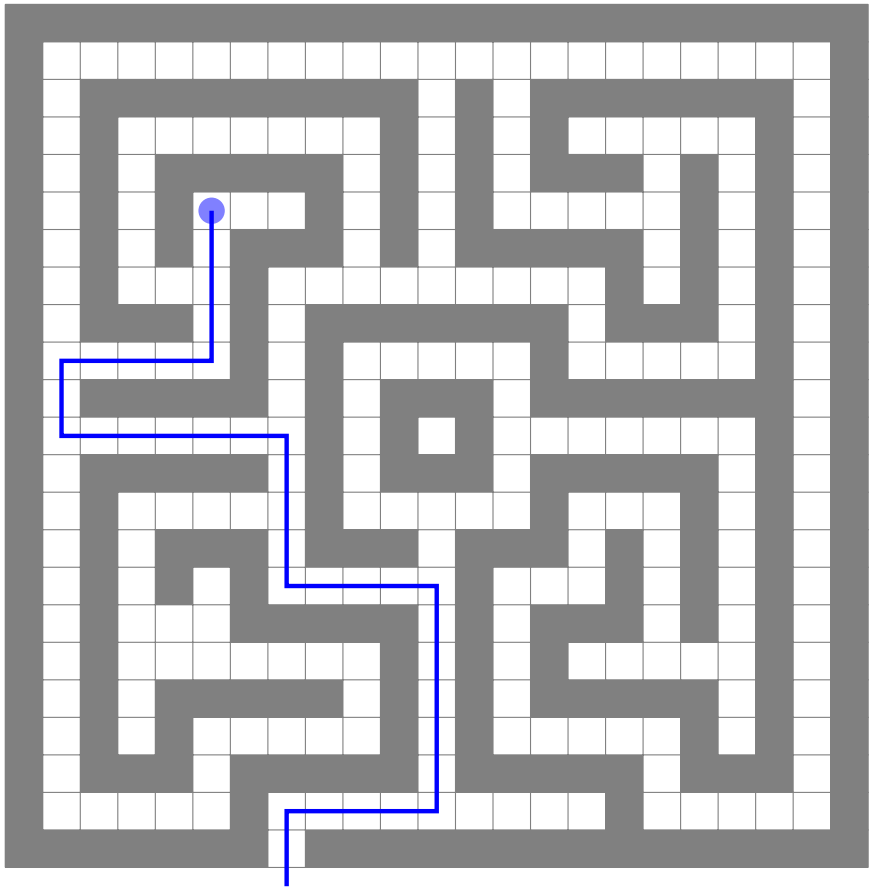
From the algorithm sketch to a recursive method (5/5)

- the method `somepath2exit` first computes the length of a path from the start square to an exit
- if such a path exists, then it also returns the dictionary `mark` which encodes a path
- the next frame (left side) shows the result of an application of `somepath2exit` and `collect_path` to reconstruct the path from the dictionary `mark`

An application of `somepath2exit`

- the left image shows the result of an application of `somepath2exit` for the start square $(17, 5)$ (depicted as blue dot)
- the path computed is shown by blue lines
- obviously, there is a shorter path (of length 37), shown on the right





- we now slightly modify the previous algorithm so that it computes a shortest path from a start square to an exit with high probability
- the idea is to randomize the order of directions to the neighbor square
- instead of a fixed order, we use a random permutation of the list `[(0,-1),(0,1),(-1,0),(1,0)]` of directions
- a permutation of a list is a list with exactly the same elements, but possibly in different order
- for a list of length n, there are $n!$ permutations

$\Rightarrow$ the list `directions` has $4 \cdot 3 \cdot 2 = 24$ permutations

- a random permutation is one of the possible permutations chosen at random
- the method `neighbors` already has a switch which calls a function `randompermutation` to permute the original list
- `randompermutation` will be implemented as part of an exercise

Iteratively calling somepath2exit (1/3)

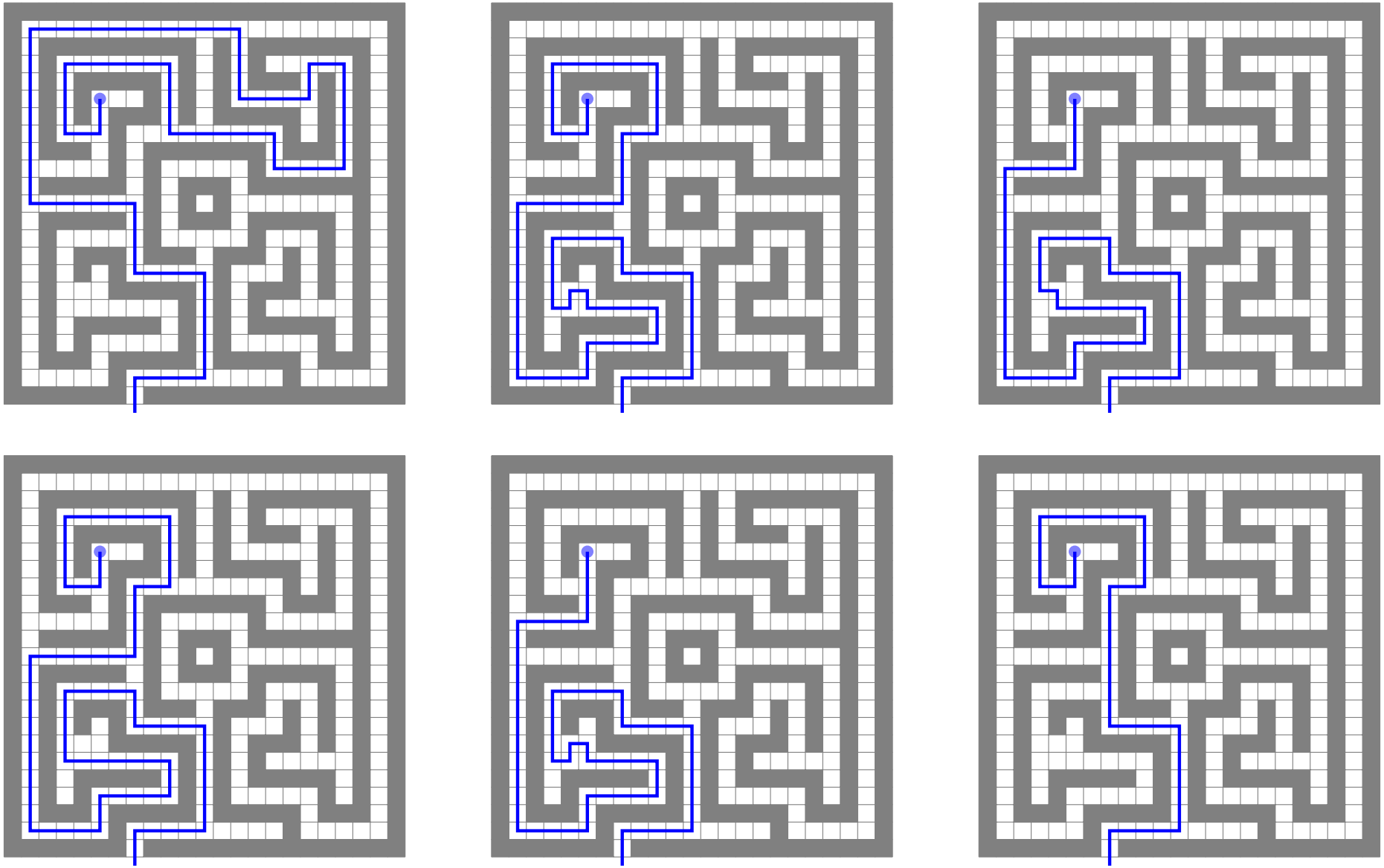
- the following method sets `self.rand_dir_ord` and calls `somepath2exit` a certain number of times, as specified by the parameter `iterations`
- it keeps track of the minimum length of a path seen so far and the corresponding dictionary `mark`
- finally, if at least one call to `somepath2exit` was successful, the marking of the minimum length path is turned into a path

```
def somepath2exit_iter(self,startsquare,iterations):
  self._rand_dir_ord = True
  min_path_len, min_mark = None, None
  for _ in range(iterations):
    found = self.somepath2exit(startsquare)
    if found:
      path_len, mark = found
      if min_path_len is None or min_path_len > path_len:
        min_path_len = path_len
        min_mark = mark
  if min_mark:
    return maze.collect_path(min_mark,startsquare)
  return None
```

Iteratively calling `somepath2exit` (2/3)

- the previous method is non-deterministic: for the same parameters, the result of the computation may be different in different calls
- as there are many different points with different choices, the number of paths is very large
- it is not guaranteed that after some number of iterations we get the shortest path
- for this reason, we used the notion of minimum length path above
- the following frame shows 6 different paths from square $(17, 5)$ to the exit for our standard maze, each obtained after 5 iterations
- none of the paths is the shortest one
- but with each iteration probability grows that shortest path is found
- e.g. using 10 iterations, in 50% of the runs we find the shortest path and in 27% of the runs the second shortest path

Iteratively calling `somepath2exit` (3/3)



Recursively computing a shortest path (1/3)

- we complete this section by showing how to compute a shortest path
- this is done in a less efficient, but simpler way
- the idea is to track two values:
  - the shortest path to an exit seen so far and
  - the current path consisting of all squares visited in the implicit recursion tree from the root to the current recursive call
- again we declare a local recursive method `shortest_path2exit_rec` which has the current path as argument, plus the square from which the search continues
- the length of the shortest path and the path itself are stored as list of two values, local to `shortest_path2exit` and visible for `shortest_path2exit_rec`
- if no successful path was seen before or the length of the shortest successful path is longer than the current path plus 1, then add the current square to the current path

Recursively computing a shortest path (2/3)

- moreover, iterate over the neighbors of the current square
- if neighbor square leads to an exit-square, a successful path was found and so update the shortest path
- otherwise, to prevent an infinite loop, we check that the neighbor square does not occur in the current path
- if this is the case, we compute the shortest path recursively from the neighbor square
- note that each recursive call leads to a different path and so we have to provide a copy of the current path for the current path parameter
- for the call of `shortest_path2exit_rec` we only have to provide the start square and an empty list for the current path parameter

Recursively computing a shortest path (3/3)

```
def shortest_path2exit(self,startsquare):
  spath = [None,None] # len and path as list
  def shortest_path2exit_rec(square,curr_path):
    i, j = square
    assert self._matrix[i][j] == 0
    if spath[0] is None or\
       spath[0] > len(curr_path) + 1:
      curr_path.append(square)
      for next_sq in self.neighbors(square):
        if self.isexit(next_sq):
          curr_path.append(next_sq)
          if spath[0] is None or\
             spath[0] > len(curr_path):
            spath[0] = len(curr_path)
            spath[1] = curr_path
        elif next_sq not in curr_path:
          new_path = curr_path.copy()
          shortest_path2exit_rec(next_sq,new_path)
  shortest_path2exit_rec(startsquare,list())
  return spath[1]
```

Iteratively computing a shortest path (1/2)

- for each recursive call, the previous method stores the values of the parameters `square` and `curr_path` on an implicit stack (first in, last out)
- this leads to a depth first traversal of the solution space
- we can make this stack explicit and use an iterative method instead
- the stack would store the same values, namely the current square and the current path
- instead of a stack we can also use a queue (first in, first out) for a breadth first traversal
- actually, we can implement both traversals in one method which has an additional parameter `df`
- a depth first traversal is performed, iff `df` is `True`
- instead of queue/stack, use list `tasks` to store tasks to be solved
- we use `pop` to extract an element from `tasks`
- DFS: extract last element of `tasks`; BFS: first element

Iteratively computing a shortest path (2/2)

```
def shortest_path2exit_itrtv(self,
                             startsquare,df):
  spath_len, spath = None, None
  tasks = [(startsquare,list())]
  while tasks:
    pop_idx = (len(tasks)-1) if df else 0
    square, curr_path = tasks.pop(pop_idx)
    i, j = square
    assert self._matrix[i][j] == 0
    if spath_len is None or\
       spath_len > len(curr_path) + 1:
      curr_path.append(square)
      for next_sq in self.neighbors(square):
        if self.isexit(next_sq):
          curr_path.append(next_sq)
          if spath_len is None or\
             spath_len > len(curr_path):
            spath_len = len(curr_path)
            spath = curr_path
        elif next_sq not in curr_path:
          new_path = curr_path.copy()
          tasks.append((next_sq,new_path))
  return spath
```

- breadth first method much faster than depth first method
- computing a shortest path for all 247 open squares of the example maze:
  BFS: 0.7 s
  DFS: 6.7 s

Displaying the results of path computations

- the result of computing a shortest path starting from $(17, 5)$ was displayed on frame 254, right image.
- all figures of this section are drawn by tikz[5], a programming language for generating figures
- tikz is a LaTeX-package, i.e. tikz-commands can be embedded in LaTeX-documents
- the tikz commands for all figures were generated by a method of class `Maze` (which was not shown here)
- the method turns the internal representation of the maze into appropriate commands for drawing a grid with gray squares at appropriate coordinates
- as their appearance in the maze is very regular, this is not too difficult
- the paths, i.e. list of squares delivered by the presented methods are transformed into tikz-commands so that they appear inside the maze

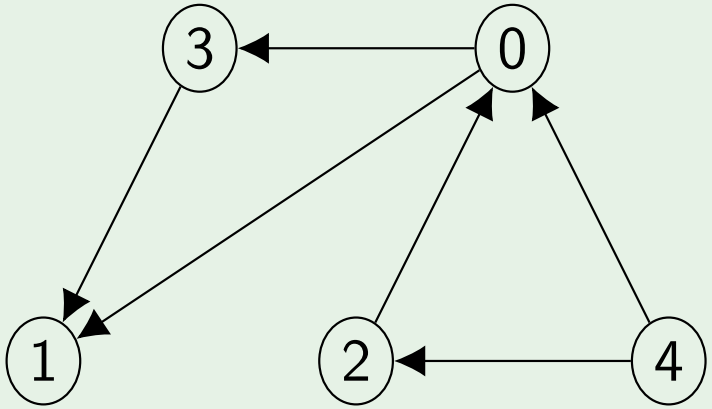
[5] `http://www.texample.net/tikz/examples/`

Definition

A graph consists of a set V of nodes (sometimes also called vertices) and a set $E \subseteq V \times V$ of edges.

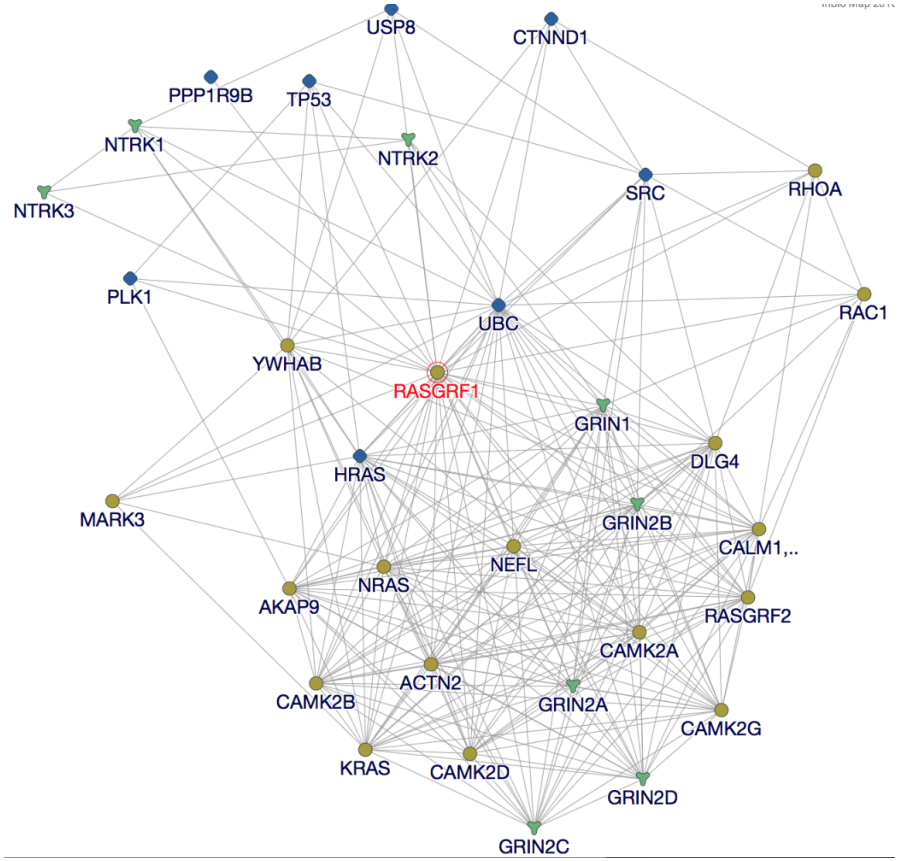
Example

The graph (V, E) with $V = \{0, 1, 2, 3, 4\}$ and $E = \{(2, 0), (0, 1), (0, 3), (3, 1), (4, 0), (4, 2)\}$ is usually drawn as follows (the placement of the nodes is arbitrary, i.e. it does not mean anything):



node labeled graph: nodes have labels (shown inside or besides node);
edge labeled graph: edges have labels (shown above or below edges);
graph is directed (the order of pairs in E matters); direction is expressed by arrows, i.e. $(x, y) \in E$ is written as $x \to y$

Figure: An undirected graph created by the web server InBioMap (`https://www.intomics.com`) when searching the term RASGRF1. A protein is represented by a node. Different node symbols represent different subcelluar locations. A protein-interaction is represented by an edge. An edge label (i.e. a confidence score for the corresponding interaction) becomes visible with a mouse-over event.



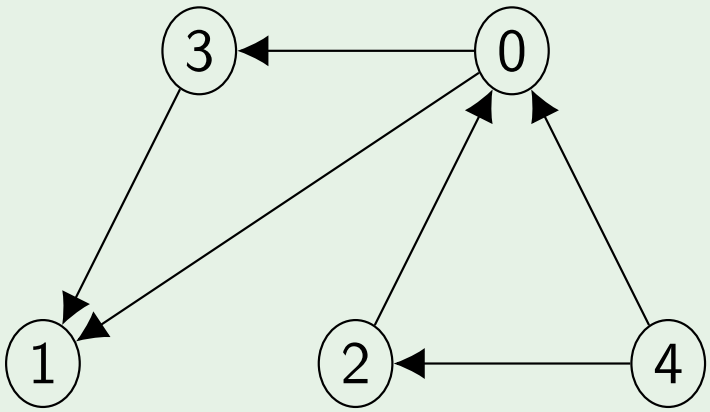
Definition

A *path* in a graph (V, E) is a sequence of nodes $v_0, v_1, \ldots, v_k$ for $k \geq 0$ such that for all i, $0 \leq i \leq k-1$ we have $(v_i, v_{i+1}) \in E$. An empty path satisfies $k = 0$, i.e. it has no edges and consists of a single node. A path starts with v_0 and ends with v_k and its length is k. So we state that it is a path from v_0 to v_k. Such a path is often written as

$$v_0 \to v_1 \to \cdots \to v_{k-1}$$

Example

Reconsider the directed graph (V, E) from Example 8.



$4 \to 2 \to 0 \to 1$
is a path of length 3 from 4 to 1

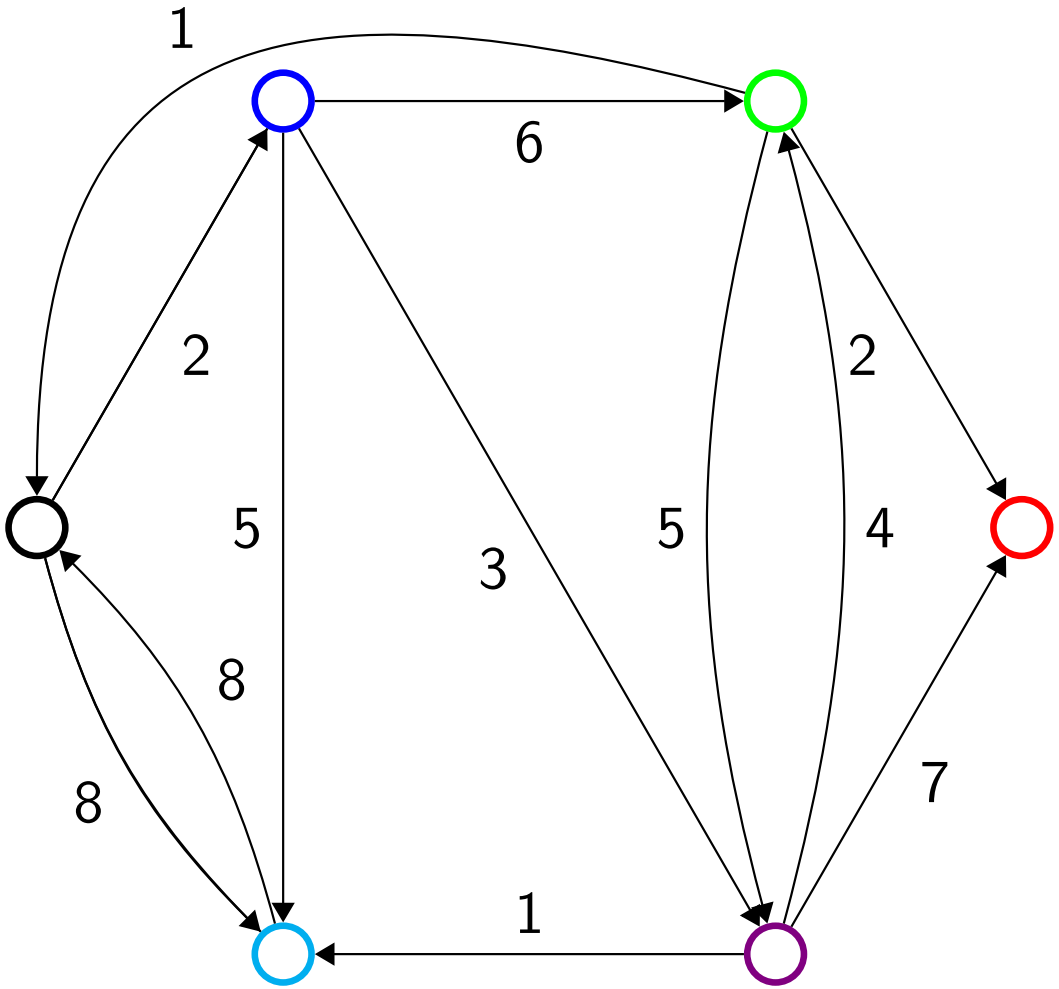
$4 \to 0 \to 3$
is a path of length 2 from 4 to 3

$3 \to 1$ is a path of length 1 from 3 to 1

$0 \to 1 \to 3$ is not a path, as $(1, 3) \notin E$

Example

Here is a directed graph with unlabeled nodes. To better distinguish the nodes, they are shown in different colors. The edges are labeled and represent a weight function $w : E \to \mathbb{R}$. .



such a graph can model:

nodes	edge labels
airports	flight times
social network users	1 = has communicated
currencies	exchange rates
strings	costs of edit operations
genes	interaction level

– for an edge labeled graph it makes sense to define the weight of a path by adding up the weights of the edges it consists of

Definition

Consider a graph (V, E) with weight function $w : E \to \mathbb{R}$. For any path $p = v_0 \to v_1 \to \cdots \to v_k$, we define the weight $w(p)$ of p by

$$w(p) = \sum_{i=0}^{k-1} w(v_i \to v_{i+1})$$

– a path of length 0 has weight 0

– when there is more than one path between two nodes, one is usually interested in the path of minimum weight

– this leads to an important optimization problem:

Definition

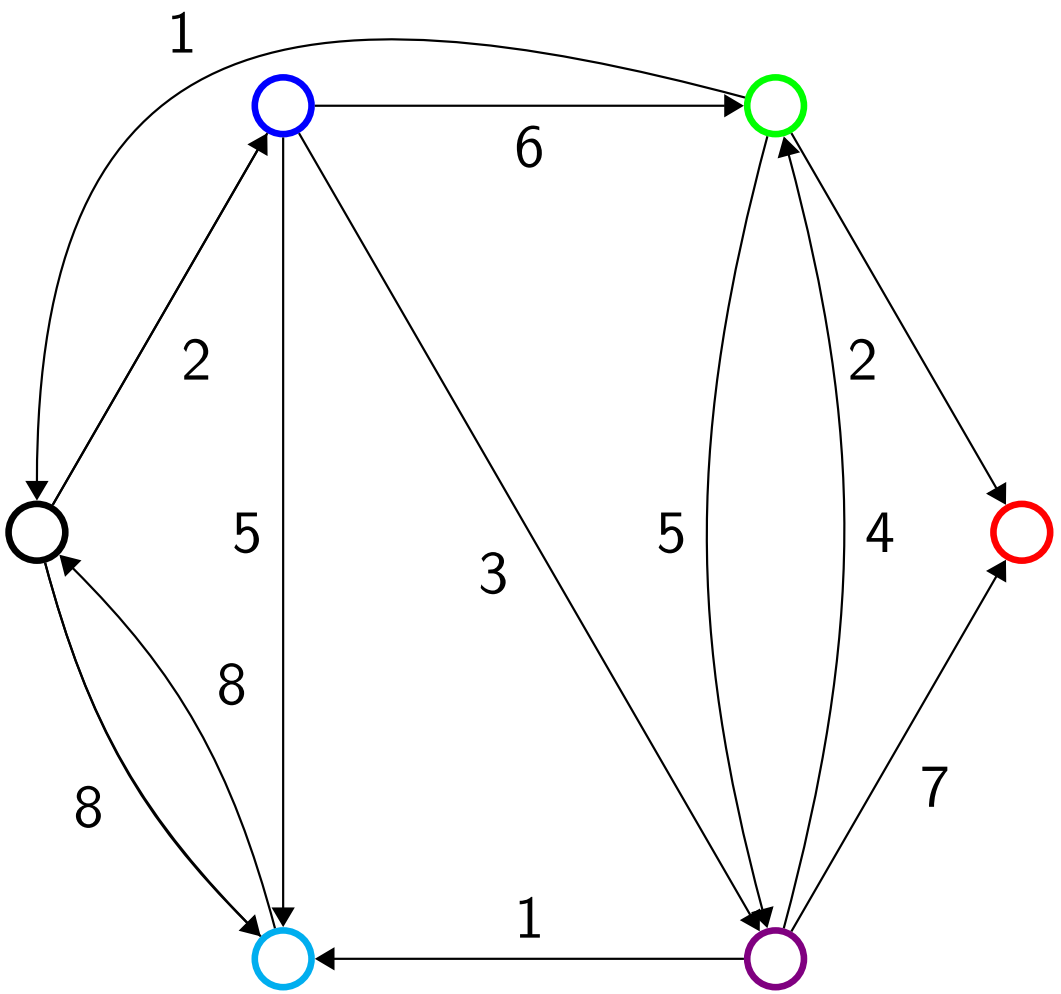
Let $G = (V, E)$ be a graph with edge weights $w : E \to \mathbb{R}$. Let $s, x \in V$ and define

$$\delta(s, x) = \begin{cases} \min\{w(p) \mid p \text{ is a path from } s \text{ to } x\} & \text{a path from } s \\ & \text{to } x \text{ exists} \\ \infty & \text{otherwise} \end{cases}$$

The *shortest-path problem* for s and x consists of computing $\delta(s, x)$ and a path p from s to x such that $w(p) = \delta(s, x)$. Such a path is called shortest or optimal path from s to x.

Example

Reconsider the graph from Example 11.



clockwise numbering of nodes from 0 to 5; start node s (0, black)

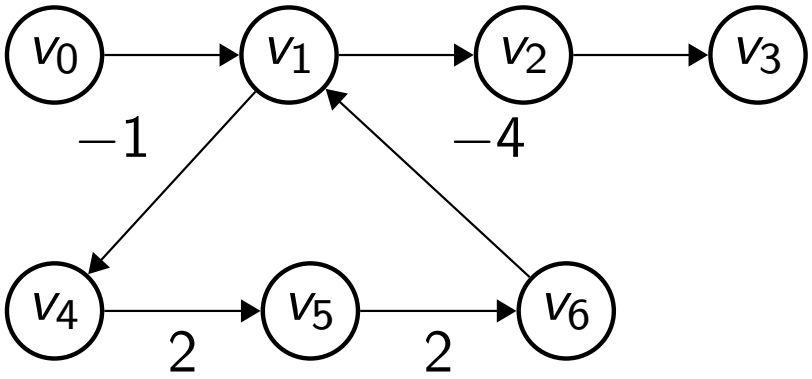
x	$\delta(s,x)$	shortest path
0	0	0
1	2	$0 \to 1$
2	8	$0 \to 1 \to 2$
3	10	$0 \to 1 \to 2 \to 3$
4	5	$0 \to 1 \to 4$
5	6	$0 \to 1 \to 4 \to 5$

Existence of shortest path

- suppose there is a path between s and u in graph G
- does a shortest path always exist?

Existence of shortest path

- suppose there is a path between s and u in graph G
- does a shortest path always exist?
- no: consider graph with a node x and non-empty path
  - from x to x (cycle),
  - with negative weight,
  - crossing the path from s to u
- then one can add additional cycles around x and decrease the weight
- any shortest path can be made shorter
- here is an example with $s = v_0$, $x = v_1$, and $u = v_3$



graph with negative weights appear e.g. in chemistry:

- compounds: nodes
- reaction: edge
- energy consumed ($\mathbb{R}_-$) or produced ($\mathbb{R}_+$): edge label

Non-negative weights and optimal subpaths (1/2)

- in many case, we can shift weights (by adding $-\min\{w(e) \mid e \in E\}$)
- so we restrict to the case of non-negative weights
- in this case we can use a simple algorithm, called Dijkstra's algorithm
- the key to the algorithm is the fact that any subpath of a shortest path has minimum weight

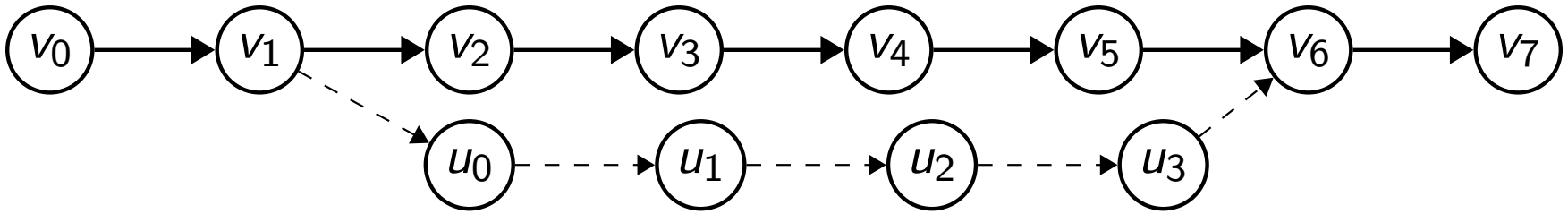
Theorem

Consider a shortest path from s to u which includes a path from x to y. Then this path is a shortest path from x to y.

this section on Dijkstra's algorithm follows `https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-17-shortest-paths-i-properties-dijkstras-algorithm-breadth-first-search/`

Non-negative weights and optimal subpaths (2/2)

- we do not give a formal proof, but instead consider an example with a shortest path $p = v_0 \to v_1 \to v_2 \to \ldots \to v_5 \to v_6 \to v_7$:



- now look at the subpath $p' = v_1 \to v_2 \to \ldots \to v_5 \to v_6$
- assume that this is not the shortest path from v_1 to v_6 $(*)$
- so there exists a path p'' from v_1 to v_6, say along the dashed edges, such that $w(p'') < w(p')$
- but then we could replace the subpath p' in p by p'' and obtain a path from v_0 to v_7 with total weight smaller than $w(p)$
- as p is the shortest path, this is not possible

$\Rightarrow$ assumption $(*)$ was wrong, i.e. the subpath p' is a shortest path

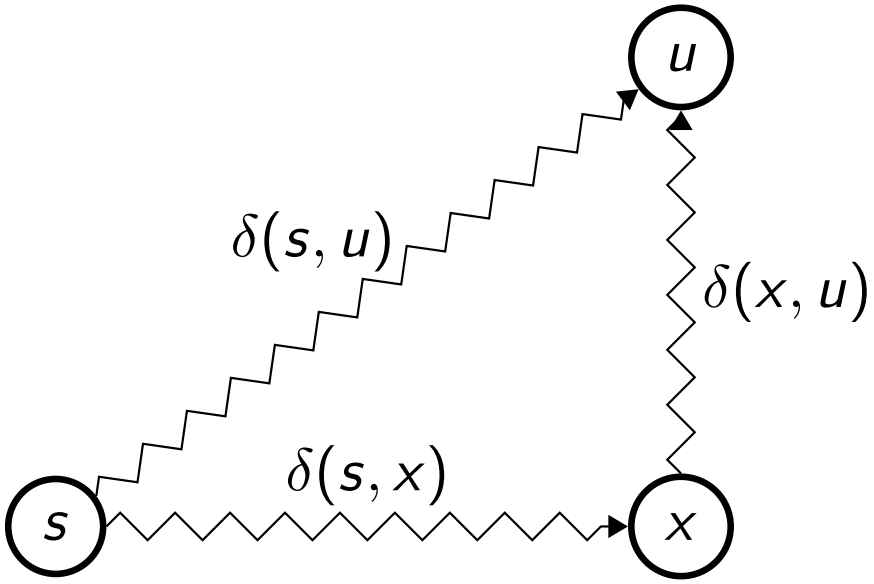
Triangle inequality

- the *optimality of subpath*-property is based on a general property which holds for many mathematical structures: the triangle inequality

Theorem

For any nodes s, u, $x \in V$ we have $\delta(s,u) \leq \delta(s,x) + \delta(x,u)$.

- again, we do not give a formal proof, but just consider the following picture, in which the zigzag-lines represent possibly long paths involving many nodes



- if we have a shortest path from s to u, any path from s to u involving some particular node x will not be shorter, even if the path from s to x and from x to u are shortest path
- so formally $\delta(s,u) \leq \delta(s,x) + \delta(x,u)$

Single source shortest path problem

- recall that the shortest path problem we had considered involves two particular node s and u
- so lets denote it by $PathP(s, u)$, where $PathP$ stands for path problem
- a generalization of this is the single source shortest path problem:

  for a given node s (the source node) determine $\delta(s, x)$ (and shortest path from s to x) <u>for all</u> $x \in V$

- let us denote this problem by $PathP(s, V)$
- with the most efficient algorithms known today (i.e. Dijkstra's algorithm), it is (in general) not easier to solve $PathP(s, u)$ for a particular $u \in V$ than to solve $PathP(s, V)$
- so consider how to solve $PathP(s, V)$
- if we have a solution for this problem, we can lookup the solution for a particular target node u

Dijkstra's algorithm (1/6)

- the main idea of Dijkstra's algorithm is to maintain a set U as well as two function $d : V \to \mathbb{R}$ and $pred : V \setminus \{s\} \to V$ such that
  - at any time, U is the set of nodes $x \in V$ such that
    - there is a path from s to x
    - all edges outgoing from x have been processed
  - for all $y \in V$, $d(y) \geq \delta(s, y)$, i.e. $d(y)$ is an upper bound on the weight of the shortest path from s to y
  - for all $y \in V$, if $pred(y) = x$ for some $x \in V$, then $x \in U$ and there is a path from s to y of weight $d(y)$ ending with edge $(x, y) \in E$.
- function *pred* allows to reconstruct shortest paths in reverse order
- the specification of Dijkstra's algorithm consists of three parts, the initialization step, the iteration step and the termination condition, all described in the following frames

Dijkstra's algorithm (2/6)

Initialization

Assignment	Rationale
set $U = \emptyset$	no nodes have been processed yet
set $d(s) = 0$	the empty path of weight 0 is the shorted path from s to s
set $d(x) = \infty$ for all $x \neq s$	we have no better estimates, as we have seen no edges yet
set $pred(y) = \perp$ for all $y \in V \setminus \{s\}$ ($\perp$ stands for undefined)	no edges have been processed

Dijkstra's algorithm (3/6)

Iteration

- in each step of the algorithm, choose some $x \in V \setminus U$ such that $d(x) \leq d(x')$ for all $x' \in V \setminus U$ (greedy strategy)
- all edges $(x, y) \in E$ are processed as follows:
  - if $d(x) + w(x, y) < d(y)$, set $d(y) = d(x) + w(x, y)$ and $pred(y) = x$ (relaxation step)
- add x to U

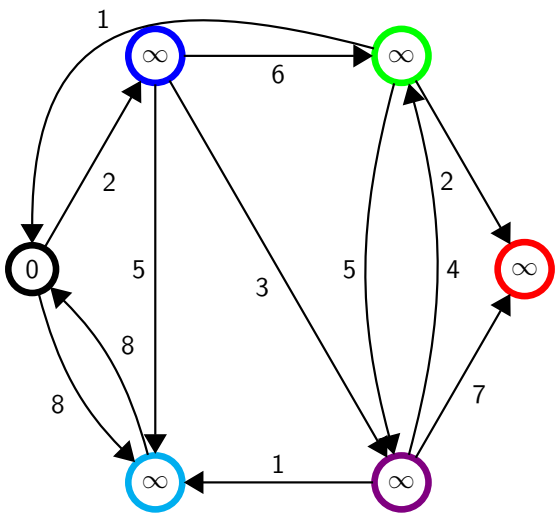
Dijkstra's algorithm (4/6)

Rationale for relaxation step

- in the relaxation step, the algorithm implicitly adds the edge $x \to y$ to the end of a path from s to x of weight $d(x)$, thus constructing a path from s to y of weight $d(x) + w(x, y)$
- if before this relaxation step, y has not been reached, $d(y) = \infty$ holds
- if before this relaxation step, y has been reached, $d(y) \neq \infty$ and $d(y)$ is the best estimate of $\delta(s, y)$ that we could deduce from previously considered paths
- in any case we know that there is a path from s to y with smaller weight $d(s) + w(x, y)$ (compared to the current value $d(y)$) and thus updating makes sense

Dijkstra's algorithm (5/6)

state of algorithm after initialization:



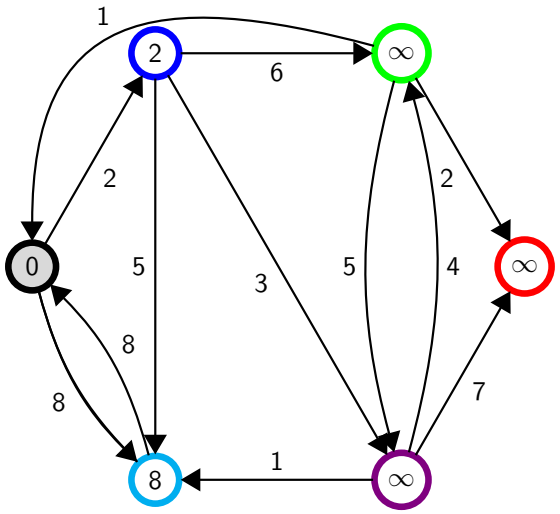
each node x is colored and labelled by $d(x)$

values of d for $x \in V \setminus U$ (in sorted order):

$0 \mapsto 0$, $1 \mapsto \infty$, $2 \mapsto \infty$, $3 \mapsto \infty$, $4 \mapsto \infty$, $5 \mapsto \infty$

node	0	1	2	3	4	5
pred	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

state of algorithm after processing node 0 (black), i.e. $U = \{0\}$:



each node x is colored and labelled by $d(x)$;

all nodes $x \in U$ are shown in grey;

values of d for $x \in V \setminus U$ (in sorted order):

$1 \mapsto 2$, $2 \mapsto 8$, $3 \mapsto \infty$, $4 \mapsto \infty$, $5 \mapsto \infty$

node	0	1	2	3	4	5
pred	$\perp$	0	$\perp$	$\perp$	$\perp$	0

Dijkstra's algorithm (6/6)

Termination

- the algorithm terminates, if for all nodes $x \in V \setminus U$ it holds $d(x) = \infty$
- then none of the nodes in $V \setminus U$ are reachable from s and so $\delta(s, x) = \infty = d(x)$ for all $x \in V \setminus U$
- for any $x \in U \setminus \{s\}$ there is a non-empty path from s to x, $d(x) = \delta(s, x)$ and $pred(x)$ is the predecessor of x in a shortest path from s to x
- as subpaths are shortest paths, one can construct the shortest paths by tracing back using the values in function *pred*

- we have described U as the set of nodes to which we add x after processing all outgoing edges from x
- an important property[6] is that for all $x \in U$, we have $d(x) = \delta(s, x)$, i.e. $d(x)$ is not just an estimate, but the final value of $\delta(s, x)$.
- so the relaxation step is only applied to nodes which are not in U

[6] a proof of this property can be found in Cormen et. al. Introduction to Algorithms, The MIT Press, 2009.

Implementation of Graphs (1/8)

- we now consider how to implement Dijkstra's algorithm in Python
- of course, we first have to implement graphs
- here we follow the definition and implement three classes, a class for nodes, a class for edges and a class representing the graph
- a node is implemented by class `Node`
- besides a label it maintains a unique integer identifier (id for short) by assigning consecutive integers using the class variable `_number`
- with the constructor, we can specify whether we want to output the node as colored circle in tikz, the graph drawing language embedded in LaTeX
- the possible color names are stored in a globally accessible list named `COLOR_MAP`

Implementation of Graphs (2/8)

```
COLOR_MAP = ['black','blue','green','red','violet',
             'cyan','yellow','orange','brown','magenta']

class Node:
  _number = 0
  def __init__(self, label, has_color=True):
    if has_color:
      assert Node._number < len(COLOR_MAP), \
             'not enough colors, so extend COLOR_MAP'
    self._label = label
    self._id = Node._number
    Node._number += 1
    self._has_color = has_color
```

Implementation of Graphs (3/8)

- the methods provided allow to extract a label and to set it
- we also specify how to hash a node to efficiently compare it

```
def set_label(self, newlabel):
  self._label = newlabel

def label(self):
  return self._label

def __hash__(self):
  return self._id
```

Implementation of Graphs (4/8)

- the following class represents edges of a directed graph
- for an edge $(s, u) \in E$ we specify the id `from_id` of the source node s and `to_id` of the target node u
- a label and a mode for the edge can optionally be supplied, where the mode specifies formatting instructions when drawing the graph in tikz.

```
class Edge:
  def __init__(self, from_id, to_id, label=None, mode=None):
    self._from_id = from_id
    self._to_id = to_id
    self._label = label
    # code for handling mode not shown

  def to_node(self):
    return self._to_id
  def from_node(self):
    return self._from_id
  def __eq__(self, other):
    return self._from_id == other._from_id and \
           self._to_id == other._to_id
```

Implementation of Graphs (5/8)

- two edges are considered equal, when the ids of the nodes on both ends are pairwise identical
- as usual, we provide a getter and a setter-method for edges

```
def label(self):
  return self._label
def set_label(self, newlabel):
  self._label = newlabel
```

- for representing a graph we introduce the following class
- for a graph we keep track of a list of nodes, a list of edges and a dictionary of lists named `_adjacence_lists`
- for key i in this dictionary we store the indexes of all edges outgoing from the node with id i
- such indexes refer to the list `_edges` of edges.
- with the constructor, we can specify of whether all nodes are shown as colored circle in tikz

Implementation of Graphs (6/8)

```
class Graph:
  def __init__(self,diameter,has_color=True):
    self._nodes = list()
    self._edges = list()
    self._adjacence_lists = defaultdict(list)
    self._has_color = has_color
```

- `diameter` is a parameter influencing the graph layout in tikz

- nodes are created in the method `add_node` and appended to the appropriate list
- for creating a node, one must provide a label
- before an edge is added to the list the index in `_edges` is determined, where the edge will be stored
- this index is appended to the adjacence list of the source node of the edge
- besides the getter-methods for nodes and edges, we implemented a method `edge2neighbors` to return a list of edges outgoing from a given node

Implementation of Graphs (7/8)

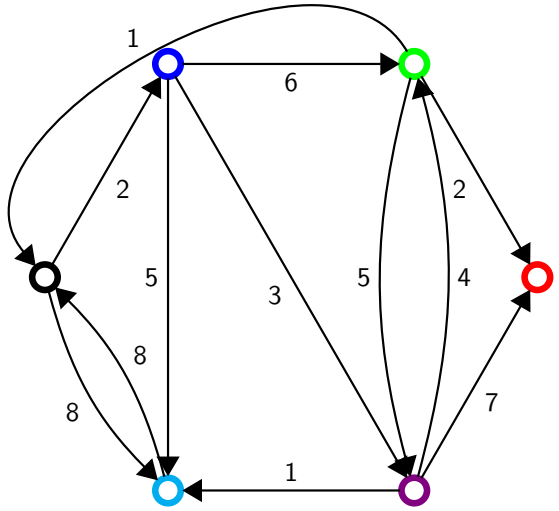
– this list is constructed by first accessing the appropriate entry in `_adjacence_lists` and then converting the list of edge indexes into edges

```
def add_node(self,label):
  self._nodes.append(Node(label,self._has_color))

def add_edge(self, edge):
  current_idx = len(self._edges)
  self._edges.append(edge)
  self._adjacence_lists[edge.from_node()].append(current_idx)

def get_edges(self):
  return self._edges

def get_nodes(self):
  return self._nodes

def edge2neighbors(self,node):
  adjacence_list = self._adjacence_lists[hash(node)]
  return [self._edges[edge_num] for edge_num in adjacence_list]
```

Implementation of Graphs (8/8)

```
def build_example_graph():
  br = ['bend right=15']
  in_out = ['out=120','in=130']
  edges = [Edge(0, 1, 2),
           Edge(0, 5, 8, br),
           Edge(1, 2, 6),
           Edge(1, 4, 3),
           Edge(1, 5, 5),
           Edge(2, 0, 1, in_out),
           Edge(2, 3, 2),
           Edge(2, 4, 5, br),
           Edge(4, 2, 4, br),
           Edge(4, 3, 7),
           Edge(4, 5, 1),
           Edge(5, 0, 8, br)]
  max_id = max([max(edge.from_node(),\
                    edge.to_node()) \
                for edge in edges])
  graph = Graph('35mm')
  for _ in range(max_id+1):
    graph.add_node(None)
  for edge in edges:
    graph.add_edge(edge)
  return graph
```

- here is an example defining the graph shown below



- we specify the list of edges with optional formatting
- from the source/target indexes of the edges we determine the maximum node id and use this to add unlabelled nodes

Implementation of Priority Queues (1/5)

- in the specification of Dijkstra's algorithm the set U of nodes and the function d play a central role
- recall, that in the outer loop of the algorithm, we have to determine a node $x \in V \setminus U$ such that $d(x) = \min\{d(x') \mid x' \in V \setminus U\}$
- instead of storing the set U, we store all nodes of $V \setminus U$ in a data structure in which they are prioritized by their d-value
- such a data structure is called *priority queue* where the priority for a key (i.e. node number) is its d-value
- for our application such a priority queue must provide the following methods:
  - construct an empty priority queue
  - check if the queue is empty
  - check if the queue contains a node
  - extract the key (i.e. node) with minimum d-value from the queue
  - set the value for a key (which may or may not already exist in the queue)

Implementation of Priority Queues (2/5)

- there are implementations of priority queues based on heaps such that the first operations run in constant time while the two last operations require $O(\log n)$ time if n is the number of nodes
- for Dijkstra's algorithm we extract each node once from the queue and do not reinsert it later
- so the number of extractions is equals the number nodes of the graph
- the initial values for each node are set once and each edge of the graph may trigger updating the value for a node
- so the number of `set_value`-operations is at most $|V| + |E|$
- as the number of edges can be on the order of $|V|^2$, this would lead to an implementation which runs in $O(|V|^2 \log |V|)$
- we use an implementation based on dictionaries in which the first three and the last operation run in constant time and the extract operation runs in linear time $\Rightarrow$ running time is $O(|V|^2)$.
- the class implementing the priority queue is shown on the next frame

Implementation of Priority Queues (3/5)

```
class PriorityQueue:
  def __init__(self):
    self._dict = dict()

  def is_empty(self):
    return len(self._dict) == 0

  def __contains__(self, key):  # overload in operator
    return key in self._dict

  def set_priority(self, key, priority):
    self._dict[key] = priority

  def extract_min(self):
    prio_max, key_max = None, None
    for key, value in self._dict.items():
      if prio_max is None or value < prio_max:
        key_max, prio_max = key, value
    assert key_max is not None
    del self._dict[key_max]
    return key_max
```

Implementation of Priority Queues (4/5)

- to represent the value ∞ in Dijkstra's algorithm, we use a constant `INFTY` defined as `sys.maxsize`
- while the priority queue only stores $d(x)$ for all $x \in V \setminus U$, we store $d(x)$ for all $x \in V$ as label of node x (possibly ∞) in the graph
- this leads to some redundancies, but does not require to store the set U
- the set U is implicit, namely the set of nodes not represented in the priority queue
- the function `pred` is implemented by a list of length $|V|$ with keys and values being node ids (undefined values are represented by `None`)
- so to set the predecessor for a target node `y` and a source node `x` we use the assignment `pred(hash(y))= hash(x)`

Implementation of Priority Queues (5/5)

```
1  def dijkstra(graph, start_node):
2    nodes = graph.get_nodes()
3    pred = [None] * len(nodes)
4    pq = PriorityQueue()
5    for node in nodes:
6      pq.set_priority(node, INFTY)
7      node.set_label(INFTY)
8    pq.set_priority(start_node, 0)
9    start_node.set_label(0)
10   while not pq.is_empty():
11     x = pq.extract_min()
12     for edge in graph.edge2neighbors(x):
13       y = nodes[edge.to_node()]
14       if y in pq:
15         new_dy = x.label() + edge.label()
16         if new_dy < y.label():
17           y.set_label(new_dy)
18           pq.set_priority(y,new_dy)
19           pred[hash(y)] = hash(x)
20   return pred
```

- final slides show how the code lines lead to updates of the graph, of `pq` and `pred`

2-9 initialization

10-19 iteration

11 extract node `u` with minimum d-value

12 enumerate edges outgoing from `u`

13 obtain target node for current edge

14 nodes $\neg$ in `pq` have final d-value $\Rightarrow$ relaxation only for nodes in `pq`

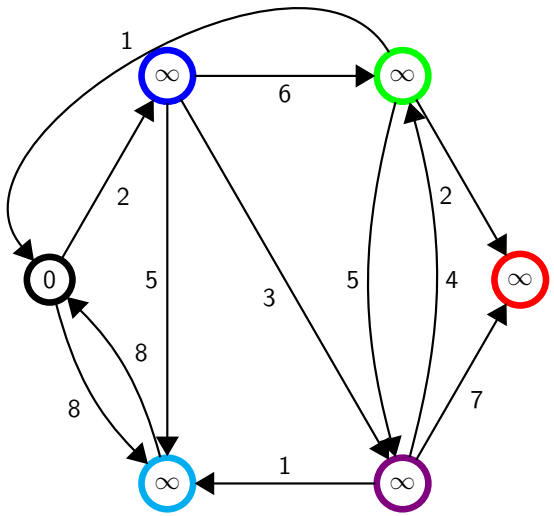
15 determine weight of path from start node to y with last edge from x to y

16 relaxation possible?

17-19 relaxation

pq: 0(0), 1(∞), 2(∞), 3(∞), 4(∞), 5(∞)

nodes	0	1	2	3	4	5
pred	⊥	⊥	⊥	⊥	⊥	⊥

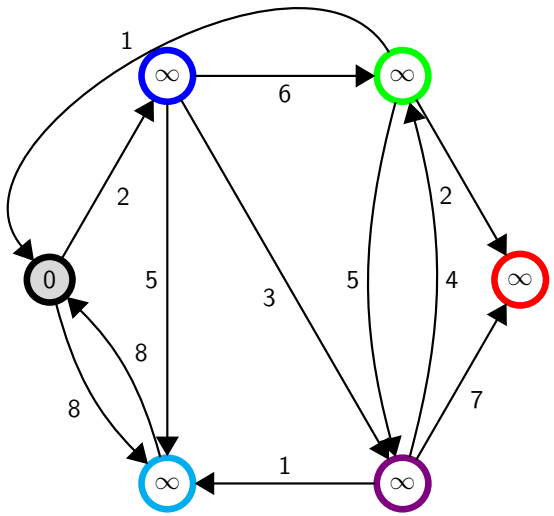



```
def dijkstra(graph, start_node):
  nodes = graph.get_nodes()
  pred = [None] * len(nodes)
  pq = PriorityQueue()
  for node in nodes:
    pq.set_priority(node, INFTY)
    node.set_label(INFTY)
  pq.set_priority(start_node, 0)
  start_node.set_label(0)
  while not pq.is_empty():
    x = pq.extract_min()
    for edge in graph.edge2neighbors(x):
      y = nodes[edge.to_node()]
      if y in pq:
        new_dy = x.label() + edge.label()
        if new_dy < y.label():
          y.set_label(new_dy)
          pq.set_priority(y,new_dy)
          pred[hash(y)] = hash(x)
  return pred
```

grey: current node
grey: current edge
dark grey: final node

pq: 1(∞), 2(∞), 3(∞), 4(∞), 5(∞)

nodes	0	1	2	3	4	5
pred	⊥	⊥	⊥	⊥	⊥	⊥

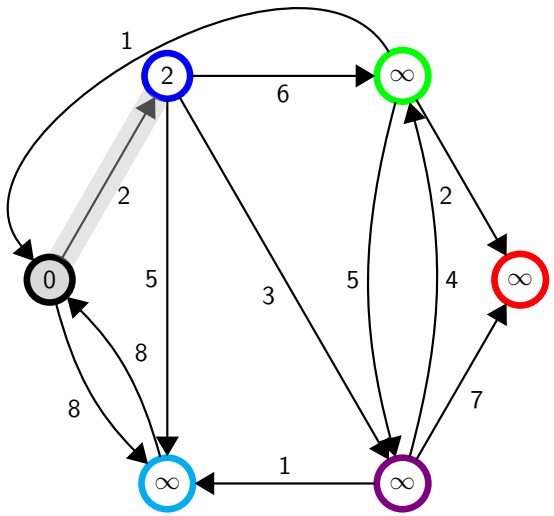



```
def dijkstra(graph, start_node):
  nodes = graph.get_nodes()
  pred = [None] * len(nodes)
  pq = PriorityQueue()
  for node in nodes:
    pq.set_priority(node, INFTY)
    node.set_label(INFTY)
  pq.set_priority(start_node, 0)
  start_node.set_label(0)
  while not pq.is_empty():
    x = pq.extract_min()
    for edge in graph.edge2neighbors(x):
      y = nodes[edge.to_node()]
      if y in pq:
        new_dy = x.label() + edge.label()
        if new_dy < y.label():
          y.set_label(new_dy)
          pq.set_priority(y,new_dy)
          pred[hash(y)] = hash(x)
  return pred
```

grey: current node
grey: current edge
dark grey: final node

pq: 1(2), 2(∞), 3(∞), 4(∞), 5(∞)

nodes	0	1	2	3	4	5
pred	⊥	0	⊥	⊥	⊥	⊥

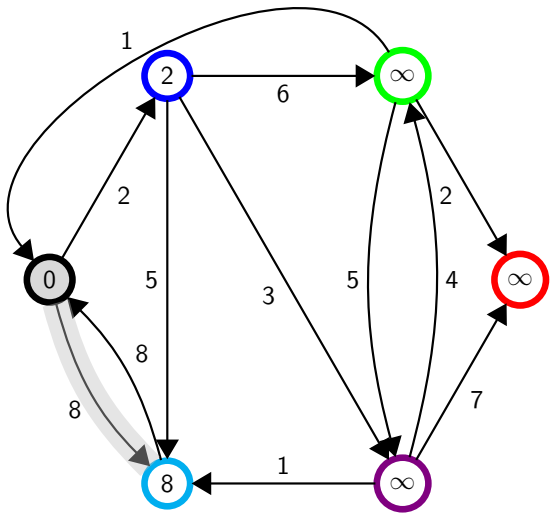



```
def dijkstra(graph, start_node):
  nodes = graph.get_nodes()
  pred = [None] * len(nodes)
  pq = PriorityQueue()
  for node in nodes:
    pq.set_priority(node, INFTY)
    node.set_label(INFTY)
  pq.set_priority(start_node, 0)
  start_node.set_label(0)
  while not pq.is_empty():
    x = pq.extract_min()
    for edge in graph.edge2neighbors(x):
      y = nodes[edge.to_node()]
      if y in pq:
        new_dy = x.label() + edge.label()
        if new_dy < y.label():
          y.set_label(new_dy)
          pq.set_priority(y,new_dy)
          pred[hash(y)] = hash(x)
  return pred
```

grey: current node
grey: current edge
dark grey: final node

pq: 1(2), 5(8), 2(∞), 3(∞), 4(∞)

nodes	0	1	2	3	4	5
pred	⊥	0	⊥	⊥	⊥	0

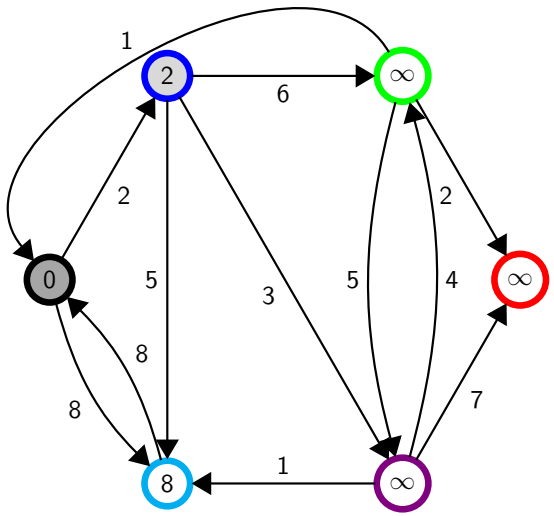



```
def dijkstra(graph, start_node):
  nodes = graph.get_nodes()
  pred = [None] * len(nodes)
  pq = PriorityQueue()
  for node in nodes:
    pq.set_priority(node, INFTY)
    node.set_label(INFTY)
  pq.set_priority(start_node, 0)
  start_node.set_label(0)
  while not pq.is_empty():
    x = pq.extract_min()
    for edge in graph.edge2neighbors(x):
      y = nodes[edge.to_node()]
      if y in pq:
        new_dy = x.label() + edge.label()
        if new_dy < y.label():
          y.set_label(new_dy)
          pq.set_priority(y,new_dy)
          pred[hash(y)] = hash(x)
  return pred
```

grey: current node
grey: current edge
dark grey: final node

pq: 5(8), 2(∞), 3(∞), 4(∞)

nodes	0	1	2	3	4	5
pred	⊥	0	⊥	⊥	⊥	0

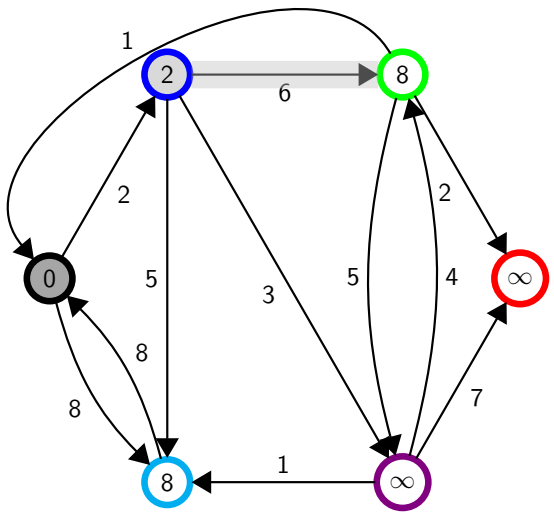



```
def dijkstra(graph, start_node):
  nodes = graph.get_nodes()
  pred = [None] * len(nodes)
  pq = PriorityQueue()
  for node in nodes:
    pq.set_priority(node, INFTY)
    node.set_label(INFTY)
  pq.set_priority(start_node, 0)
  start_node.set_label(0)
  while not pq.is_empty():
    x = pq.extract_min()
    for edge in graph.edge2neighbors(x):
      y = nodes[edge.to_node()]
      if y in pq:
        new_dy = x.label() + edge.label()
        if new_dy < y.label():
          y.set_label(new_dy)
          pq.set_priority(y,new_dy)
          pred[hash(y)] = hash(x)
  return pred
```

grey: current node
grey: current edge
dark grey: final node

pq: 2(8), 5(8), 3(∞), 4(∞)

nodes	0	1	2	3	4	5
pred	⊥	0	1	⊥	⊥	0

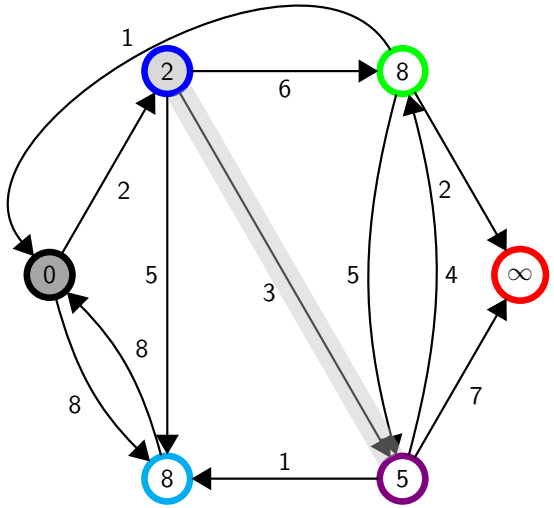



```
def dijkstra(graph, start_node):
  nodes = graph.get_nodes()
  pred = [None] * len(nodes)
  pq = PriorityQueue()
  for node in nodes:
    pq.set_priority(node, INFTY)
    node.set_label(INFTY)
  pq.set_priority(start_node, 0)
  start_node.set_label(0)
  while not pq.is_empty():
    x = pq.extract_min()
    for edge in graph.edge2neighbors(x):
      y = nodes[edge.to_node()]
      if y in pq:
        new_dy = x.label() + edge.label()
        if new_dy < y.label():
          y.set_label(new_dy)
          pq.set_priority(y,new_dy)
          pred[hash(y)] = hash(x)
  return pred
```

grey: current node
grey: current edge
dark grey: final node

pq: 4(5), 2(8), 5(8), 3(∞)

nodes	0	1	2	3	4	5
pred	⊥	0	1	⊥	1	0

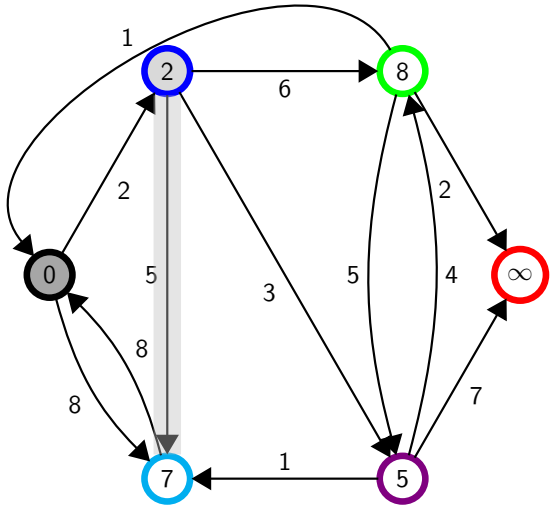



```
def dijkstra(graph, start_node):
  nodes = graph.get_nodes()
  pred = [None] * len(nodes)
  pq = PriorityQueue()
  for node in nodes:
    pq.set_priority(node, INFTY)
    node.set_label(INFTY)
  pq.set_priority(start_node, 0)
  start_node.set_label(0)
  while not pq.is_empty():
    x = pq.extract_min()
    for edge in graph.edge2neighbors(x):
      y = nodes[edge.to_node()]
      if y in pq:
        new_dy = x.label() + edge.label()
        if new_dy < y.label():
          y.set_label(new_dy)
          pq.set_priority(y,new_dy)
          pred[hash(y)] = hash(x)
  return pred
```

grey: current node
grey: current edge
dark grey: final node

pq: 4(5), 5(7), 2(8), 3(∞)

nodes	0	1	2	3	4	5
pred	⊥	0	1	⊥	1	1

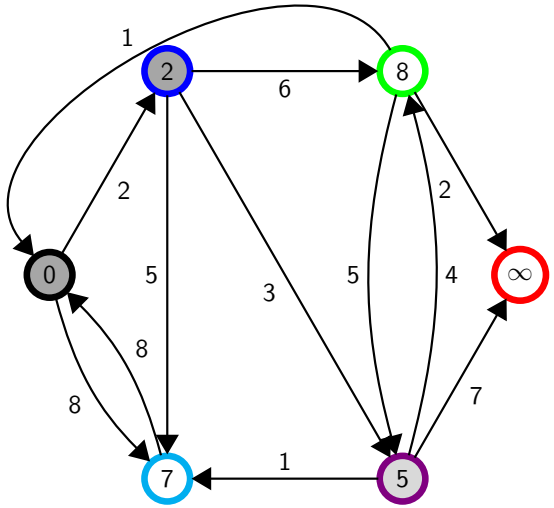



```
def dijkstra(graph, start_node):
  nodes = graph.get_nodes()
  pred = [None] * len(nodes)
  pq = PriorityQueue()
  for node in nodes:
    pq.set_priority(node, INFTY)
    node.set_label(INFTY)
  pq.set_priority(start_node, 0)
  start_node.set_label(0)
  while not pq.is_empty():
    x = pq.extract_min()
    for edge in graph.edge2neighbors(x):
      y = nodes[edge.to_node()]
      if y in pq:
        new_dy = x.label() + edge.label()
        if new_dy < y.label():
          y.set_label(new_dy)
          pq.set_priority(y,new_dy)
          pred[hash(y)] = hash(x)
  return pred
```

grey: current node
grey: current edge
dark grey: final node

pq: 5(7), 2(8), 3(∞)

nodes	0	1	2	3	4	5
pred	⊥	0	1	⊥	1	1

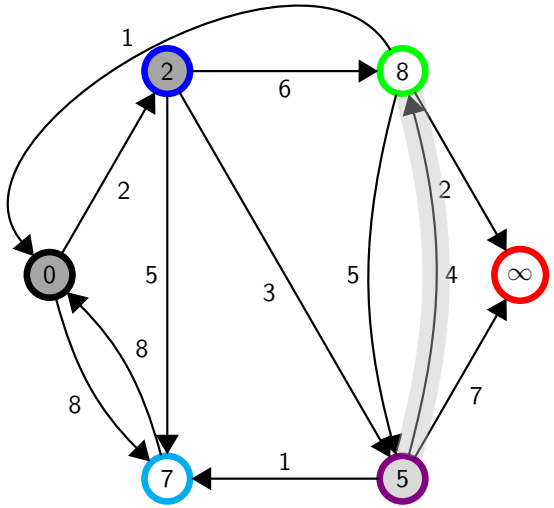



```
def dijkstra(graph, start_node):
  nodes = graph.get_nodes()
  pred = [None] * len(nodes)
  pq = PriorityQueue()
  for node in nodes:
    pq.set_priority(node, INFTY)
    node.set_label(INFTY)
  pq.set_priority(start_node, 0)
  start_node.set_label(0)
  while not pq.is_empty():
    x = pq.extract_min()
    for edge in graph.edge2neighbors(x):
      y = nodes[edge.to_node()]
      if y in pq:
        new_dy = x.label() + edge.label()
        if new_dy < y.label():
          y.set_label(new_dy)
          pq.set_priority(y,new_dy)
          pred[hash(y)] = hash(x)
  return pred
```

grey: current node
grey: current edge
dark grey: final node

pq: 5(7), 2(8), 3(∞)

nodes	0	1	2	3	4	5
pred	⊥	0	1	⊥	1	1

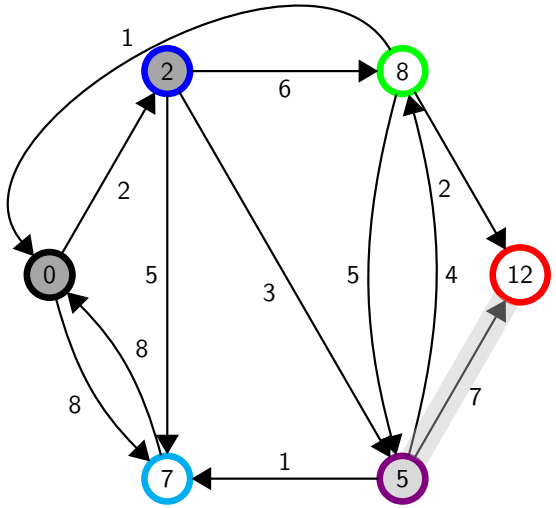



```
def dijkstra(graph, start_node):
  nodes = graph.get_nodes()
  pred = [None] * len(nodes)
  pq = PriorityQueue()
  for node in nodes:
    pq.set_priority(node, INFTY)
    node.set_label(INFTY)
  pq.set_priority(start_node, 0)
  start_node.set_label(0)
  while not pq.is_empty():
    x = pq.extract_min()
    for edge in graph.edge2neighbors(x):
      y = nodes[edge.to_node()]
      if y in pq:
        new_dy = x.label() + edge.label()
        if new_dy < y.label():
          y.set_label(new_dy)
          pq.set_priority(y,new_dy)
          pred[hash(y)] = hash(x)
  return pred
```

grey: current node
grey: current edge
dark grey: final node

pq: 5(7), 2(8), 3(12)

nodes	0	1	2	3	4	5
pred	⊥	0	1	4	1	1

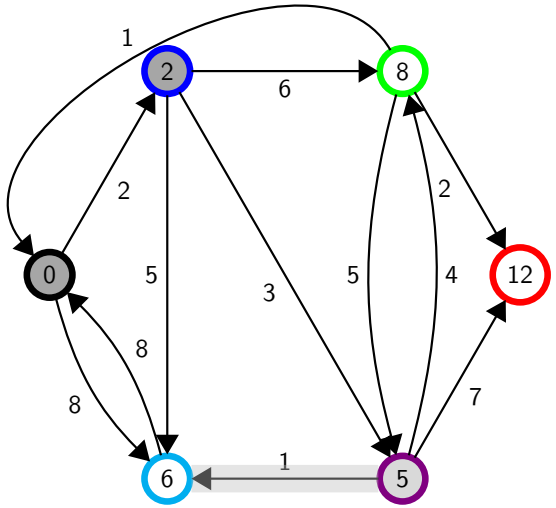



```
def dijkstra(graph, start_node):
  nodes = graph.get_nodes()
  pred = [None] * len(nodes)
  pq = PriorityQueue()
  for node in nodes:
    pq.set_priority(node, INFTY)
    node.set_label(INFTY)
  pq.set_priority(start_node, 0)
  start_node.set_label(0)
  while not pq.is_empty():
    x = pq.extract_min()
    for edge in graph.edge2neighbors(x):
      y = nodes[edge.to_node()]
      if y in pq:
        new_dy = x.label() + edge.label()
        if new_dy < y.label():
          y.set_label(new_dy)
          pq.set_priority(y,new_dy)
          pred[hash(y)] = hash(x)
  return pred
```

grey: current node
grey: current edge
dark grey: final node

pq: 5(6), 2(8), 3(12)

nodes	0	1	2	3	4	5
pred	⊥	0	1	4	1	4

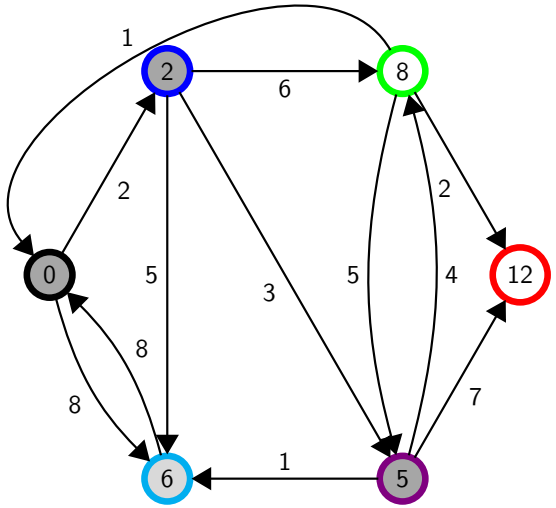



```
def dijkstra(graph, start_node):
  nodes = graph.get_nodes()
  pred = [None] * len(nodes)
  pq = PriorityQueue()
  for node in nodes:
    pq.set_priority(node, INFTY)
    node.set_label(INFTY)
  pq.set_priority(start_node, 0)
  start_node.set_label(0)
  while not pq.is_empty():
    x = pq.extract_min()
    for edge in graph.edge2neighbors(x):
      y = nodes[edge.to_node()]
      if y in pq:
        new_dy = x.label() + edge.label()
        if new_dy < y.label():
          y.set_label(new_dy)
          pq.set_priority(y,new_dy)
          pred[hash(y)] = hash(x)
  return pred
```

grey: current node
grey: current edge
dark grey: final node

pq: 2(8), 3(12)

nodes	0	1	2	3	4	5
pred	⊥	0	1	4	1	4

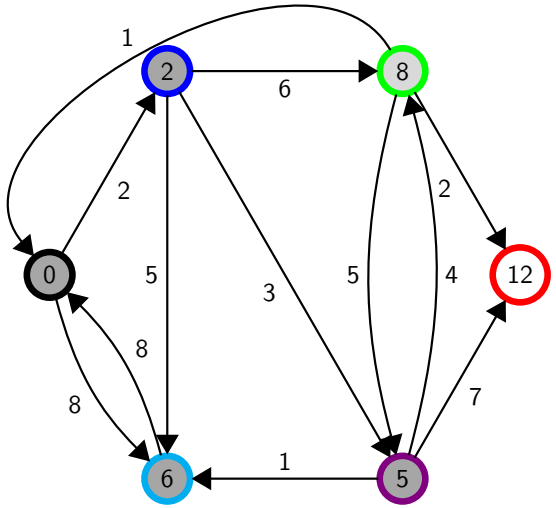



```
def dijkstra(graph, start_node):
  nodes = graph.get_nodes()
  pred = [None] * len(nodes)
  pq = PriorityQueue()
  for node in nodes:
    pq.set_priority(node, INFTY)
    node.set_label(INFTY)
  pq.set_priority(start_node, 0)
  start_node.set_label(0)
  while not pq.is_empty():
    x = pq.extract_min()
    for edge in graph.edge2neighbors(x):
      y = nodes[edge.to_node()]
      if y in pq:
        new_dy = x.label() + edge.label()
        if new_dy < y.label():
          y.set_label(new_dy)
          pq.set_priority(y,new_dy)
          pred[hash(y)] = hash(x)
  return pred
```

grey: current node
grey: current edge
dark grey: final node

pq: 3(12)

nodes	0	1	2	3	4	5
pred	⊥	0	1	4	1	4

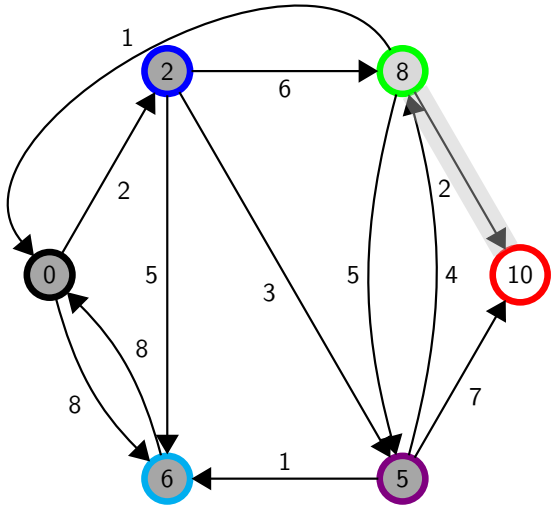



```
def dijkstra(graph, start_node):
  nodes = graph.get_nodes()
  pred = [None] * len(nodes)
  pq = PriorityQueue()
  for node in nodes:
    pq.set_priority(node, INFTY)
    node.set_label(INFTY)
  pq.set_priority(start_node, 0)
  start_node.set_label(0)
  while not pq.is_empty():
    x = pq.extract_min()
    for edge in graph.edge2neighbors(x):
      y = nodes[edge.to_node()]
      if y in pq:
        new_dy = x.label() + edge.label()
        if new_dy < y.label():
          y.set_label(new_dy)
          pq.set_priority(y,new_dy)
          pred[hash(y)] = hash(x)
  return pred
```

grey: current node
grey: current edge
dark grey: final node

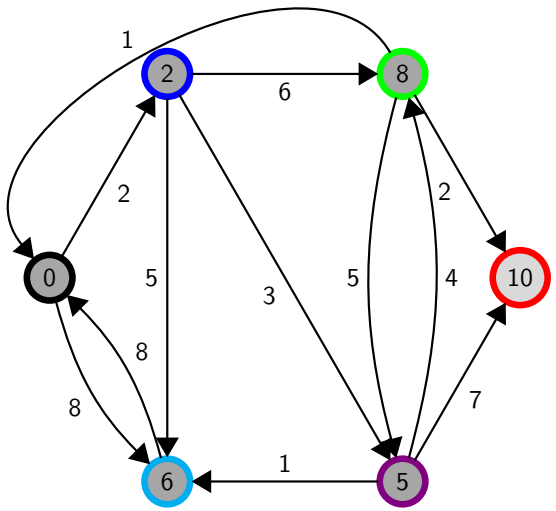
pq: 3(10)

nodes	0	1	2	3	4	5
pred	⊥	0	1	2	1	4




```
def dijkstra(graph, start_node):
  nodes = graph.get_nodes()
  pred = [None] * len(nodes)
  pq = PriorityQueue()
  for node in nodes:
    pq.set_priority(node, INFTY)
    node.set_label(INFTY)
  pq.set_priority(start_node, 0)
  start_node.set_label(0)
  while not pq.is_empty():
    x = pq.extract_min()
    for edge in graph.edge2neighbors(x):
      y = nodes[edge.to_node()]
      if y in pq:
        new_dy = x.label() + edge.label()
        if new_dy < y.label():
          y.set_label(new_dy)
          pq.set_priority(y,new_dy)
          pred[hash(y)] = hash(x)
  return pred
```

grey: current node
grey: current edge
dark grey: final node

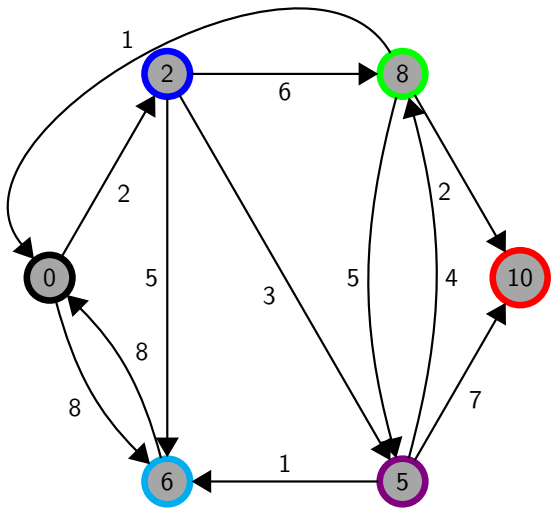


pq:

nodes	0	1	2	3	4	5
pred	$\perp$	0	1	2	1	4

```
def dijkstra(graph, start_node):
  nodes = graph.get_nodes()
  pred = [None] * len(nodes)
  pq = PriorityQueue()
  for node in nodes:
    pq.set_priority(node, INFTY)
    node.set_label(INFTY)
  pq.set_priority(start_node, 0)
  start_node.set_label(0)
  while not pq.is_empty():
    x = pq.extract_min()
    for edge in graph.edge2neighbors(x):
      y = nodes[edge.to_node()]
      if y in pq:
        new_dy = x.label() + edge.label()
        if new_dy < y.label():
          y.set_label(new_dy)
          pq.set_priority(y,new_dy)
          pred[hash(y)] = hash(x)
  return pred
```

grey: current node
grey: current edge
dark grey: final node



pq:

nodes	0	1	2	3	4	5
pred	$\perp$	0	1	2	1	4

```
def dijkstra(graph, start_node):
  nodes = graph.get_nodes()
  pred = [None] * len(nodes)
  pq = PriorityQueue()
  for node in nodes:
    pq.set_priority(node, INFTY)
    node.set_label(INFTY)
  pq.set_priority(start_node, 0)
  start_node.set_label(0)
  while not pq.is_empty():
    x = pq.extract_min()
    for edge in graph.edge2neighbors(x):
      y = nodes[edge.to_node()]
      if y in pq:
        new_dy = x.label() + edge.label()
        if new_dy < y.label():
          y.set_label(new_dy)
          pq.set_priority(y,new_dy)
          pred[hash(y)] = hash(x)
  return pred
```

shortest paths

$0 \to 1$

$0 \to 1 \to 2$

$0 \to 1 \to 2 \to 3$

$0 \to 1 \to 4$

$0 \to 1 \to 4 \to 5$

dynamic frames created by software

developed by Fabian Hausmann

Sorting using Python's build-in methods[7]

- Python3 has basically two methods for sorting:
  - `sorted(l)` returns a copy of the original list in sorted order
  - `l.sort()` sorts list `l` in-place, i.e. the original list is modified

```
unsorted_word_list = ['ag','ga','a','aaa','ca']
sorted_word_list = sorted(unsorted_word_list)
print('unsorted_word_list={}'.format(unsorted_word_list))
print('  sorted_word_list={}'.format(sorted_word_list))
unsorted_int_list = [5,3,1,-4,0,6]
unsorted_int_list.sort()
print('unsorted_list2={}'.format(unsorted_int_list))
```

```
unsorted_word_list=['ag', 'ga', 'a', 'aaa', 'ca']
  sorted_word_list=['a', 'aaa', 'ag', 'ca', 'ga']
unsorted_int_list=[-4, 0, 1, 3, 5, 6]
```

[7] closely follows `https://docs.python.org/3/howto/sorting.html`

Sorting using Python's build-in methods (1/7)

- while `sort()` can only be applied to lists, `sorted` accepts any iterable, such as a dictionary:

```
eop_dist = {'=' : 5, 'I' : 3, 'X' : 4, 'D' : 1}
print('sorted keys of eop_dist={}'.format(sorted(eop_dist)))
```

```
sorted keys of eop_dist=['=', 'D', 'I', 'X']
```

- `list.sort()` and `sorted()` have a parameter `key` to specify a function to be called on each list element prior to making comparisons
- so it is easy to e.g. sort a list of words by their length:

```
length_sorted_word_list = sorted(unsorted_word_list,key=len)
print('length_sorted_word_list = {}'
       .format(length_sorted_word_list))
```

```
length_sorted_word_list = ['a', 'ag', 'ga', 'ca', 'aaa']
```

Sorting using Python's build-in methods (2/7)

- The value of the parameter `key` must be a function that takes a single argument and returns a value to use for sorting.
- the implementation takes care that during the sorting the functions are only evaluated once for each list element
- so the overhead is not too large
- here is an example, in which we use a local function `apply_dict` to sort the keys of a dictionary by their associated values

```
def values_sorted_dict_keys(d):
  def apply_dict(k):
    return d[k]
  return sorted(d,key=apply_dict)

print('values_sorted_eop_dist_keys={}'
        .format(values_sorted_dict_keys(eop_dist)))
```

```
values_sorted_eop_dist_keys=['D', 'I', 'X', '=']
```

Sorting using Python's build-in methods (3/7)

- using a lambda-expression, we can achieve the same result without declaring own functions

```
print('values_sorted_eop_dist_keys={}'
      .format(sorted(eop_dist,key=lambda k: eop_dist[k])))
```

- so a lambda expression introduces a nameless function with arguments listed before the colon `:`
- the nameless function returns the value after the colon
- in our case, `k` is the argument for the nameless function and the returned value is the value in `eop_dist` for key `k`
- lambda functions are very useful to keep code short, but are recommended only for very simple functions

Sorting using Python's build-in methods (4/7)

– here is another example with protein meta data stored in triples:

```
protein_triples = [('Q65209','African swine fever virus',141),
                   ('Q00020','Broad bean mottle virus',1164),
                   ('P03588','Brome mosaic virus',961),
                   ('Q83264','Cucumber mosaic virus',993)]

for protein in sorted(protein_triples,key=lambda p: p[2]):
  print('{}\t{}'.format(protein[1],protein[2]))
```

```
African swine fever virus 141
Brome mosaic virus        961
Cucumber mosaic virus     993
Broad bean mottle virus   1164
```

Sorting using Python's build-in methods (5/7)

– the parameter `key` and lambda-expressions can also be used for named attributes, like in the following example

```
class Protein:
  def __init__(self, accession, name, length):
    self.accession = accession
    self.name = name
    self.length = length
  def __str__(self):
    return '{}\t{}'.format(self.name,self.length)

protein_list = [Protein('Q65209','African swine fever virus',141),
                Protein('Q00020','Broad bean mottle virus',1164),
                Protein('P03588','Brome mosaic virus',961),
                Protein('Q83264','Cucumber mosaic virus',993)]
for protein in sorted(protein_list,key=lambda p: p.name):
  print(protein)
```

```
African swine fever virus 141
Broad bean mottle virus   1164
Brome mosaic virus        961
Cucumber mosaic virus     993
```

Sorting using Python's build-in methods (6/7)

- the use of the key parameter shown above is very common
- so the `operator`-module of Python provides convenience functions `itemgetter()` and `attrgetter()` to make accessor functions easier and faster

```
from operator import itemgetter, attrgetter

for protein in sorted(protein_triples,key=itemgetter(2)):
  print('{}\t{}'.format(protein[1],protein[2]))

for protein in sorted(protein_list,key=attrgetter('length')):
  print(protein)
```

- the `itemgetter`-function is also useful for sorting a dictionary by its values, this time in reverse order

```
for eop, count in sorted(eop_dist.items(),\
                         key=itemgetter(1),\
                         reverse=True):
  print('{}\t{}'.format(eop,count))
```

```
=       5
X       4
I       3
D       1
```

Sorting using Python's build-in methods (7/7)

- sorting by `list.sort()` and `sorted` is stable, i.e. when multiple items have the same key, their original order is preserved

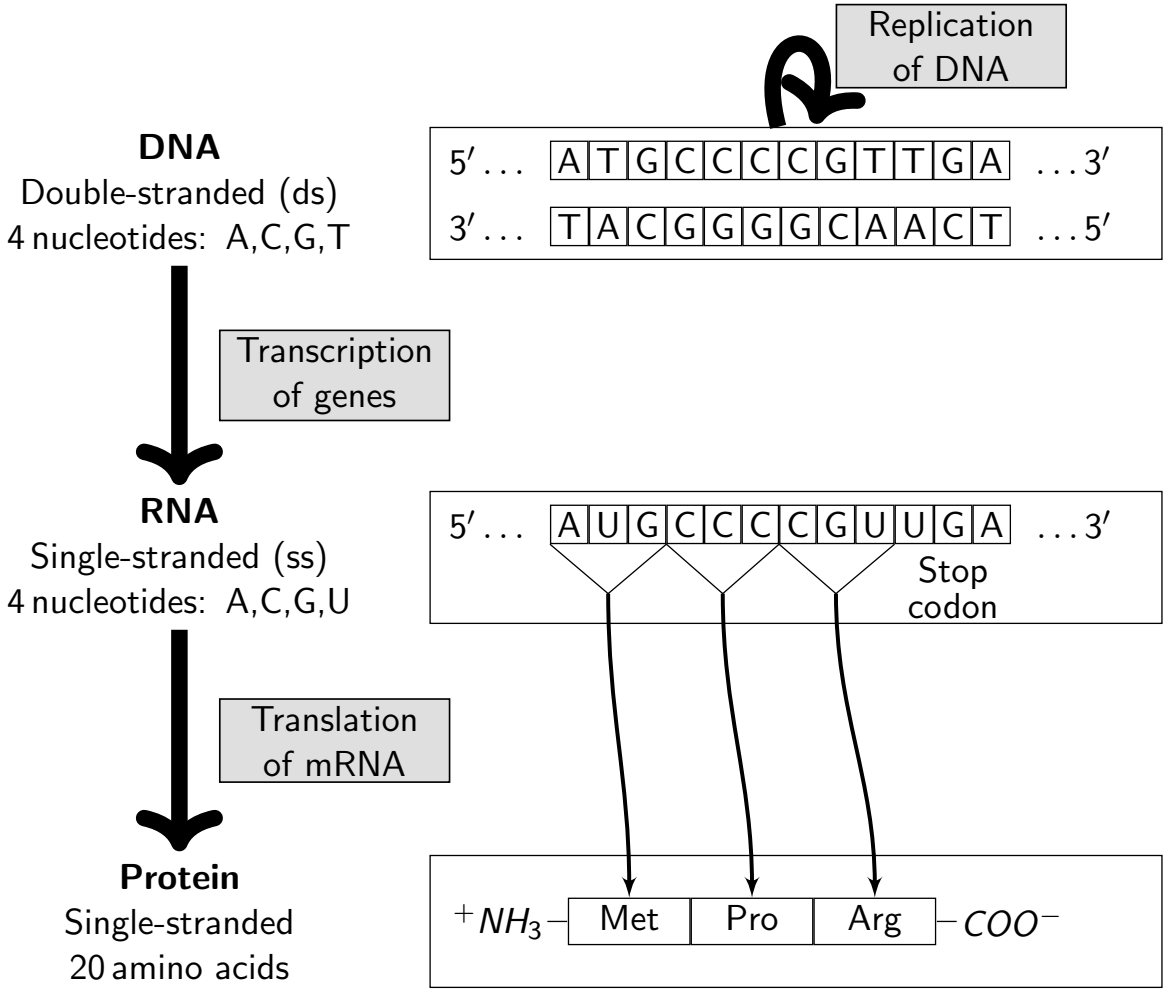
```
colors = [('red', 1), ('blue', 1), ('red', 2), ('blue', 2)]
print(sorted(colors,key=itemgetter(0)))
```

```
[('blue', 1), ('blue', 2), ('red', 1), ('red', 2)]
```

Codon translation (1/6)

central dogma in molecular biology

- DNA is the carrier of genetic information
- proteins are the cellular machines which do the work



Codon translation (2/6)

– each of 64 base triplets (codons) translates to an amino acid according to following table (T is used instead of U)

TTT	Phe	F	TCT	Ser	S	TAT	Tyr	Y	TGT	Cys	C
TTC	Phe	F	TCC	Ser	S	TAC	Tyr	Y	TGC	Cys	C
TTA	Leu	L	TCA	Ser	S	TAA	stop	*	TGA	stop	*
TTG	Leu	L	TCG	Ser	S	TAG	stop	*	TGG	Trp	W
CTT	Leu	L	CCT	Pro	P	CAT	His	H	CGT	Arg	R
CTC	Leu	L	CCC	Pro	P	CAC	His	H	CGC	Arg	R
CTA	Leu	L	CCA	Pro	P	CAA	Gln	Q	CGA	Arg	R
CTG	Leu	L	CCG	Pro	P	CAG	Gln	Q	CGG	Arg	R
ATT	Ile	I	ACT	Thr	T	AAT	Asn	N	AGT	Ser	S
ATC	Ile	I	ACC	Thr	T	AAC	Asn	N	AGC	Ser	S
ATA	Ile	I	ACA	Thr	T	AAA	Lys	K	AGA	Arg	R
ATG	Met	M	ACG	Thr	T	AAG	Lys	K	AGG	Arg	R
GTT	Val	V	GCT	Ala	A	GAT	Asp	D	GGT	Gly	G
GTC	Val	V	GCC	Ala	A	GAC	Asp	D	GGC	Gly	G
GTA	Val	V	GCA	Ala	A	GAA	Glu	E	GGA	Gly	G
GTG	Val	V	GCG	Ala	A	GAG	Glu	E	GGG	Gly	G

– table may vary slightly depending on organism (we use fixed table)

Codon translation (3/6)

- our goal is develop python code that translates a DNA sequence into a protein sequence
- we ignore the transcription and take a group of three consecutive nucleotides (codon) from DNA and translate it into an aminoacid
- iterate this process for all non-overlapping codons

```
def codon2aa(codon):
  codon = codon.upper()
  if   codon == 'TCA': return 'S'      # Serine
  elif codon == 'TCC': return 'S'      # Serine
  elif codon == 'TCG': return 'S'      # Serine
  elif codon == 'TCT': return 'S'      # Serine
  # ... 64 - 7 similar lines
  elif codon == 'GGC': return 'G'      # Glycine
  elif codon == 'GGG': return 'G'      # Glycine
  elif codon == 'GGT': return 'G'      # Glycine
  else:
    sys.stderr.write('Bad codon "{}"\n'.format(codon))
    exit(1)
```

Codon translation (4/6)

- code serves its purpose, but many checks are necessary to perform the translation
- 64 possible codons and 20 aminoacids
- translation is not injective, i.e. different codons may translate into the same aminoacid
- genetic code is redundant $\Rightarrow$ using REs we can enumerate the codons which translate into the same aminoacid:

Codon translation (5/6)

```
def codon2aa(codon):
  codon = codon.upper()
  if   re.search(r'GC.',codon):     return 'A'  # Alanine
  elif re.search(r'TG[TC]',codon): return 'C'  # Cysteine
  elif re.search(r'GA[TC]',codon): return 'D'  # Aspartic Acid
  elif re.search(r'GA[AG]',codon): return 'E'  # Glutamic Acid
  elif re.search(r'TT[TC]',codon): return 'F'  # Phenylalanine
  # ...  20 + 1 - 9 similar lines
  elif re.search(r'GT.',codon):         return 'V'  # Valine
  elif re.search(r'TGG',codon):         return 'W'  # Tryptophan
  elif re.search(r'TA[TC]',codon):      return 'Y'  # Tyrosine
  elif re.search(r'TA[AG]|TGA',codon): return '_'  # Stop
  else:
    sys.stderr.write('Bad codon "{}"\n'.format(codon))
    exit(1)
```

- `/[TC]/` matches a single character, either `T` or `C`
- `/TC.|AG[TC]/` matches `/TC./` or `/AG[TC]/`

Codon translation (6/6)

– a third (and most efficient) variant of `codon2aa` uses a dictionary:

```
genetic_code = {
  'TCA' : 'S',     # Serine
  'TCC' : 'S',     # Serine
  'TCG' : 'S',     # Serine
  'TCT' : 'S',     # Serine
  'TTC' : 'F',     # Phenylalanine
  # ...  55 more similar lines
  'GGA' : 'G',     # Glycine
  'GGC' : 'G',     # Glycine
  'GGG' : 'G',     # Glycine
  'GGT' : 'G',     # Glycine
}
def codon2aa(codon):
  if codon in genetic_code:
    return genetic_code[codon]
  else:
    sys.stderr.write('Bad codon {}\n'
                     .format(codon))
    exit(1)
```

- first part consists of defining a dictionary with 64 entries
- keys are the codons and values are the associated aminoacids
- dictionary is defined as global variable to only initialize it once
- in `codon2aa` first check if key `codon` exists in dictionary
- if this is the case, return the corresponding value
- otherwise, report error

Translating DNA into proteins (1/1)

```
def dna2peptide(seq):
  peptide_list = list()
  for codon in re.findall(r'[acgt]{3}',seq,flags=re.I):
    peptide_list.append(codon2aa(codon))
  return ''.join(peptide_list)

dna = 'CGACGTCTTCGTACGGGACTAGCTCGTGTCGGTCGC'
peptide = dna2peptide(dna)
print('translated DNA {}\ninto protein {}'.format(dna, peptide))
```

```
translated DNA CGACGTCTTCGTACGGGACTAGCTCGTGTCGGTCGC
into protein RRLRTGLARVGR
```

- `findall` marches over the sequence stored in `seq`, matching the non-overlapping substrings of three consecutive bases
- `flags=re.I` means to match case insensitive
- matches are the sought codons
- process each codon by applying `codon2aa` to it
- the resulting amino acid is appended to the string `peptide`

Storing sequences from a multiple FASTA file (1/11)

- we now want to apply the `dna2peptide`-function to sequence files in the FASTA format, most widely used in Bioinformatics
- FASTA format is basically consisting of lines of sequence data
- length of line is not specified, but when generating FASTA format, it is best to limit the line length to some constant ≤ 80 (as some programs cannot handle longer lines)
- each sequence in FASTA-formatted file has header line
- this is a line beginning with the character > followed by some text
- here is an example:

```
>gi|2763999|gb|R30040.1|R30040 12645 Lambda-PRL2 A. thaliana cDNA
TGCAAACCATAACCAAACCATCATCATCATCATCTNTTGGTGACAGAAGAAAAGAGTTGAGGAAC
AGAGAAAATGGCAGCACAAGCACTTGTNTCTTCTTCACTTACCTCCTCTNTTCAGACAGCTAGAC
...
>gi|4714003|dbj|C99879.1|C99879 C99879 A. thaliana mRNA
CACAAATGGAGTCTAGGTTTCACATTACTTGCTTTCCTCTTCATCACTTCCTCTTCCGCTGAGCT
CATCATTAAACAGGTCACACAGGGCAGAGGAATAGAGTACAACAACTCTTACAGTCTCACGTCGA
...
```

Storing sequences from a multiple FASTA file (2/11)

- we develop a class for storing the information in a multiple FASTA file
- as each sequence consists of a header and a sequence we introduce the following class, which corresponds to a named tuple:

```
class SeqEntry:
  def __init__(self,header,sequence):
    self.header = header
    self.sequence = sequence
```

- in addition, the class has a method `show` to print the sequence of a sequence entry in lines of a given maximal width `line_width` which is 70 by default

Storing sequences from a multiple FASTA file (3/11)

```
def show(self,line_width = 70):
  line_list = list()
  print('>{}'.format(self.header))
  for startpos in range(0,len(self.sequence),line_width):
    print(self.sequence[startpos : startpos + line_width])
```

- first comes the header followed by the symbol >
- using the `range`-iterator, one enumerates start positions of the substrings in `self.sequence` to extract and print
- parameters of `range`:

  e.g. `list(range(0,300,70))` ⇒ `[0, 70, 140, 210, 280]`
- slice operator `:` allows to extract substrings in given index range: for a string `s`, `s[i:j]` returns the substring from position `i` to position `j-1`
- last line may be shorter than `line_width`, but this is covered by slice operator: it only extracts characters until the end of given string

Storing sequences from a multiple FASTA file (4/11)

- for the next class we need three helper functions
- the first one wraps the code for opening a file and returns the corresponding stream
- if the filename is the dash symbol, it is interpreted that one wants to read from `sys.stdin`

```
def filename2stream(filename):
  if filename != '-':
    try:
      stream = open(filename,'r')
    except IOError as err:
      sys.stderr.write('{}: {}\n'.format(sys.argv[0],err))
      exit(1)
    return stream
  return sys.stdin
```

Storing sequences from a multiple FASTA file (5/11)

- the second method joins a list of sequences into one sequence and eliminates the white spaces from this

```
def seq_list2seq(seq_list):
  sequence = ''.join(seq_list)
  return re.sub('\s','',sequence)
```

- the third method `fasta2seq_entries` parses the FASTA file and delivers a list of sequence entries
- it initializes three variables:
  - `self.seq_entries` will hold the list of sequence entries
  - `seq_list` will store the list of lines for the current sequence
  - a variable for storing the current header string
- it reads line by line from the given stream
- using `re.search`, check if the line does not begin with the symbol > and is not consisting of white spaces only
- then it is a sequence line which is appended to `seq_list`

Storing sequences from a multiple FASTA file (6/11)

- if `re.search` is successful, then one checks if the line is not the empty string and begins with the symbol >
- if yes, then a header line is detected
- if `header` is `None` it is the first header line, and so `seq_list` is empty
- if `header` is not `None`, the there must be a previous sequence with lines stored in `seq_list`
- so we extract the sequence from `seq_list` and store it with `header` in a new sequence entry appended to `seq_entries`
- for collecting the lines for the next sequence, we have to clear `seq_list`
- in any case we store the current line (except for the symbol > and the last symbol `\n`) in `header`
- after having read all lines, store the last sequence entry whose lines are in `seq_list`

Storing sequences from a multiple FASTA file (7/11)

```
def fasta2seq_entries(stream):
  seq_entries = list()
  header, seq_list = None, list()
  for line in stream:
    if not re.search(r'^(>|\s*$)',line):
      seq_list.append(line.rstrip())
    elif len(line) > 0 and line[0] == '>':
      if header:
        seq_entry = SeqEntry(header,seq_list2seq(seq_list))
        seq_entries.append(seq_entry)
        seq_list.clear()
      header = line[1:-1]
  if header:
    seq_entry = SeqEntry(header,seq_list2seq(seq_list))
    seq_entries.append(seq_entry)
  return seq_entries
```

Storing sequences from a multiple FASTA file (8/11)

- the initial part of the definition of class `Multiseq` simply calls methods to deliver a list of sequence entries
- in case the name of the input file ends with suffix `.gb`, the corresponding function for parsing the Genbank format is called
- this format will be discussed later and the implementation of the function `genbank2seq_entries` will be an exercise
- in case the name of the input file does not end with suffix `.gb`, the corresponding parser for the Fasta-format is called

```
class Multiseq:
  def __init__(self,filename):
    stream = filename2stream(filename)
    if re.search(r'\.gb$',filename):
      self.seq_entries = genbank2seq_entries(stream)
    else:
      self.seq_entries = fasta2seq_entries(stream)
    if filename != '-':
      stream.close()
```

Storing sequences from a multiple FASTA file (9/11)

- `class Multiseq` has four more methods:
  - the method `__getitem__` is declared to overload the index access operator `[]`, i.e. we allow to extract a sequence entry by its index in the list `self.seq_entries`
  - so if we have a statement like `multiseq = Multiseq(filename)` we can access the *i*th sequence entry by `multiseq[i]`
  - the method `__iter__` delivers an iterator to the list, so that we can iterate over the sequence entries with a `for`-loop (without actually needing to know that the member-variable `self.seq_entries` exists)
  - the method `__len__` overloads the method `len` and delivers the number of sequence entries in the `Multiseq`-instance
  - the method `show` displays all sequence entries on `sys.stdout`

Storing sequences from a multiple FASTA file (10/11)

```
def __getitem__(self, idx):
  assert idx >= 0 and idx < len(self.seq_entries)
  return self.seq_entries[idx]

def __iter__(self):
  return iter(self.seq_entries)

def __len__(self):
  return len(self.seq_entries)

def show(self,line_width):
  for seq_entry in self:
    seq_entry.show(line_width)
```

Storing sequences from a multiple FASTA file (11/11)

- finally lets put everything together into a file `fnarw.py` to read a FASTA file, translate it to a peptide and output this:

```
from multiseq import Multiseq
from print_sequence import print_sequence
from dna2aa import dna2peptide

multiseq = Multiseq('sample.fna')
dna = multiseq[0].sequence
peptide = dna2peptide(dna)
print_sequence(peptide,60)
```

- `print_sequence` is a function consisting of the last two lines of `SeqEntry.show()`
- we reuse the class `Multiseq` several times, like in a long case study about restriction enzymes

```
$ head -n 2 sample.fna
>gi|16127994|ref|NC_000913.1| Escherichia coli K12 genome
agatggcggcgctgaggggtcttgggggctctaggccggccacctactgg
$ fnarw.py | head -n 1
RWRR_GVLGALGRPPTGLQRRRRMGPAQ_EYAAWEA_LEAEVVVGAFATAWDAAEWSVQV
```

Mapping restriction enzymes (1/10)

- restriction enzymes are proteins that cut DNA at short, specific sequences
- examples:
  - EcoIRI cuts between `G` and `A` where it finds `GAATTC`
  - HindIII cuts between the `A`s where it finds `AAGCTT`
- there are $\approx$ 1000 known restriction enzymes
- restriction map: all positions where a given restriction enzyme cuts
- very important for planning wet-lab experiments
- goal of this section: write a Python script that looks for restriction enzymes in a sequence
- the restriction enzymes database `http://rebase.neb.com/rebase/rebase.html` is a widely use source for information about restriction enzymes
- several years ago we obtained a fairly complete list of enzymes, which does not seem to be available anymore

Mapping restriction enzymes (2/10)

– here are the first 20 lines of a file describing restriction enzymes:

```
REBASE version 301                                                  bionet.301

    ==============================================================
    REBASE, The Restriction Enzyme Database   http://rebase.neb.com
    Copyright (c)  Dr. Richard J. Roberts, 2002.   All rights reserved.
    ==============================================================

Rich Roberts                                                        Dec 27 2002

AaaI (XmaIII)                       C^GGCCG
AacI (BamHI)                        GGATCC
AaeI (BamHI)                        GGATCC
AagI (ClaI)                         AT^CGAT
AaqI (ApaLI)                        GTGCAC
AarI                                CACCTGCNNNN^
AarI                                ^NNNNNNNNGCAGGTG
AasI (DrdI)                         GACNNNN^NNGTC
AatI (StuI)                         AGG^CCT
AatII                               GACGT^C
```

Mapping restriction enzymes (3/10)

- the first ten lines up until the line beginning with `Ritch Robers` serves as a comment and can be discarded
- each of the remaining lines consists of two or three columns
- first item in each line is name of restriction enzyme
- names in parentheses are synonyms (can be ignored for our purpose)
- last column specifies recognition site, which is to be searched
- symbol ^ denotes cut point
- the recognition site is given as a sequence of bases and additional symbols `N`, `S`, `Y`, `W`, `R`, `K`, `V`, `B`, `D`, `H`, `M`, the IUB ambiguity characters
- each such character matches a subset of $\{$`A`, `C`, `G`, `T`$\}$

Mapping restriction enzymes (4/10)

`R` means `G` or `A`
`Y` means `C` or `T`
`M` means `A` or `C`
`K` means `G` or `T`
`S` means `G` or `C`
`W` means `A` or `T`
`B` means not `A` (i.e. `C` or `G` or `T`)
`D` means not `C` (i.e. `A` or `G` or `T`)
`H` means not `G` (i.e. `A` or `C` or `T`)
`V` means not `T` (i.e. `A` or `C` or `G`)
`N` means `A` or `C` or `G` or `T`

- so the recognition site `GTMKAC` matches `GT` followed by `A` or `C` followed by `G` or `T` followed by `AC`
- this can be expressed by the RE `GT[AC][GT]AC`
- so if we translate the recognition site patterns into REs expressions we can search for them using `re.search` or similar methods

Mapping restriction enzymes (5/10)

– to transform the recognition site into a valid RE, we use the following function, based on a dictionary mapping each IUB-character to a string representing a RE

```
def iub_to_regexp(iub):
  iub2character_class = {
        'A' : 'A',
        'C' : 'C',
        'G' : 'G',
        'T' : 'T',
        'R' : '[GA]',
        'Y' : '[CT]',
        'M' : '[AC]',
        'K' : '[GT]',
        'S' : '[GC]',
        'W' : '[AT]',
        'B' : '[CGT]',
        'D' : '[AGT]',
        'H' : '[ACT]',
        'V' : '[ACG]',
        'N' : '[ACGT]'
    }
```

Mapping restriction enzymes (6/10)

```
# list of regexps mapped to IUB characters
mapped = list()
# Replace each IUB item with its character class
for iubchar in iub:
  if iubchar in iub2character_class:
    mapped.append(iub2character_class[iubchar])
  else:
    sys.stderr.write('{}: unknown IUB-character {}\n'
                     .format(sys.argv[0],iubchar))
    exit(1)
return ''.join(mapped)
```

Mapping restriction enzymes (7/10)

– the next function parses the REBASE file:

```
def parseREBASE(rebasefile):
  rebase_dict = dict()   # dictionary to be returned
  stream = myopen(rebasefile)

  for line in stream:
    if not re.search(r'^(\s+|REBASE|Rich Roberts)',line):
      fields = line.split()     # split the 2 or 3 fields
      # Remove parenthesized names by not saving the middle
      # field (if any), just the first and last
      re_name = fields.pop(0)        # extract first element
      re_site = fields.pop()         # extract last element
      # Remove ^ signs from the recognition sites
      re_site = re.sub(r'\^','',re_site)
      regex = iub_to_regexp(re_site)  # translate recog. site
      rebase_dict[re_name] = (re_site,regex)

  print('parsed {} restriction enzymes'.format(len(rebase_dict)))
  return rebase_dict # Return dictionary with reformatted REBASE
```

Mapping restriction enzymes (8/10)

– now match the REs and obtain their start positions

```
def match_positions_fwd(regexp, sequence):
  poslist = list()
  # match regexp against sequence, be case insensitive
  for m in re.finditer(regexp, sequence, flags=re.I):
    poslist.append(m.start())
  return poslist
```

re.finditer(pattern,string,flags=0) ⇒ match iterator

return an iterator yielding match objects over all non-overlapping matches for RE pattern in string. String is scanned left-to-right, and matches are returned in the order found. `flags` allows to modify semantics of pattern.

– `flags=re.I` means to match case insensitive, i.e. each alphabetic letter matches the corresponding letter in lower or in upper case

– `m.start()` for the match object `m` delivers the start position of the current match in `sequence`

Mapping restriction enzymes (9/10)

- now put all the codes together, assuming that
  - `inputfile` is the name of a Fasta-formatted file with the DNA sequence and the list of restriction
  - `queries` is a list of enzyme names to be mapped

```
multiseq = Multiseq(inputfile)
dna = multiseq[0].sequence
rebase_dict = parseREBASE('REBASE.txt') # Get REBASE into dict
for query in queries:  # iterate over queries = sys.args[2:]
  if query in rebase_dict:
    site, regexp = rebase_dict[query] # elem at index 0 and 1
    poslist = match_positions_fwd(regexp, dna)
    if not poslist: # list is empty
      print('{}={} does not occur in DNA'.format(query,site))
    else:
      print('{}={} occurs at pos {}'
            .format(query,site,', '.join(map(str,poslist))))
  else:
    sys.stderr.write('{}: {} is not a valid name\n'
                     .format(sys.argv[0],query))
    exit(1)
```

Mapping restriction enzymes (10/10)

```
$ restriction.py Ath.fna AccI Afl83II
parsed 3338 restriction enzymes
AccI=GTMKAC occurs at pos 876
Afl83II=GGCC occurs at pos 323, 455
```

Mapping restrict. enz.: a class implementation (1/5)

- now put the functions related to restriction enzymes into a class
- recall: a class specifies data and methods to manipulate the data
- data:
  - map to transform iub-characters to character class (same for all objects ⇒ can be a class variable)
  - rebase-dictionary (can be different in different instances of the class, e.g. from different versions of REBASE ⇒ instance variable)
  - DNA-sequence to search in (is very likely to be different in different instances of the class ⇒ instance variable)
- public methods (which can be accessed by users of the class)
  - `__init__`-methods for class (must always be public method):
    - directly called after creates an instance of the class
    - initializes rebase-dictionary and DNA-sequence
  - method for matching a given RE

Mapping restrict. enz.: a class implementation (2/5)

```
class RestrictFind: # class definition: keyword class and name
  iub2character = { # class var: exists only once
                          'A' : 'A',
                          'C' : 'C',
                          'G' : 'G',
                          'T' : 'T',
                          'R' : '[GA]',
                          'Y' : '[CT]',
                          'M' : '[AC]',
                          'K' : '[GT]',
                          'S' : '[GC]',
                          'W' : '[AT]',
                          'B' : '[CGT]',
                          'D' : '[AGT]',
                          'H' : '[ACT]',
                          'V' : '[ACG]',
                          'N' : '[ACGT]'
                      }
  def __init__(self, seqfile, rebase_file = 'REBASE.txt'):
    self.rebase_dict = self._parseREBASE(rebase_file) # inst var
    multiseq = Multiseq(seqfile)
    self.dna = multiseq[0].sequence
```

Mapping restrict. enz.: a class implementation (3/5)

```
def get_restriction_matches(self, query):
  if query in self.rebase_dict:
    recognition_site, regexp = self.rebase_dict[query]
    poslist = self._match_positions_fwd(regexp)
  else:
    raise Exception('"{}" is not a valid name'.format(query))
  return (recognition_site, poslist)
# all methods above are public, private methods begin with _
def _match_positions_fwd(self, regexp):
  poslist = list()
  for m in re.finditer(regexp,self.dna, flags=re.I):
    poslist.append(m.start())
  return poslist

def _iub_to_regexp(self, iub):
  mapped = list()
  for iubchar in iub: # Replace IUB item with its character class
    if iubchar in self.iub2character:
      mapped.append(self.iub2character[iubchar])
    else:
      raise Exception('unknown IUB-character {}'.format(iubchar))
```

Mapping restrict. enz.: a class implementation (4/5)

```
    return ''.join(mapped)

  def _parseREBASE(self, rebasefile):
    rebase_dict = dict()   # dict to be returned
    stream = open(rebasefile,'r')
    for line in stream:
      if not re.findall('^(\s+|REBASE|Rich Roberts)',line):
        fields = line.split()     # split the 2 or 3 fields
        # Remove parenthesized names by not saving the middle
        # field (if any), just the first and last
        re_name = fields.pop(0)        # extract first element
        re_site = fields.pop()          # extract last element
        # Remove ^ signs from the recognition sites
        re_site = re.sub(r'\^','',re_site)
        regex = self._iub_to_regexp(re_site)  # translate recog.
            site
        rebase_dict[re_name] = (re_site,regex)
    return rebase_dict  # Return dictionary with REBASE content
```

Mapping restrict. enz.: a class implementation (5/5)

- as in previous example, assume that `inputfile` and `queries` are given
- there are two situations, where an exception is raised:
  - in the private method `_iub_to_regexp` called by `__init__`
  - in public method `_get_striction_matches` called in iteration over queries
- we handle both exception by enclosing the instantiation of a `RestrictionFind`-object in a `try`/`except`
- the output is the same as what is shown on frame 360

```
try:
  rf = RestrictFind(inputfile)  # create instance rf of class
  for query in queries:  # iterate over queries = sys.args[2:]
    site, poslist = rf.get_restriction_matches(query)
    if not poslist:
      print('{}={} does not occur in DNA'.format(query,site))
    else:
      print('{}={} occurs at pos {}'
              .format(query,site,', '.join(map(str,poslist))))
except Exception as err:
  sys.stderr.write('{}: {}\n'.format(sys.argv[0],err))
  exit(1)
```

Generators (1/9)

- we have seen many examples of iterating over collections of items, like characters in a string, elements in a list, or lines in an input stream
- now we consider how to implement own generators (a special form of iterators)
- generator is method that yields values rather than returning them

```
def arith(x):  # generates two values
  yield x + 1  # first this
  yield x * 2  # then this

for y in arith(5):   # iterate over the generated values
  print(y)
```

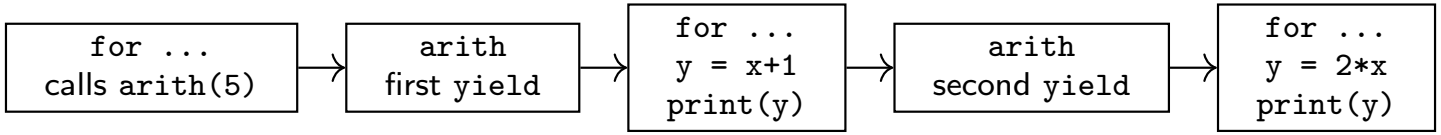
```
6
10
```

- the expressions associated with `yield` are delivered in an iteration in the order they appear in the generator

Generators (2/9)

- after the `yield`, the control flow continues with the block in the iteration (`print` in our context)
- once this block has been executed the control flow continues with the statement directly after the previous `yield`
- so the control flow switches between the iterating `for`-loop with the `print`-statement and the generator `arith`

```
def arith(x):   # generates two values
  yield x + 1   # first this
  yield x * 2   # then this

for y in arith(5):    # iterate over the generated values
  print(y)
```



Generators (3/9)

- example: sequence of fibonacci numbers $f_1, f_2, f_3, \ldots$ is with $f_1 = f_2 = 1$ and $f_i = f_{i-1} + f_{i-2}$ for $i \geq 3$:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, \ldots$$

- models the growth of a population under ideal conditions

```
def fib_up_to(maxfib):
  p = pp = 1  # p is previous, pp is previous of previous
  while pp <= maxfib:
    yield pp   # iteration will receive pp as value
    current_sum = p + pp
    pp = p
    p = current_sum

for f in fib_up_to(1000): # use function as iterator
  print('{} '.format(f),end='')  # f will be yielded value
print()
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Generators (4/9)

- one can even leave the maximum value undefined
- instead decide in the executed block about the largest fibonacci number to be created and displayed

```
def fib_infinite():
  p = pp = 1  # p is previous, pp is previous of previous
  while True:
    yield pp   # iteration will receive pp as value
    current_sum = p + pp
    pp = p
    p = current_sum

for f in fib_infinite(): # use function as iterator without limit
  if f > 1000:           # f is yielded value
    break
  print('{} '.format(f),end='')
print()
```

- generators are e.g. used when reading large files with information grouped in blocks, like in the following simplified `mol2`-formatted file:

Generators (5/9)

```
MOLECULE PQQ
      1 N1        -16.1920   17.8820   13.5410 N.pl3  1 PQQ1  0.0000
      2 C2        -15.7570   17.0390   12.5390 C.2    1 PQQ1  0.0000
      3 C2X       -16.6460   16.1640   11.7100 C.2    1 PQQ1  0.0000
MOLECULE CLM
      1 C1          1.1530   17.1150   14.2980 C.3    1 CLM1  0.0000
      2 CL1A        2.0330   16.7450   12.8850 Cl     1 CLM1  0.0000
```

- so each molecule starts with a line beginning with the keyword `MOLECULE` following by its name
- this is followed by lines describing the atoms belonging to the molecule
- this is a simplification of the widely used mol2-format, which has a more complicated header (of several lines), an own header for the atom list and a block with bindings
- see future exercise sheet with material containing a complete mol2-generator

Generators (6/9)

- now let us develop a generator for parsing this format such that each iteration yields the current molecule-entry with its name and the atom list
- for each line we have to consider two cases:
  - if the line is the header of a new molecule there are two subcases:
    - it is the header of the first molecule in which case we only have to save its name
    - it is not the header of the first molecule in which case we have to report the name of the previous molecule and its atom list in a yield statement
  - if the line begins with a space followed by some non-space character, this must be an atom line and we add this to the current atom list
- after we have processed the last line, we have to output the last molecule in a yield statement
- this idea can easily be turned into a generator `molminiIterator`
- this takes a filename as parameter, opens the file and iterates over the lines as described above

Generators (7/9)

- we use two variables for
  - the name of the current molecule (initially `None`) and
  - the corresponding atom list (initially empty)

```
def molminiIterator(stream):
  molecule_name = None
  atom_list = list()
  for line in stream:
    line = line.rstrip()
    m = re.search(r'^MOLECULE\s(.*)$',line)
    if m:
      if atom_list: # output previous molecule
        yield molecule_name, atom_list # will be processed
        atom_list = list()      # reset
      molecule_name = m.group(1)
    elif re.search(r'^\s+\S',line):  # atom line
      atom_list.append(line)
  if atom_list:                 # process last molecule
    yield molecule_name, atom_list
```

Generators (8/9)

- the main part of the program opens the input file,
- calls the generator for the opened stream,
- counts the number of molecules and atoms, and
- output the total number of molecules and the average number of atoms

```
if len(sys.argv) != 2:
  sys.stderr.write('Usage: {} <filename>\n'.format(sys.argv[0]))
  exit(1)
try:
  stream = open(sys.argv[1])
except IOError as err:
  sys.stderr.write('{}: {}\n'.format(sys.argv[0],err))
  exit(1)
```

Generators (9/9)

```
count_molecules = count_atoms = 0
for molecule_name, atom_list in molminiIterator(stream):
  count_molecules += 1
  count_atoms += len(atom_list)
stream.close()
print('found {} molecules with {:.2f} atoms on average'
      .format(count_molecules,count_atoms/count_molecules))
```

Sets and Multisets (1/9)

- a set is a collection of items, in which each elements occurs only once
- there are many cases, were we want elements to occur more than once in a collection
- in such a case, a multiset may be the more appropriate notion
- formally a multiset is a pair $(\mathcal{A}, M)$, where $\mathcal{A}$ is a set and $M : \mathcal{A} \to \mathbb{N}_0$ is a function
- for each $x \in \mathcal{A}$, $M(x)$ is the number of occurrences of x in the multiset (also called multiplicity of x)
- $\sum_{x \in \mathcal{A}} M(x)$ is the size of the multiset $(\mathcal{A}, M)$, denoted by $|(\mathcal{A}, M)|$
- suppose that $\mathcal{A}$ is finite and ordered, i.e.
  - $\mathcal{A} = \{x_0, \ldots, x_{q-1}\}$ for some $q \geq 0$
  - $x_0 < \cdots < x_{q-1}$ for some order $<$.

  Then $(\mathcal{A}, M)$ can uniquely be represented by a list of length $|(\mathcal{A}, M)|$ in which each x_i appears $M(x_i)$ times and the elements in the list are ordered by $<$.

Sets and Multisets (2/9)

Example: Let $\mathcal{A} = \{a, b, c, d\}$ and $M(a) = 0$, $M(b) = 1$, $M(c) = 2$, $M(d) = 1$. Then the size of $(\mathcal{A}, M)$ is

$$M(a) + M(b) + M(c) + M(d) = 0 + 1 + 2 + 1 = 4$$

Assuming the order $a < b < c < d$, $(\mathcal{A}, M)$ can represented by the list $[b, c, c, d]$ of length 4.

Example:

- let $\mathcal{A}$ be the ordered alphabet of 20 amino acids
- suppose we can assign a weight $\sigma(a) \in \mathbb{R}_+$ to each amino acid
- this may be a value expressing the molecular weight
- let q be some fixed positive integer

Sets and Multisets (3/9)

- define the weight $\sigma(u)$ of an amino acid sequence u of length q by

$$\sigma(u) = \sum_{i=1}^{q} w[i]$$

- to efficiently compute these weights for all q-grams of many sequences, we need a lookup table that stores $\sigma(u)$ for each q-gram u
- there are 20^q different q-grams
- the weight of a q-gram does not depend on the order of the amino acids
- e.g. under any weight function w and for $q = 4$ the weight of `TMFH` and `FHTM` are the same
- the weight only depends on the distribution of the amino-acids in the q-gram, e.g. `TMFH` and `FHTM` have the same distribution

Sets and Multisets (4/9)

- so, in this context it is more appropriate to represent a q-gram by a multiset and compute a lookup table for each possible multiset of size q
- so the next question to answer is: how many multisets of size q exist?

Sets and Multisets (5/9)

- there are $\binom{r+q-1}{q}$ multisets of size q over a set of size r
- so lets write a Python function to compute this
- it is based on a function `binom_evaluate(n,k)` to evaluate $\binom{n}{k}$

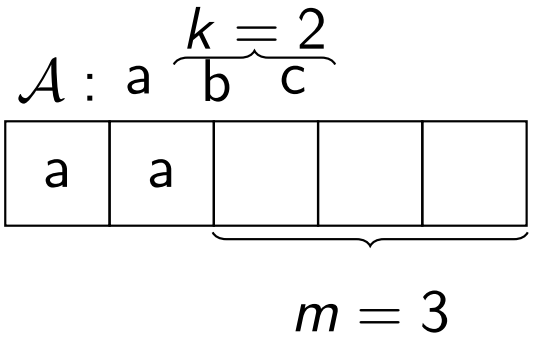
```
def binom_evaluate(n,k):
  if k == 0 or n == k:
    return 1
  if n == 0:
    return 0
  result = 1
  for j in range(1,k+1):
    result = (result * (n - k + j))//j
  return result

def multisets_number(q,r):
  return binom_evaluate(r + q - 1, q)
```

- to understand how multisets can be constructed, lets first write our own function `multisets_count1` which determines the number of multisets recursively, using a function `count_rec1(q,r)`.

Sets and Multisets (6/9)

- as discussed above, a multiset of size q can be considered as a list of length q in which the members of $\mathcal{A}$ appear in sorted order (according to their multiplicity)
- so once we have chosen to place an element a on some position of the multiset, no elements smaller than a can be placed for the following positions
- for this reason, instead of directly working with the elements from $\mathcal{A}$, in `count_rec1` we use a parameter `k`, which is the remaining number of the ordered elements from $\mathcal{A}$ to choose from
- parameter `m` of `count_rec1` is the remaining number of positions in the multiset where to place elements, see the following illustration:

Sets and Multisets (7/9)



- so if `m=0`, then we have already placed all elements of the multiset (without explicitly constructing it) and we count this as 1.
- if `m>0` and `k=0`, we have no elements to choose from for the remaining positions of the multiset, so in this case we have no complete multiset and thus the local counter remains 0
- if `m>0` and `k>0`, we can take between 0 and `m` of the remaining positions and assign to these the next element (so let `take`$\in \{0, \ldots, m\}$)
- this means that `m-take` positions remain
- on these we can place the remaining `k-1` elements from the alphabet

Sets and Multisets (8/9)

- the number of multisets obtained for each choice `take` (in the range from 0 to `m`) can be determined by calling `count_rec1` recursively
- as all described choices are possible, we have to add up the counts for all possible values of the variable `take` in the local variable `count` which is finally returned

```
def multisets_count1(multiset_size,
                     num_elems):
  def count_rec1(m,k):
    if m == 0:
      return 1
    count = 0
    if k > 0:
      for take in range(0,m+1):
        count += count_rec1(m - take,
                            k - 1)
    return count
  return count_rec1(multiset_size,
                    num_elems)
```

Sets and Multisets (9/9)

- a simple test shows that `multisets_count1` computes the correct numbers, shown in the following table for an alphabet of size 20
- besides the number of multisets of size q, we also show the number of sequences of length q, both for an alphabet of size 20

r	q	number of multisets	number of sequences
20	2	210	400
20	3	1 540	8 000
20	4	8 855	160 000
20	5	42 504	3 200 000
20	6	177 100	64 000 000
20	7	657 800	1 280 000 000

- but the recursive function is much slower than the function `multisets_number`, as the number of recursive calls is at least as large as the number determined

Elimination of recursion (1/10)

- the main goal of this section is to exemplify how to transform recursive functions into iterative functions
- we start with the recursive function above
- as a first step towards our goal, we rewrite `count_rec1` into another recursive function `count_rec2`, which does not return a computed value but accumulates it at index 0 of a list `count_list`

```
def multisets_count1(multiset_size,
                     num_elems):
  def count_rec1(m,k):
    if m == 0:
      return 1
    count = 0
    if k > 0:
      for take in range(0,m+1):
        count += count_rec1(m - take,
                            k - 1)
    return count
  return count_rec1(multiset_size,
                    num_elems)
```

- use of list is necessary, as we want a mutable data structure

```
def multisets_count2(multiset_size,
                     num_elems):
  count_list = [0]
  def count_rec2(m,k):
    if m == 0:
      count_list[0] += 1
    elif k > 0:
      for take in range(0,m+1):
        count_rec2(m - take,k - 1)
  count_rec2(multiset_size,num_elems)
  return count_list[0]
```

Elimination of recursion (2/10)

- to eliminate the recursion, we introduce a stack which holds the tasks to be solved
- in our case a task consists of pairs `(m,k)` of integers which have the meaning as the parameters or `count_rec1` or `count_rec2`
- initially, `m` is the size of the multisets to count and `k` is the number of elements in the alphabet
- in the iteration one checks if there are still tasks to be solved and if so, then a task is popped from the stack
- the rest of the function is almost identical to the code in `count_rec2`:. Here are the differences:
  - instead of a list-element, we use an integer-counter for the number of multisets
  - the recursive calls are replaced by `append`-and `pop`-operations on the stack

Elimination of recursion (3/10)

```
def multisets_count2(multiset_size,
                     num_elems):
  count_list = [0]
  def count_rec2(m,k):
    if m == 0:
      count_list[0] += 1
    elif k > 0:
      for take in range(0,m+1):
        count_rec2(m - take,k - 1)
  count_rec2(multiset_size,num_elems)
  return count_list[0]
```

```
def multisets_count3(multiset_size,
                     num_elems):
  stack = [(multiset_size,num_elems)]
  count = 0
  while stack:
    m, k = stack.pop()
    if m == 0:
      count += 1
    elif k > 0:
      for take in range(0,m+1):
        stack.append((m - take,k - 1))
  return count
```

- a simple test shows that all three `multisets_count`-functions compute the correct numbers
- the counting methods were just an exercise to simplify the development of methods enumerating multisets

Elimination of recursion (4/10)

- consider the simple case for computing the multisets of size $q = 2$ and the alphabet $\mathcal{A} = \{0, 1\}$
- as noted before the multisets can be considered as lists in which the elements appear in sorted order
- so in our special case we have the multisets $[0, 0]$, $[0, 1]$ and $[1, 1]$
- to generalize, let $\mathcal{A} = \{0, 1, \ldots k-1\}$ and keep $q = 2$, then we have the following multisets:

$$
\begin{array}{lll}
[0, j] & \text{for} & 0 \leq j \leq k-1 \\
[1, j] & \text{for} & 1 \leq j \leq k-1 \\
[2, j] & \text{for} & 2 \leq j \leq k-1 \\
\vdots & & \vdots \\
[k-1, j] & \text{for} & k-1 \leq j \leq k-1
\end{array}
$$

- these multisets can be enumerated as follows:

```
def multisets_enum_2loops(num_elems):
  for i in range(0,num_elems):
    for j in range(i,num_elems):
      yield [i, j]
```

Elimination of recursion (5/10)

- to enumerate multisets of size 3, we use 3 nested loops
- the ith loop stores the value of its iteration variable at index i of the generated multiset

- each inner loop starts with the current value of the iteration variable of the closest outer loop
- to enumerate multisets of size m, we use m nested loops
- this schema leads to the following function which works for multisets up to size 4

```
def multisets_enum_loops(multiset_size,
                         num_elems):
  assert multiset_size <= 4
  if multiset_size == 2:
    for i in range(0,num_elems):
      for j in range(i,num_elems):
        yield [i, j]
  elif multiset_size == 3:
    for i in range(0,num_elems):
      for j in range(i,num_elems):
        for k in range(j,num_elems):
          yield [i, j, k]
  elif multiset_size == 4:
    for i in range(0,num_elems):
      for j in range(i,num_elems):
        for k in range(j,num_elems):
          for l in range(k,num_elems):
            yield [i, j, k, l]
```

Elimination of recursion (6/10)

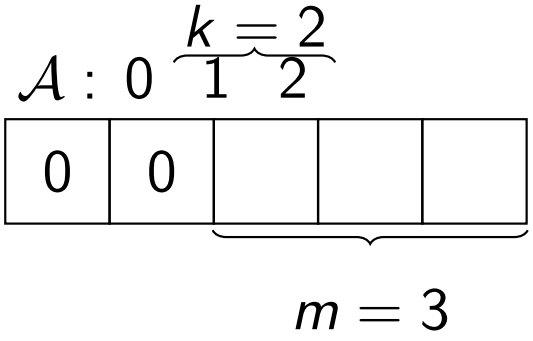
- the above solution is
  - very fast, because no intermediate data structures, like a stack (implicit or explicit), are required
  - but inflexible, as an increase in the multiset size requires an extra loop

⇒ develop a recursive solution and later turn this into an iterative one

```
def multisets_enum_rec(multiset_size,num_elems):
  multiset_list = list()
  multiset = [None] * multiset_size
  def multiset_rec(m,k):
    if m == 0:
      multiset_list.append(multiset.copy())
    elif k > 0:
      for take in range(0,m+1):
        for d in range(multiset_size - m,\
                       multiset_size - m + take):
          multiset[d] = num_elems - k
        multiset_rec(m-take,k-1)
  multiset_rec(multiset_size,num_elems)
  return multiset_list
```

Elimination of recursion (7/10)

- the previous method uses a recursive function which has the same structure as `count_rec2`
- in particular, the case distinction are identical and they appear in the same order
- however, as we want to construct multisets, rather then only counting them, we have to maintain a list of size `multiset_size` which is continuously updated
- this is illustrated as before, but this time using the alphabet symbols 0, 1, and 2



Elimination of recursion (8/10)

- initially, the list entries are all `None`
- whenever we see the boundary case `m=0`, we know that we have filled all positions of the list, and so have a complete multiset
- this is copied to the list which stores all generated multisets
- if `m>0` and `k>0` then, as previously, we fill between 0 and `m` of the remaining positions of the list with the next available symbol
- as `k` is the remaining number of symbols, `num_elems - k` is the symbol number (note that we use symbols from 0 to `num_elems-1`)
- similarly, as `m` is the remaining number of positions to fill, `multiset_size - m` is the first position to fill and as we have to fill `take` positions, `multiset_size -m + take` defines the exclusive upper bound
- in case `take=0`, we have no updates
- once we have performed the updates we fill the remaining `m-take` positions of the list with the remaining `k-1` symbols

Elimination of recursion (9/10)

- as previously, to obtain an iterative version from the recursive one, we introduce a stack to hold the tasks to be solved
- this follows the pattern function `multisets_count3`, i.e. recursion is replaced by applying the `append`- and `pop`-method to the stack
- the rest of the code is identical to `multiset_rec`, except that we implement a generator, rather than appending multisets to a list

```
def multisets_enum_stack(multiset_size,num_elems):
  stack = [(multiset_size,num_elems)]
  multiset = [None] * multiset_size
  while stack:
    m, k = stack.pop()
    if m == 0:
      yield multiset
    elif k > 0:
      for take in range(0,m+1):
        for d in range(multiset_size - m,\
                       multiset_size - m + take):
          multiset[d] = num_elems - k
        stack.append((m-take,k-1))
```

Elimination of recursion (10/10)

- we combine the strengths of the two functions (efficiency of `multisets_enum_loops` and flexibility of `multisets_enum_stack`) into one function

```
def multisets_enum(multiset_size,num_elems):
  if multiset_size <= 4:
    for ms in multisets_enum_loops(multiset_size,num_elems):
      yield ms
  else:
    for ms in multisets_enum_stack(multiset_size,num_elems):
      yield ms
```

- as the previous method generates multisets over the symbol numbers from 0 to `num_elems`, we implement a function transforms the multisets into strings over some sufficiently large alphabet, given as list of characters

```
def multiset_map2alphabet(multiset,alphabet):
  assert len(multiset) <= len(alphabet)
  return list(map(lambda i : alphabet[i],multiset))
```

Genbank (1/1)

- Genbank® (Genetic Sequence DataBank) is the NIH (National Institute of Health) genetic sequence database
- Genbank is an annotated collection of all publicly available DNA sequences
- GenBank is part of the International Nucleotide Sequence Database Collaboration [8]
- this comprises the DNA DataBank of Japan (DDBJ), the European Nucleotide Archive (ENA), and GenBank at NCBI.
- the content of the three databases is equivalent (daily exchange of data), but Genbank comes with its own format
- this chapter describes how to parse and extract information from a Genbank formatted file, using Python
- we will recapitulate previous techniques, but also learn how to use regular expressions to match over several lines of the input

[8] `https://www.ncbi.nlm.nih.gov/genbank/`

The Genbank format (1/3)

- a Genbank entry consists of annotation part and sequence part
- the annotation part itself can be divided into a description part defining (among other thing):
  - accession numbers,
  - definitions of the kind of sequence in the entry,
  - the source where the sequence comes from,
  - references, and
  - a feature table.
- Genbank is a text format designed to be readable by humans, but also automatically by appropriate parsers
- details on the format are specified in `https://www.ncbi.nlm.nih.gov/Sitemap/samplerecord.html`
- the following two pages show an excerpt of a Genbank entry
- we use the suffix `.gb` for files with one or more Genbank entries

The Genbank format (2/3)

```
LOCUS       AB031069     2487 bp    mRNA            PRI       27-MAY-2000
DEFINITION  Homo sapiens PCCX1 mRNA for protein containing CXXC domain 1,
            complete cds.
ACCESSION   AB031069
VERSION     AB031069.1  GI:8100074
KEYWORDS    .
SOURCE      Homo sapiens embryo male lung fibroblast cell_line:HuS-L12 cDNA to
            mRNA.
  ORGANISM  Homo sapiens
            Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
            Mammalia; Eutheria; Primates; Catarrhini; Hominidae; Homo.
REFERENCE   1  (sites)
  AUTHORS   Fujino,T., Hasegawa,M., Shibata,S., Kishimoto,T., Imai,Si. and
            Takano,T.
  TITLE     PCCX1, a novel DNA-binding protein with PHD finger and CXXC domain,
            is regulated by proteolysis
  JOURNAL   Biochem. Biophys. Res. Commun. 271 (2), 305-310 (2000)
  MEDLINE   20261256
REFERENCE   2  (bases 1 to 2487)
  AUTHORS   Fujino,T., Hasegawa,M., Shibata,S., Kishimoto,T., Imai,S. and
            Takano,T.
  TITLE     Direct Submission
  JOURNAL   Submitted (15-AUG-1999) to the DDBJ/EMBL/GenBank databases.
            Tadahiro Fujino, Keio University School of Medicine, Department of
            Microbiology; Shinanomachi 35, Shinjuku-ku, Tokyo 160-8582, Japan
            (E-mail:fujino@microb.med.keio.ac.jp,
            Tel:+81-3-3353-1211(ex.62692), Fax:+81-3-5360-1508)
FEATURES             Location/Qualifiers
     source          1..2487
                     /organism="Homo sapiens"
```

The Genbank format (3/3)

```
                     /db_xref="taxon:9606"
                     /sex="male"
                     /cell_line="HuS-L12"
                     /cell_type="lung fibroblast"
                     /dev_stage="embryo"
     gene            229..2199
                     /gene="PCCX1"
     CDS             229..2199
                     /gene="PCCX1"
                     /note="a nuclear protein carrying a PHD finger and a CXXC
                     domain"
                     /codon_start=1
                     /product="protein containing CXXC domain 1"
                     /protein_id="BAA96307.1"
                     /db_xref="GI:8100075"
                     /translation="MEGDGSDPEPPDAGEDSKSENGENAPIYCICRKPDINCFMIGCD
                     NCNEWFHGDCIRITEKMAKAIREWYCRECREKDPKLEIRYRHKKSRERDGNERDSSEP


                     AMTNRAGLLALMLHQTIQHDPLTTDLRSSADR"
BASE COUNT      564 a    715 c    768 g    440 t
ORIGIN
        1 agatggcggc gctgaggggt cttgggggct ctaggccggc cacctactgg tttgcagcgg
       61 agacgacgca tggggcctgc gcaataggag tacgctgcct gggaggcgtg actagaagcg


     2401 gcctcctctc cctgggtttt gttaataaaa ttttgaagaa accaaaaaaa aaaaaaaaaa
     2461 aaaaaaaaaa aaaaaaaaaa aaaaaaa
//
```

Overview of the different parsing tasks

gb2fasta.py	extract sequence part from a Genbank file
gb2fields.py	get annotation fields from Genbank-file: `LOCUS`, `DEFINITION`, `ACCESSION`, `ORGANISM`
gb2annseq.py	extract annotation/seq. using REs
splitGB.py	split Genbank entry, use REs
searchGB.py	apply some searches to sequence and to annotation of Genbank-file
parseAnno.py	parse some annotations into a dict. to obtain a structured representation
getAnno.py	get the annotation from first Genbank record
parseFeatures.py	extract features from the `FEATURES` field of a Genbank-record
features.py	extract feature entries from Genbank-file

gb2fasta.py: extract sequence from Genbank file (1/3)

```
def gb2fasta(filename):
  seq_lines = None  # initialize list of sequence lines
  stream = myopen(filename,'r')
  for line in stream:
    if re.search(r'^\/\/\n', line): # end-of-record line //\n
      break        # break out of the nearest enclosing loop
    elif seq_lines is not None: # we are in a sequence
      seq_lines.append(line)   # add current line to array
    elif re.search(r'^ORIGIN', line): # line before sequence part
      seq_lines = list()    # now start with the sequences
  stream.close()
  return re.sub(r'[\s0-9]','', ''.join(seq_lines))

for filename in sys.argv[1:]:     # process command line args
  sequence = gb2fasta(filename)
  print('>')                      # empty fasta header
  print_sequence(sequence, 50)  # print formatted, width 50
```

gb2fasta.py: extract sequence from Genbank file (2/3)

- suppose the file `Record.gb` contains a single Genbank entry
- the following lines show the output, when applying `gb2fasta.py` to this file

```
$ ./gb2fasta.py Record.gb
>
agatggcggcgctgaggggtcttgggggctctaggccggccacctactgg
tttgcagcggagacgacgcatggggcctgcgcaataggagtacgctgcct

⋮

gcctcctctccctgggttttgttaataaaattttgaagaaaccaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

- the next task is to extract information from the `LOCUS`-, `DEFINITION`-, `ACCESSION`-, and `ORGANISM`-lines and store this in a dictionary

gb2fasta.py: extract sequence from Genbank file (3/3)

- while the DEFINITION-information may be spread over several continuous lines, the information for LOCUS-, ACCESSION-, and ORGANISM-lines (we are interested in) appears on a single line
- the ORGANISM-line is indented by two blanks while all other are top-level keys with no indentation

```
LOCUS       AB031069     2487 bp    mRNA            PRI       27-MAY-2000
DEFINITION  Homo sapiens PCCX1 mRNA for protein containing CXXC domain 1,
            complete cds.
ACCESSION   AB031069
VERSION     AB031069.1  GI:8100074
KEYWORDS    .
SOURCE      Homo sapiens embryo male lung fibroblast cell_line:HuS-L12 cDNA to
            mRNA.
  ORGANISM  Homo sapiens
            Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
            Mammalia; Eutheria; Primates; Catarrhini; Hominidae; Homo.
```

- the information appears in the order specified above
- in particular, the ACCESSION-line directly follows the DEFINITION-line, a feature we will exploit

gb2fields.py: Get annotation fields from Genbank-file (1/3)

```
indef = False   # used for parsing DEFINITION (can be multiline)
values = dict()  # empty dictionary
stream = myopen(sys.argv[1],'r')
for line in stream:
  if re.search(r'^LOCUS', line):
    line = re.sub(r'^LOCUS\s*','', line) # del. LOCUS at beginning
    values['LOCUS'] = line.rstrip()
  elif re.search(r'^DEFINITION', line):
    line = re.sub(r'^DEFINITION\s*','', line) # delete DEFINITION
    values['DEFINITION'] = line.rstrip()
    indef = True           # now inside DEFINITION
  elif re.search(r'^ACCESSION', line):
    line = re.sub(r'^ACCESSION\s*','', line) # delete ACCESSION
    values['ACCESSION'] = line.rstrip()
    indef = False         # now again outside of DEFINITION
  elif re.search(r'^  ORGANISM', line):
    line = re.sub(r'^\s*ORGANISM\s*', '', line) # delete ORGANISM
    values['ORGANISM'] = line.rstrip()
  elif indef:             # still inside DEFINITION
    line = re.sub(r'^\s+',' ', line) # initial multispaces => space
    values['DEFINITION'] += line.rstrip()
stream.close()
```

gb2fields.py: Get annotation fields from Genbank-file (2/3)

- the previous code iterates over all lines and has an `if`-statement for each of the five corresponding keywords
- in this statement it is checked, whether the current line matches this keyword at the beginning of the line (including blanks for `ORGANISM`)
- if a match occurs the keyword is deleted in the `line`-string and the remaining string is stored in a dictionary using the keyword as key
- for the `DEFINITION`-line we additionally set a local variable `indef` to `True` to correctly handle multiline input
- so whenever none of the five keywords match, we check if we are still in the lines belonging to the `DEFINITION`
- if this is the case, we append the line to the `DEFINITION`-entry in the dictionary
- now we have all relevant information in the dictionary `values` and want to print it out

gb2fields.py: Get annotation fields from Genbank-file (3/3)

- we can enumerate the items in the dictionary `values` by an iteration of the form `for key,value in values.items():`
- but this would give output in undetermined order
- to simplify testing, we prefer the defined order which we get as follows:

```
for key in ['LOCUS','DEFINITION','ACCESSION','ORGANISM']:
  print('*** {} ***\n{}'.format(key,values[key]))
```

```
$ ./gb2fields.py Record.gb
```

```
*** LOCUS ***
AB031069     2487 bp    mRNA            PRI       27-MAY-2000
*** DEFINITION ***
Homo sapiens PCCX1 mRNA for protein containing CXXC domain 1, complete cds.
*** ACCESSION ***
AB031069
*** ORGANISM ***
Homo sapiens
```

gb2annseq.py: extract annotation/seq. using REs (1/7)

- to combine the extraction of the annotation and the sequence, we use regular expressions matching multiple lines

Parsing strategy

- read entire gb-record into string, and process it using REs
- the following function reads the file contents as one long string
- it splits the file contents on the given separator `sep` into an list of units
- as the last element of the split is always the empty string, we discard it after the split

```
def get_file_data(filename, sep = '\n'): # default sep is newline
  stream = myopen(filename)
  units = stream.read().split(sep)
  stream.close()
  assert len(units) > 0 and units[-1] == ''
  units.pop()
  return units
```

gb2annseq.py: extract annotation/seq. using REs (2/7)

- a gb-record begins with the word `LOCUS` and ends with `//` on a separate line.

⇒ read record into a long multiline string, using `//\n` as separator of two consecutive Genbank records

```
record = get_file_data(filename,'//\n')[0] # get first record
annotation = None
dna = None
m = re.search('^(LOCUS.*ORIGIN\s*\n)(.*)',record, flags=re.M|re.S)
if m:
  annotation = m.group(1)
  dna = m.group(2)
else:
  sys.stderr.write('{}: Cannot separate annot./seq. in {}\n'
                   .format(sys.argv[0],filename))
  exit(1)
print('annotation:\n{}DNA:\n{}'.format(annotation,dna),end='')
```

- the REs and flags we used in `re.search` are explained below
- the output of the code presented here is shown on frame 411

gb2annseq.py: extract annotation/seq. using REs (3/7)

- the standard meaning of the symbols ^, $, and . in REs:
  - ^ anchors the RE to the beginning of the string
  - $ anchors the RE to the end of the string
  - . matches any character <u>except</u> newline
- to handle multiline strings appropriately, we use the flags `re.M` and `re.S`
- the operator | between these flags in `flags=re.M|re.S` means that both are set
- the flag `re.S` makes the . match any character <u>including</u> newline.

⇒ this allows to treat the entire string as one single *line* with embedded newlines (multiline).

- the flag `re.M` has the following effect:
  - the character ^ matches at the beginning of the string and at the beginning of each line (immediately following a newline)
  - the pattern character $ matches at the end of the string and at the end of each line (immediately preceding a newline)

gb2annseq.py: extract annotation/seq. using REs (4/7)

- example 1: `re.search(r'^.*$','AAC\nGTT')`
  as . does not match newline (default behavior), this match is unsuccessful
- example 2: `re.search(r'^.*$','AAC\nGTT',flags=re.S)`
  as . matches newline (due to `re.S`) this match is successful and the entire string is matched
- example 3: `re.search(r'^.*$','AAC\nGTT',flags=re.M)`
  as $ matches after the embedded newline, this match is successful from the beginning up to the first embedded newline. That is, `AAC` is the matching substring.
- example 4: `re.search(r'^.*$','AAC\nGTT',flags=re.S|re.M)`
  as . matches newline; this match is successful, i.e. the entire string is matched.

gb2annseq.py: extract annotation/seq. using REs (5/7)

– this can be verified by a simple program:

```
def test_match(flags):
  m = re.search(r'^.*$','AAC\nGTT',flags=flags)
  if m:
    print('match with flags={} in range [{}-{}]'
          .format(flags,m.start(),m.end()))
  else:
    print('no match with flags={}'.format(flags))

test_match(0)
test_match(re.S)
test_match(re.M)
test_match(re.S|re.M)
```

```
no match with flags=0
match with flags=16 in range [0-7]
match with flags=8 in range [0-3]
match with flags=24 in range [0-7]
```

– `re.S` stands for $16 = 2^4$,

– `re.M` stands for $8 = 2^3$ and

– `re.S|re.M` stands for $2^3 + 2^4$

gb2annseq.py: extract annotation/seq. using REs (6/7)

now separate the annotation from the sequence data using the regexp:
`^(LOCUS.*ORIGIN\s*\n)(.*)`

- the first part (enclosed in the first pair of braces) begins with `LOCUS` and ends with `ORIGIN` followed by any number of spaces followed by a newline.
- the second part (enclosed in the second pair of braces) is the remaining text (ending before the line matching `^//$` in the original string)
- the first part of the match is assigned to the variable `annotation` in:

  ```
  annotation = m.group(1)
  ```

- the second part is assigned to the variable `dna` in

  ```
  dna = m.group(2)
  ```

gb2annseq.py: extract annotation/seq. using REs (7/7)

```
$ ./gb2annseq.py Record.gb
```

```
annotation:
LOCUS       AB031069     2487 bp    mRNA            PRI       27-MAY-2000
DEFINITION  Homo sapiens PCCX1 mRNA for protein containing CXXC domain 1,
            complete cds.
ACCESSION   AB031069
```

⋮

```
BASE COUNT      564 a    715 c    768 g    440 t
ORIGIN
DNA:
        1 agatggcggc gctgaggggt cttgggggct ctaggccggc cacctactgg tttgcagcgg
```

⋮

```
     2281 ctgtttctcc ggttctccct gtgcccatcc accggttgac cgcccatctg cctttatcag
     2341 agggactgtc cccgtcgaca tgttcagtgc ctggtggggc tgcggagtcc actcatcctt
     2401 gcctcctctc cctgggtttt gttaataaaa ttttgaagaa accaaaaaaa aaaaaaaaaa
     2461 aaaaaaaaaa aaaaaaaaaa aaaaaaa
```

splitGB.py: split Genbank entry, use REs (1/1)

– to simplify reusing the previous code, we put it into a generator

```
def split_genbank(filename):
  records = get_file_data(filename,'//\n')
  for record in records:
    mo = re.search(r'^(LOCUS.*ORIGIN\s*\n)(.*)',record,
                   flags = re.M | re.S)
    assert mo
    annotation = mo.group(1)
    sequence = re.sub('[\s0-9]','', mo.group(2))
    yield annotation, sequence
```

searchGB.py: apply some searches to Genbank-file (1/2)

- here is the first application of `split_genbank` in which we search the annotation and the DNA sequence for some fixed patterns

```
def search_genbank(multiline, regexp):
  poslst = list()
  # match regexp against multiline sequence, be case insensitive
  for mo in re.finditer(regexp, multiline, flags=re.I|re.M):
    poslst.append(mo.start())  # append start position of match
  return poslst

seqnum = 0
for annotation, dna in split_genbank(filename):
  if search_genbank(dna, 'AAA[CG].'):
    print('Sequence found in record {}'.format(seqnum))
  if search_genbank(annotation, 'homo sapiens'):
    print('Annotation found in record {}'.format(seqnum))
  seqnum += 1
```

searchGB.py: apply some searches to Genbank-file (2/2)

```
$ searchGB.py Library.gb
```

```
Sequence found in record 0
Annotation found in record 0
Sequence found in record 1
Annotation found in record 1
Sequence found in record 2
Annotation found in record 2
Sequence found in record 3
Annotation found in record 3
Sequence found in record 4
Annotation found in record 4
```

parseAnno.py: parse some annotations into a dict. (1/3)

- the next step is to split the annotation string from a Genbank record into units, where each unit consists of a keyword and the text belonging to that keyword
- to understand this, here (again) is a short part of a Genbank record

```
LOCUS       AB031069     2487 bp    mRNA            PRI       27-MAY-2000
DEFINITION  Homo sapiens PCCX1 mRNA for protein containing CXXC domain 1,
            complete cds.
ACCESSION   AB031069
VERSION     AB031069.1  GI:8100074
KEYWORDS    .
SOURCE      Homo sapiens embryo male lung fibroblast cell_line:HuS-L12 cDNA to
            mRNA.
  ORGANISM  Homo sapiens
            Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
            Mammalia; Eutheria; Primates; Catarrhini; Hominidae; Homo.
```

- here the units would consist of 1, 2, 1, 1, 1, and 5 lines, respectively

parseAnno.py: parse some annotations into a dict. (2/3)

parsing method:

- given the annotation string, return a dictionary with the field names (e.g. `LOCUS`, `ACCESSION`) as keys and field contents as values.
- annotation fields all begin with a keyword in capital letters at the beginning of a line.
- access these top level strings, treat them as keys and store the corresponding lines as values.
- each top level keyword, matched by `\n([A-Z])`, is first prefixed with a special character not appearing in the original Genbank entry
- the resulting modified string is then split into a list at the position where this special character was inserted
- the keywords are then extracted by trying to match capital letter words only at the beginning of a line.
- these words are used as keys of a dictionary which store the line contents as values

parseAnno.py: parse some annotations into a dict. (3/3)

```
def parse_annotation(annotation):
  results = OrderedDict() # entries are ordered by time of input

  # mark beginnings with special character and split there
  sep = '\001'   # \1 is back reference to first group
  tops = re.sub('\n([A-Z])', '\n{}\\1'.format(sep),\
                annotation).split(sep)
  for value in tops:
    # get key from line, mo is match object
    # the BASE COUNT has a space in it, treat separately
    mo = re.search(r'^(BASE COUNT|[A-Z]+)',value)
    if mo:
      key = mo.group(1)
    else:
      sys.stderr.write('{}: Cannot find key in line {}\n'
                       .format(sys.argv[0],value))
      exit(1)
    results[key] = value   # store value in the dictionary
  return results
```

getAnno.py: get annotation-dict. from Genbank file (1/2)

- here is an application of the `parse_annotation`-method

```
import sys, re
from splitGB import split_genbank
from parseAnno import parse_annotation

if len(sys.argv) != 2:
  sys.stderr.write('Usage: {} <genbankfile>\n'.format(sys.argv[0]))
  exit(1)

filename = sys.argv[1]
for annotation, dna in split_genbank(filename):
  fields = parse_annotation(annotation)
  for key, value in fields.items():
    stars = '*' * 8
    print('{} {} {}'.format(stars,key,stars))
    print(value, end='')
  break # only output first
```

- the output of this program is shown on the next page

getAnno.py: get annotation-dict. from Genbank file (2/2)

```
$ ./getAnno.py Library.gb
```

```
******** LOCUS ********
LOCUS       AB031069     2487 bp    mRNA            PRI       27-MAY-2000
******** DEFINITION ********
DEFINITION  Homo sapiens PCCX1 mRNA for protein containing CXXC domain 1,
            complete cds.
******** ACCESSION ********
ACCESSION   AB031069


⋮

******** FEATURES ********
FEATURES             Location/Qualifiers
     source          1..2487
                     /organism="Homo sapiens"
                     /db_xref="taxon:9606"
                     /sex="male"


⋮

******** BASE COUNT ********
BASE COUNT      564 a    715 c    768 g    440 t
******** ORIGIN ********
ORIGIN
```

parseFeatures.py: extract features from the FEATURES field of a Genbank-record (1/2)

- the feature table can contain several entries (called feature entries), each of which is matched by the following RE:

```
^( {5}\S.*\n(^ {21}\S.*\n)*)
```

```
FEATURES             Location/Qualifiers
     source          1..2487
                     /organism="Homo sapiens"
                     /db_xref="taxon:9606"
                     /sex="male"
                     /cell_line="HuS-L12"
                     /cell_type="lung fibroblast"
                     /dev_stage="embryo"
     gene            229..2199
```

⋮

```
                     /translation="MEGDGSDPEPPDAGEDSKSENGENAPIYCICRKPDINCFMIGCD
```

⋮

```
                     AMTNRAGLLALMLHQTIQHDPLTTDLRSSADR"
```

parseFeatures.py: extract features from the FEATURES field of a Genbank-record (2/2)

- a feature entry thus begins with a keyword after 5 white spaces ...
- followed by a none-white space, ...
- followed by any number of characters until a newline appears
- the 2nd part of feature entry can occur any number of times, begins with 21 blanks and otherwise has the same structure as the first part
- using `re.finditer` we can easily extract the feature entries from the multiline input string
- with `.group(0)` we extract the string matching the entire RE and yield it

```
def parse_features(features):
  for m in re.finditer('^ {5}\S.*\n( {21}\S.*\n)*',\
                       features, flags = re.M):
    yield m.group(0)
```

features.py: extract feature entries from Genbank-file (1/2)

- the following code iterates over the feature entries enumerated by `parse_features`
- using a RE, we extract the feature name and output it on single line enclosed in stars, before the feature entry itself is output

```
for annotation, dna in split_genbank(filename):
  fields = parse_annotation(annotation)
  for featureentry in parse_features(fields['FEATURES']):
    mo = re.search('^ {5}(\S+)', featureentry)
    if mo:
      stars = '*' * 8
      feature_name = mo.group(1)
      print('{} {} {}'.format(stars,feature_name,stars))
      print(featureentry, end = '')
  break # only output first
```

features.py: extract feature entries from Genbank-file (2/2)

```
$ ./features.py Library.gb
```

```
******** source ********
     source          1..2487
                     /organism="Homo sapiens"
                     /db_xref="taxon:9606"
                     /sex="male"
                     /cell_line="HuS-L12"
                     /cell_type="lung fibroblast"
                     /dev_stage="embryo"
******** gene ********
     gene            229..2199
                     /gene="PCCX1"
******** CDS ********
     CDS             229..2199
                     /gene="PCCX1"


⋮

                     YESQTSFGSMYPTRIEGATRLFCDVYNPQSKTYCKRLQVLCPEHSRDPKVPADEVCGC
                     PLVRDVFELTGDFCRLPKRQCNRHYCWEKLRRAEVDLERVRVWYKLDELFEQERNVRT
                     AMTNRAGLLALMLHQTIQHDPLTTDLRSSADR"
```

Taxonomy trees

Taxonomy (in Biology)

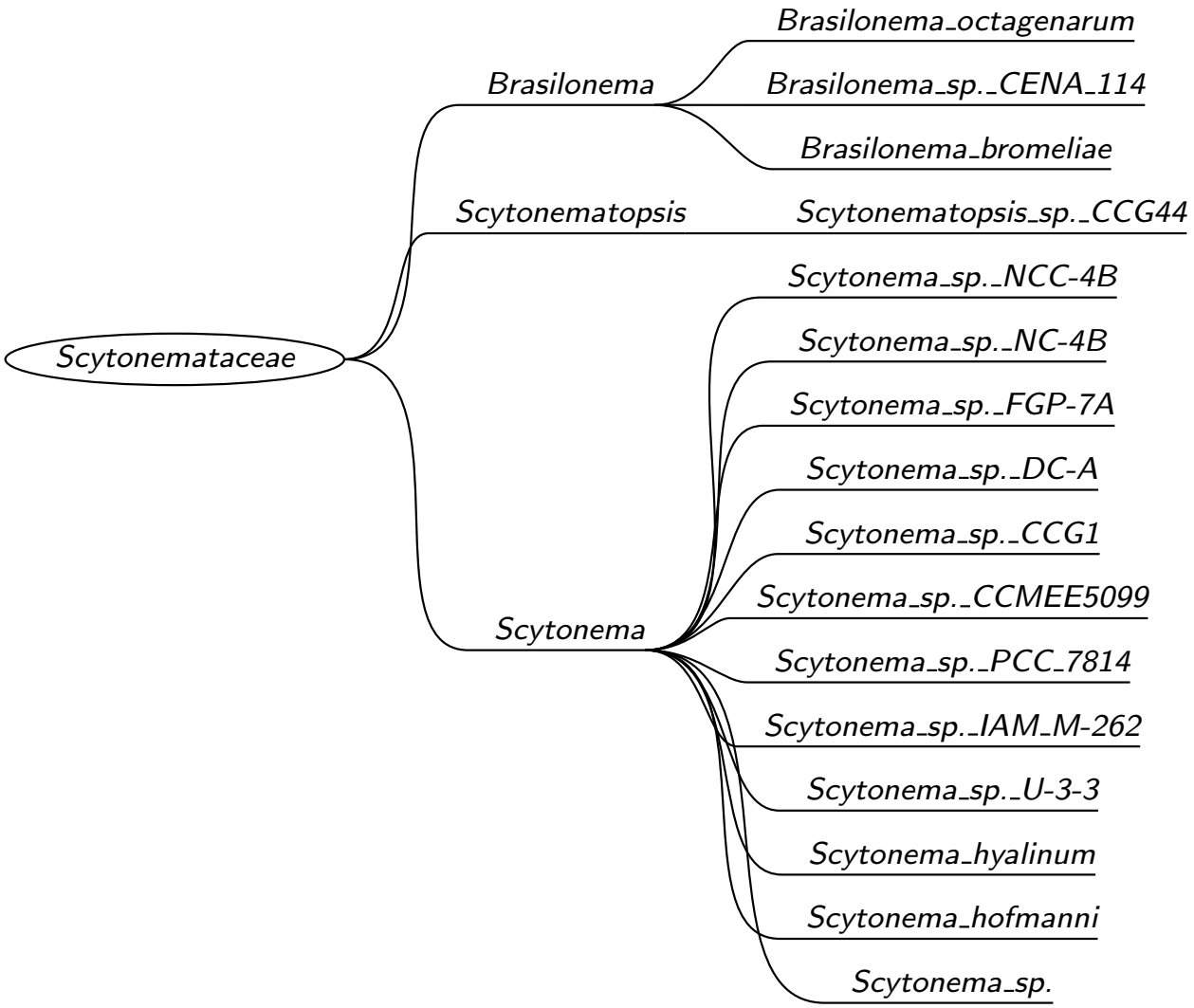
- define groups of biological organisms on the basis of shared characteristics
- give names to those groups
- organisms are grouped together into taxa (singular: taxon) and given a taxonomic rank
- groups of a given rank can be aggregated to form a super group of higher rank and thus create a taxonomic hierarchy of levels
- this hierarchy is usually shown in form of a tree, depicted later

1. domain
2. superkingdom
3. superphylum
4. phylum/subphylum
5. class/subclass
6. order/suborder
7. family/subfamily
8. tribe
9. genus/subgenus
10. species/subspecies
11. strain/substrain
12. varietas
13. forma

Distribution of ranks of taxa in NCBI taxonomy of bacteria

num	rank	occurrences
0	superkingdom	1
1	superphylum	3
2	phylum	63
3	subphylum	25
4	class	69
5	subclass	3
6	order	174
7	suborder	7
8	family	375
9	subfamily	1
10	tribe	2
11	genus	2 956
12	subgenus	3
13	species group	63
14	species subgroup	11
15	species	333 152
16	subspecies	586
17	varietas	19
18	forma	3
19	no rank	40 494

A small part of the bacterial taxonomy tree for the *Scytonemataceae*-family of bacteria



A simple text format for representing taxonomy trees (1/2)

- here we consider how to parse these trees from simple input-formats and show how to manipulate them
- input format: tree is expression consisting of parentheses enclosing names of taxa and taxonomic groups

- opening parentheses (starts a tree level
- closing parentheses) completes a tree level
- name of group follows)
- members of the same group separated by commas
- format closes with ;

Example

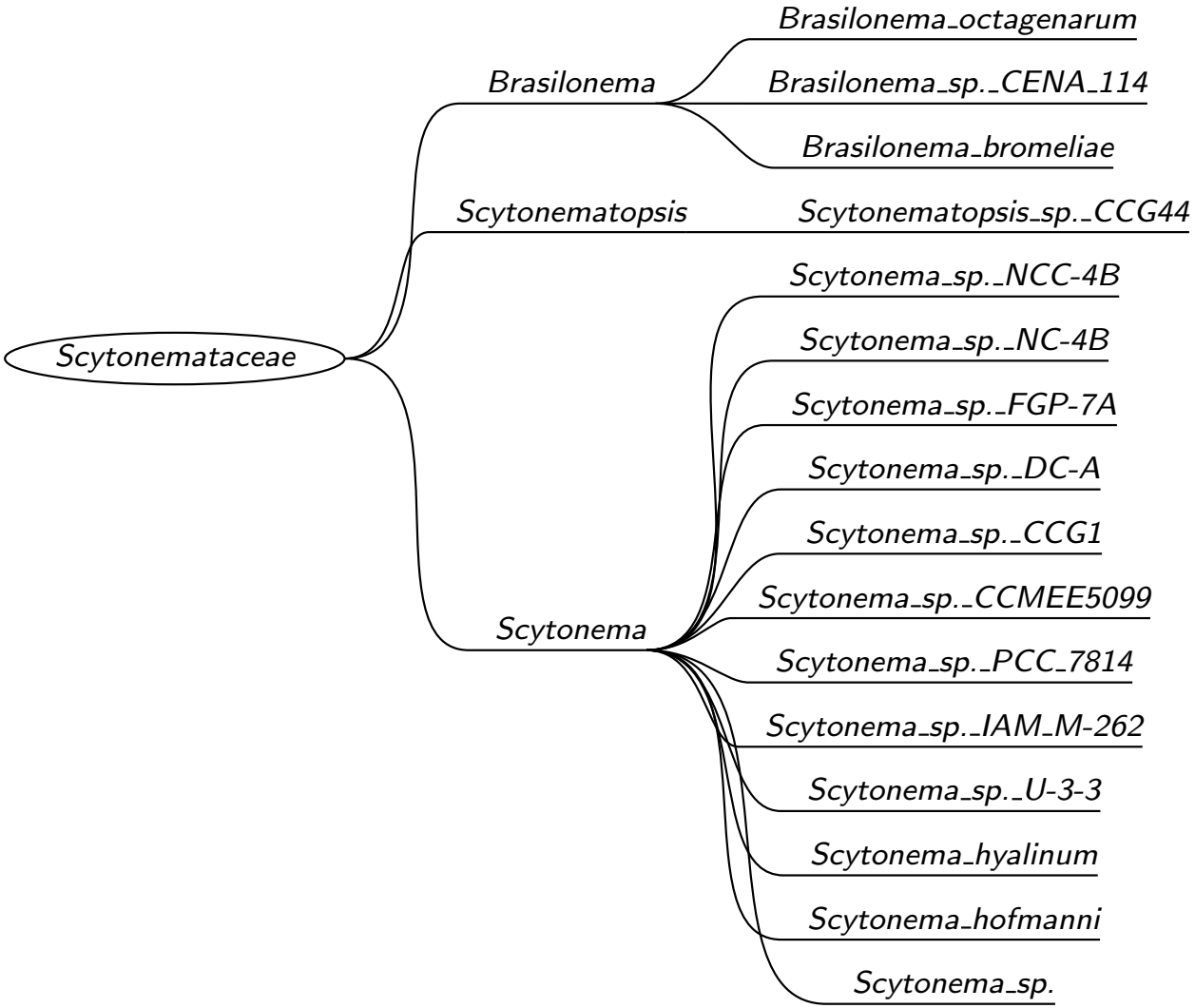
`((L0,L1)B0,(L2)B1,(L3,L4,L5)B2)B3;`

- L0, L1, . . . , L5 are names of leaf nodes
- B0, B1, . . . , B3 are names of branch nodes

- right column shows example with units split on different lines

```
(
  (
    L0,
    L1
  )B0,
  (
    L2
  )B1,
  (
    L3,
    L4,
    L5
  )B2
)B3;
```

A simple text format for representing taxonomy trees (2/2)




```
(
  (
    Scytonema_sp.,
    Scytonema_hofmanni,
    Scytonema_hyalinum,
    Scytonema_sp._U-3-3,
    Scytonema_sp._IAM_M-262,
    Scytonema_sp._PCC_7814,
    Scytonema_sp._CCMEE5099,
    Scytonema_sp._CCG1,
    Scytonema_sp._DC-A,
    Scytonema_sp._FGP-7A,
    Scytonema_sp._NC-4B,
    Scytonema_sp._NCC-4B
  )Scytonema,
  (
    Scytonematopsis_sp._CCG44
  )Scytonematopsis,
  (
    Brasilonema_bromeliae,
    Brasilonema_sp._CENA_114,
    Brasilonema_octagenarum
  )Brasilonema
)Scytonemataceae;
```

part of file with taxonomy tree for bacteria (maximal depth: 12, branching nodes: 10 696, leaves: 362 279)

Parsing the lexical units of the tree format (1/7)

- usually no indentation and line breaks, but here shown for clarity

- format is almost identical to popular Newick format
  - this is often used for phylogenetic trees
  - no names for branching nodes
  - additionally branch length can be represented

- we also use the notion Newick

```
(
  (
    L0,
    L1
  )B0,
  (
    L2
  )B1,
  (
    L3,
    L4,
    L5
  )B2
)B3;
```

Parsing lexical units

- first step in parsing the tree format is to separate the input into lexical units, using the following regular expression:

  `\)?[^\(\),;]+[,;]?|\(`

- the first part before the symbol `|`, matches a string with three parts:
  1. an optional `)`
  2. a non-empty string not consisting of brackets, comma and semi colon ⇒ name of branch node or leaf
  3. an optional comma or semi colon
- the second part after the symbol `|` matches `(`

Parsing the lexical units of the tree format (2/7)

- so the expression

```
re.findall(r'\)?[^\(\) ,;]+[,;]?|\(',
            '((L0,L1)B0,(L2)B1,(L3,L4,L5)B2)B3;')
```

  will deliver the list

```
['(','(','L0,','L1',')B0,','(','L2',')B1,',
 '(','L3,','L4,','L5',')B2',')B3;']
```

- this was shown line by line and indented on the previous frame
- the next step is to transform the strings delivered by `findall` into one of the three possible lexical units:

input	lexical unit
`(`	`(OPEN,None)`
`)string`	`(CLOSE,string)`
`string`	`(LEAF,string)`

```
[(OPEN, None), (OPEN, None), (LEAF, 'L0'),
 (LEAF, 'L1'), (CLOSE,'B0'), (OPEN, None),
 (LEAF, 'L2'), (CLOSE, 'B1'), (OPEN, None),
 (LEAF,'L3'), (LEAF, 'L4'), (LEAF, 'L5'),
 (CLOSE, 'B2'), (CLOSE, 'B3')]
```

- note: trailing comma and semi colon are omitted in the lexical unit

Parsing the lexical units of the tree format (3/7)

- before we explain how to implement a function implementing the transformation into lexical units, we have to introduce a class for the three kinds of lexical units

```
from enum import Enum
class LexicalKind(Enum):
  OPEN = 0
  CLOSE = 1
  LEAF = 2
  def __str__(self):
    return self.name
```

- here we introduce an enumeration class, comprised of a set of symbolic names (members) bound to unique, constant values
- within an enumeration, the members can be compared by identity, and the enumeration itself can be iterated (see `https://docs.python.org/3/library/enum.html`)

Parsing the lexical units of the tree format (4/7)

- in our case we have introduced three values for the three different kinds of lexical units
- we use the values 0, 1, and 2 for the constants, but could have used other values, as long as they are different
- as the constants are members of the class, we denote them by `LexicalKind.OPEN` etc, but will omit the prefix `LexicalKind.` when showing them as strings (see `__str__`)
- a lexical unit is then a pair consisting of a `LexicalKind`-value and optionally a string if the lexical kind is not `LexicalKind.OPEN`.

Parsing the lexical units of the tree format (5/7)

```
def lexical_units_enum(inputfile):
  if inputfile == '-':
    stream = sys.stdin
  else:
    try:
      stream = open(inputfile,'r')
    except IOError as err:
      sys.stderr.write('{}: {}\n'.format(sys.argv[0],err))
      exit(1)
  for line in stream:
    line = re.sub(r'\s*','',line) # delete all white spaces
    for match in re.findall(r'\)?[^\(\),;]+[,;]?|\(',line):
      if match == '(':
        yield (LexicalKind.OPEN,None)
      else:
        mo = re.search(r'\)(?P<branch_name>[^,;]+)[,;]?',match)
        if mo: # match )branch_name optionally followed by , or ;
          yield (LexicalKind.CLOSE,mo.group('branch_name'))
        else: # no ( and ) => leaf
          yield (LexicalKind.LEAF,re.sub(r',$','',match))
  if inputfile != '-':
    stream.close()
```

Parsing the lexical units of the tree format (6/7)

- note that we used name captures in the subexpression `(?P<branch_name>[^,;]+)`
- in case of a match to the entire expression, this introduces a name for the string matching this subexpression
- instead of calling `group` with a number, we use the introduced name as a string argument for `group`

- for debugging purposes we implement a function which shows the lexical units indented according to their level in the tree (which we want to construct)

```
def lexical_units_indent(lu_stream):
  level = 0
  indent = lambda level : ' ' * (2 * level)
  for lu in lu_stream:
    if lu[0] == LexicalKind.OPEN:
      print('{}{}'.format(indent(level),lu[0]))
      level += 1
    else:
      if lu[0] == LexicalKind.CLOSE:
        assert level > 0
        level -= 1
      print('{}{} {}'.format(indent(level),
                              lu[0],lu[1]))
```

Parsing the lexical units of the tree format (7/7)

- suppose we have combined the functions `lexical_units_enum` and `lexical_units_indent` into a program `lexical_units.py`
- let the file `toy.tre` contain the string `((L0,L1)B0,(L2)B1,(L3,L4,L5)B2)B3;`

```
$ ./lexical_units.py toy.tre
```

```
OPEN
  OPEN
    LEAF L0
    LEAF L1
  CLOSE B0
  OPEN
    LEAF L2
  CLOSE B1
  OPEN
    LEAF L3
    LEAF L4
    LEAF L5
  CLOSE B2
CLOSE B3
```

Representing trees with nodes (1/2)

- we represent a node of the tree by its name and a list of successors
- so introduce class `Node` with the member variables `name` and `successors`

```
class Node:
  def __init__(self,name,successors):
    self.name = name
    self.successors = successors
```

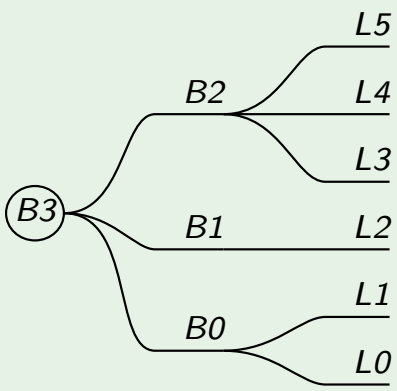
- successors-list can contain Node-instances
- leaves have an empty successors-list

Example

The tree denoted by
`((L0,L1)B0,(L2)B1,(L3,L4,L5)B2)B3;`
will be represented by the Python-expression

```
Node('B3',[Node('B0',[Node('L0',[]),Node('L1',[])]),
           Node('B1',[Node('L2',[])]),
           Node('B2',[Node('L3',[]),
                      Node('L4',[]),
                      Node('L5',[])])])
```

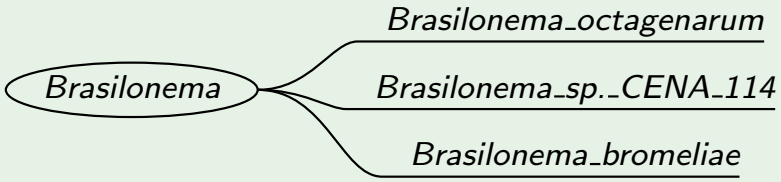
Here is a graphical representation:



Representing trees with nodes (2/2)

Example

the *Brasilonema*-subtree



is represented by the following structure:

```
Node('Brasilonema',[Node('Brasilonema_bromeliae',[]),
                    Node('Brasilonema._sp._CENA_114',[]),
                    Node('Brasilonema_octagenarum',[])])
```

Parsing the taxonomy tree from the lexical units (1/4)

- next construct from the stream of lexical units a corresponding tree structure
- due to nesting of the tree structure, the `OPEN`- and `CLOSE`-lexical units for the same branching node are at arbitrary distance in the stream
- so we need a stack on which we push, for each `OPEN`, a new branching node with undefined name and empty list of successors
- once we see a `CLOSE`-lexical unit, we know that it corresponds to the uncompleted branching node at the top of the stack
- so each `CLOSE` delivers the name of the branching node on top of the stack
- moreover, with a `CLOSE` all nodes which are children of this top-stack nodes have been seen and thus the top-stack node is completed and popped from the stack
- once a node ist completed, it is appended to the successors list of the node then on top of the stack

Parsing the taxonomy tree from the lexical units (2/4)

Example

```
OPEN
  OPEN
    LEAF L0
    LEAF L1
  CLOSE B0
  OPEN
    LEAF L2
  CLOSE B1
  OPEN
    LEAF L3
  CLOSE B2
CLOSE B3
```

- when reading lexical unit `LEAF L0`, two uncompleted branch nodes are on the stack
- leaf with name `L0` will be added to the succ.-list of the node on the top of the stack (top-node, for short)
- when reading `CLOSE B0`, two uncompleted branch nodes are on the stack and one completes the top-node by naming it `B0`
- we pop the top-node from the stack and add it to the succ.-list of the then top-node (which is the only remaining element on the stack)
- when reading `CLOSE B3`, one uncompleted branch nodes is on the stack and so complete the top-node by naming it `B3`
- we pop it from the stack and since the stack is empty, it must be the root; so we store it in a corresponding member variable

Parsing the taxonomy tree from the lexical units (3/4)

– while the functions related to the parsing of lexical units are in their own module, we implement the method for constructing the tree in a class `Taxtree`

```
class Taxtree:
  def __init__(self,inputfile):
    self._root = None   # instance var to hold root of tax. tree
    stack = list()
    for lex_unit in lexical_units_enum(inputfile):
      if lex_unit[0] == LexicalKind.OPEN: # new node no name yet
        newbranch = Node(None,list())
        stack.append(newbranch) # stack top contains current node
      elif lex_unit[0] == LexicalKind.CLOSE: # complete branch node
        assert stack  # stack must contain at least one br.node
        brnode = stack.pop()
        brnode.name = lex_unit[1] # store name of br. node
        if not stack:  # if stack is empty
          self._root = brnode   # now tree is complete
        else: # parent of brnode is top element of stack
          stack[-1].successors.append(brnode)
```

Parsing the taxonomy tree from the lexical units (4/4)

```
    else: # lexical unit is LEAF
      assert lex_unit[0] == LexicalKind.LEAF and stack
      # leaf with empy list of successors
      newleaf = Node(lex_unit[1],list())
      # parent of leaf is top element of stack
      stack[-1].successors.append(newleaf)
  assert not stack and self._root is not None
  self._name_map = dict()
  for node, depth, parent in self.enum_nodes():
    assert node.name not in self._name_map
    self._name_map[node.name] = (node, depth, parent)
def root(self):
  return self._root
```

- `__init__` iterates over the lexical units and handles them as described
- the tree structure can be accessed via the member variable `_root`, which stores the root node
- after completing the tree structure, all nodes with their depth and parent are enumerated (using a method shown later)
- this information is stored in a dictionary with the node name as key

Output of tree in the original format (1/2)

– to test the parser, output constructed tree in original format (using the following recursive method) and compare it with original file

```
def to_newick_rec(self,fo,depth,node,last_successor):
  indentation = ' ' * (2 * depth)
  fo.write(indentation + '(\n')  # start of subtree for br. node
  lastidx = len(node.successors) - 1 # output successor subtrees
  for idx, subnode in enumerate(node.successors):
    if not subnode.successors:  # subnode is leaf
      termsymbol = '' if idx == lastidx else ','
      fo.write(indentation + '  {}{}\n'
               .format(subnode.name,termsymbol))
    else:
      self.to_newick_rec(fo,depth+1,subnode,idx == lastidx)
  if last_successor:  # is node last successor of parent?
    termsymbol = ''
  else:
    termsymbol = ',' if depth > 0 else ';'
  # now output name of branching node
  fo.write(indentation + '){}{}\n'.format(node.name,termsymbol))
```

Output of tree in the original format (2/2)

- the main difficulty when producing the newick-formatted output, is to correctly place commas and semi colons, as in `((L0,L1)B0,(L2)B1,(L3,L4,L5)B2)B3;`
- commas separate the successors with the same parent node
- so add a comma after each successor except for the last
- for a leaf, we can simply append a comma after the name, if it is not the last successor, as for `L0`, `L3`, and `L4`
- for a branching node, we pass the information whether it is the last successor of its parent as an additional parameter named `last_successor`
- this is evaluated after processing all successors and when writing the name of the node, as for `B0` and `B1`
- in the special case that we output the name of the root node, we add a semi colon, as for `B3`
- to output the complete tree, use this function:

```
def to_newick(self,fo = sys.stdout):
  self.to_newick_rec(fo,0,self._root,False)
```

Computing statistics on the tree (1/3)

- the next task is to generate some statistics on the distribution of nodes of different depths for a given taxonomic tree
- this can conveniently be done by enumerating the nodes together with their level and a reference to the parent
- for class `Taxtree` we prefer an iterative method (already used on frame 441), which stores branching nodes and their depth on a stack

```
def enum_nodes(self):  # depth first enumeration
  stack = list()
  stack.append((self._root,0))
  yield self._root, 0, None  # root.parent is None
  while stack:
    branchnode, depth = stack.pop()
    for subnode in branchnode.successors:
      if subnode.successors:  # a branching node?
        stack.append((subnode,depth+1))
      yield subnode, depth+1, branchnode
```

- the parent of the root is `None`, a value referred to later

Computing statistics on the tree (2/3)

- the `statistics`-method enumerates the nodes, maintains their maximal depth, counts the number of leaves and branching nodes, and determines the distribution of the depths of leaves/branching nodes
- `defaultdict(int)` ⇒ dictionaries with default value 0
- add `from collections import defaultdict`

```
def statistics(self):
  maxdepth, numleaves, numbranching = 0, 0, 0
  leaves_depth_dist = defaultdict(int)
  branch_depth_dist = defaultdict(int)
  for node, depth, parent in self.enum_nodes():
    if maxdepth < depth:
      maxdepth = depth
    if not node.successors:
      numleaves += 1
      leaves_depth_dist[depth] += 1
    else:
      numbranching += 1
      branch_depth_dist[depth] += 1
  return maxdepth, numbranching, numleaves, \
         branch_depth_dist, leaves_depth_dist
```

Computing statistics on the tree (3/3)

Example

- Consider the NCBI taxonomy tree for all bacteria.
- The subtree of all *Escherichia coli*-species has depth 3 with 101 branching nodes and 2 971 leaves with the following depth-distribution

depth	leaves of this depth	branch nodes of this depth
0	0	1
1	2391	96
2	487	4
3	93	0

Accessing information of tree nodes by name

- recall that the class has an instance variable `self.name_tab` (see frame 441) which maps every node name to its node, its depth and its parent
- to make this information accessible, we add three corresponding methods:

```
def find_node(self,name):
  assert name in self._name_map
  node, depth, parent = self._name_map[name]
  return node

def find_depth(self,name):
  assert name in self._name_map
  node, depth, parent = self._name_map[name]
  return depth

def find_parent(self,name):
  assert name in self._name_map
  node, depth, parent = self._name_map[name]
  return parent
```

Finding paths in the tree (1/2)

- an important task is to determine the names of the nodes on the path from the root to a given taxon in the tree
- we use the available parent information for finding the path

```
def find_path(self,name):
  path = list()
  node = self.find_node(name)
  while node:
    path.append(node.name)
    node = self.find_parent(node.name)
  return list(reversed(path))
```

- the `while` loop stops, once we appended a node whose parent is `None`
- as the root is the only node whose parent is `None` we always append the name of the root as last element to `path`
- the `while`-loop constructs the path backwards, so that the name of the root-node is at the end of the list
- as we want it at the start of the list, we finally reverse `path`

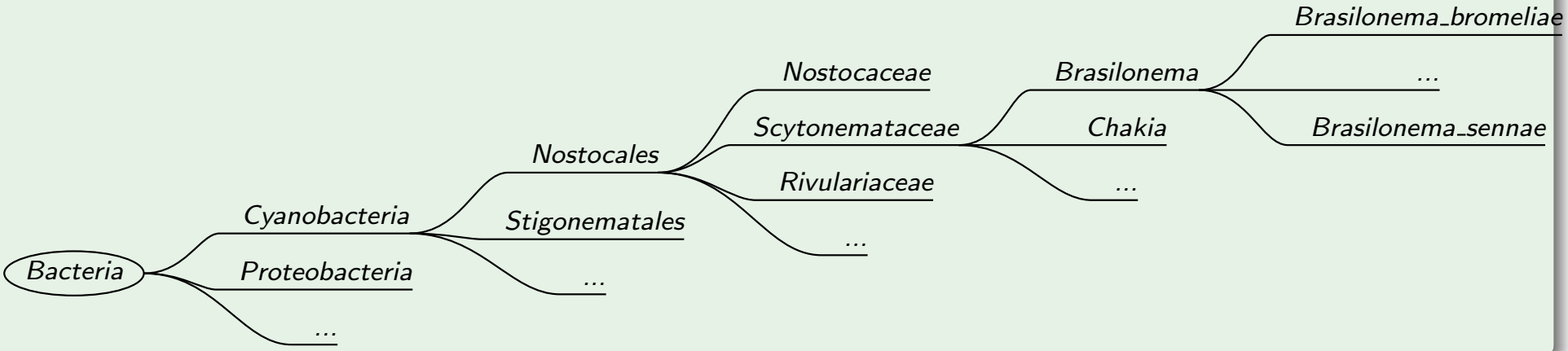
Finding paths in the tree (2/2)

Example

- consider the NCBI taxonomy tree for all bacteria
- for the taxon *Brasilonema bromeliae* we obtain the following path

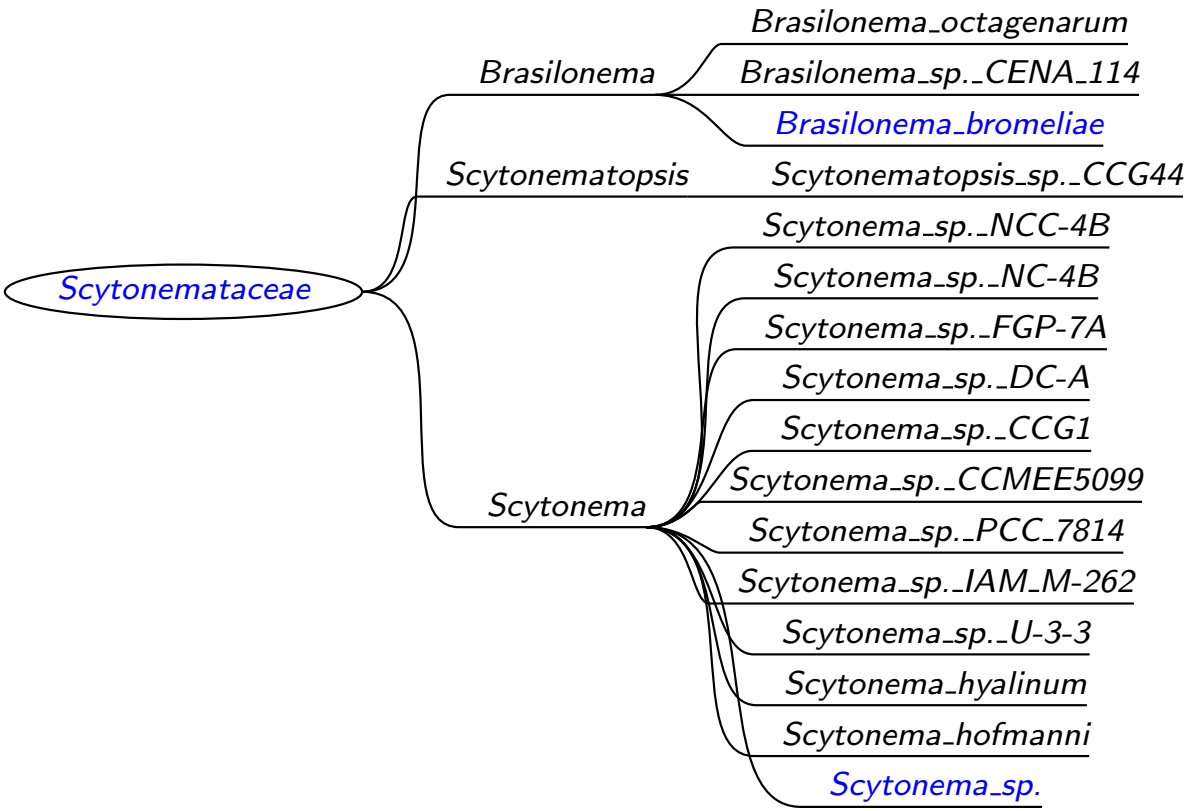
```
[Bacteria', 'Cyanobacteria', 'Nostocales', 'Scytonemataceae',
'Brasilonema', 'Brasilonema_bromeliae']
```

- Here is the path with little context in the complete tree



From paths to lowest common ancestors

- the next task is to find the lowest common ancestor (LCA, for short), i.e. the name of the node of maximum depth on the paths for two different taxa
- for example, in the following tree, the LCA of Brasilonema_bromeliae and Scytonema_sp. is Scytonemataceae.



- finding the LCA is the most important task for taxonomy trees
- LCA-computation will be left as an exercise

XML: eXtensible Markup Language [9]

- the previous sections (e.g. on the Genbank-format or on taxonomy trees) show that parsing a format which was designed in the 1980's can be very tedious
- there are many other historical text file formats in different areas of the sciences which are often even more complicated to parse
- to relief programmers from the burden of parsing such formats, a more structured and generic representation of information would be desirable
- the most widely used such format is XML, discussed in this section

[9] content of slides partially derived from
`http://www.info.univ-angers.fr/~richer/ibss2011/ibss2011_richer_crs3.pdf`

XML: eXtensible Markup Language (1/1)

- set of rules for encoding documents in machine-readable format
- text format, also readable for humans
- uses tags to mark information: `<html>` ... `</html>`
- tags can have attributes and they can be nested:

```
<taga attribute="attrval">
  <tagb>
    some text
  </tagb>
  <tagb>
    some other text
  </tagb>
</taga>
```

- indentation and line breaks are not required
- comment: `<!-- a comment -->`

XML: eXtensible Markup Language (1/6)

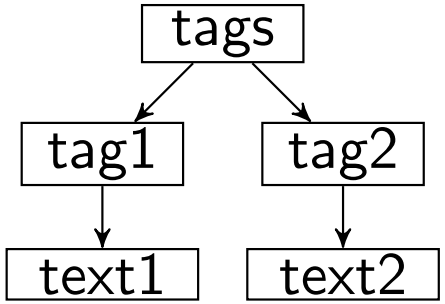
tags

enable to define information:

```
<tag> text </tag>
```

- here tag is assigned a *text*
- tags can be nested ⇒ representation of hierarchical information (trees)

```
<tags>
  <tag1> text1 </tag1>
  <tag2> text2 </tag2>
</tags>
```



XML: eXtensible Markup Language (2/6)

tags and attributes

- attributes enable to add semantic information:

```
<tag attribute="attrval" ...> text </tag>
```

- this means that `tag` encloses a text for which `attribute` is set to `attrval`
- example:

```
<title language="english">Learning Python</title>
<title language="german">Erlernen von Python</title>
```

XML: eXtensible Markup Language (3/6)

example of XML document:

- we want to describe books in a library
- each book is defined by
- an integer identifier (1, 2, ...)
- a title
- a list of authors
- an editor
- a price expressed in a given currency
- a list of keywords

XML: eXtensible Markup Language (4/6)

```
<library>
<book id="1">
  <title>learn XML in 10 seconds</title>
  <list_of_authors>
    <author first="John" last="Fast" />
    <author first="Tom" last="Doyle" />
  </list_of_authors>
  <editor> Harald Smith </editor>
  <price currency="dollar">40</price>
  <list_of_keywords>
    <keyword>XML</keyword>
  </list_of_keywords>
</book>
</library>
```

XML: eXtensible Markup Language (5/6)

title of book

Simple string of characters:

```
<title>learn XML in 10 seconds</title>
```

the authors

there can be several authors ⇒ we use a tag inside which we put the authors:

```
<list_of_authors>
  <author ... />
  ...
 </list_of_authors>
```

XML: eXtensible Markup Language (6/6)

an author

- is defined by its first and last names:

```
<author first="John" last="Fast" />
```

  first and last are the attributes of author

- we also could have used another level:

```
<author>
  <first_name>John</first_name>
  <last_name>Fast</last_name>
</author>
```

  but this would add too much text

advantages of XML for computer scientists

- built-in document validation (DTD or XML schemas):
  - XML files should (but not required to) start with XML prolog:
    ```
    <?xml version="1.0" encoding="utf-8" ?>
    <!DOCTYPE library SYSTEM "library.dtd" >
    ```
  - here grammar for the XML-document is stored in `library.dtd`

  ⇒ allows for validation of the XML-document, e.g. using the `xmllint` program

Example

Suppose we have forgotten the last closing tag `</library>` in the XML-file `library.xml` shown on page 456.

the tool `xmllint` reports this error:

```
$ xmllint --noout library.xml
library.xml:16: parser error: Premature end of data in tag library line 3
```

advantages of XML for computer scientists

more advantages:

- XML is readable and modifiable
- XML is extensible by introducing new tags
- parsing XML is generic and much simpler than self-defined formats (like Genbank or Newick)
- almost all languages provide classes or software libraries for parsing XML
- platform independent (OS, language)

disadvantages of XML

- verbose ⇒ large documents with many redundancies ⇒ compress it
- best level of description is often not obvious: attributes or tags (as in the authors example)

XML in Bioinformatics (1/2)

In Bioinformatics a variety of heterogeneous data formats is used:

- sets of sequences
- result of alignment, phylogeny
- 3D-representation of protein folding
- small molecules
- annotations, microarrays

lack of consistency even for the basics ⇒ automatic processing made difficult

the presence of so many different formats in bioinformatics makes this an important application domain for XML

XML in Bioinformatics (2/2)

XML formats for Bioinformatics

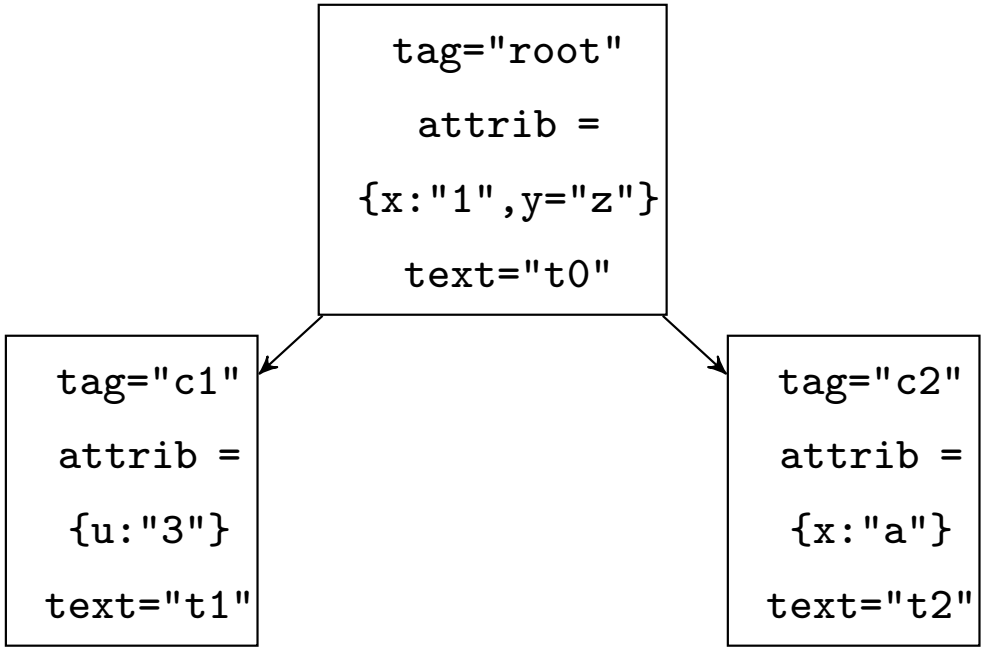
- AGAVE (Architecture for Genomic Annotation, Visualization and Exchange)
- BIOML (BIOpolymer Markup Language)
- BSML (Bioinformatic Sequence Markup Language)
- HSAML (Multiple sequence alignment format with guide tree data and quality scores)
- MAGE-ML (XML format for microarray data conforming to the MAGE object model)
- NCBI
- PEML (Proteomics Experiment Markup Language)
- PSI-MI (Proteomics Standard Initiative Molecular Interaction)
- SBML (Systems Biology Markup Language)
- UniProt XML

Parsing XML in Python

- now parse an XML-file using the `xml.etree`-module of Python
- it provides a class `ElementTree` with a function `parse` to deliver the tree
- in this tree XML-tag is represented by an instance `node` of class `ElementTree` with at least three member variables:
  - `node.tag` is the XML-tag
  - `node.text` is the `text`
  - `node.attrib` is a dictionary of attribute/value pairs

- child-nodes can be obtained by an iteration over `node`

```
<root x="1" y="z">
  t0
  <c1 u="3"> t1 </c1>
  <c2 x="a"> t2 </c2>
</root>
```



Parsing a Genbank record in XML format (1/2)

- the following Genbank entry in XML only contains a few tags
- the original entry from NCBI is much larger and with tags named `INSDSeq`, `INSDSeq_locus` etc.

```
<Seq>
  <Seq_locus>AAU04286</Seq_locus>
  <Seq_length>436</Seq_length>
  <Seq_moltype>AA</Seq_moltype>
  <Seq_topology>linear</Seq_topology>
  <Seq_division>BCT</Seq_division>
  <Seq_create-date>19-AUG-2004</Seq_create-date>
  <Seq_definition>citrate (Si)-synthase; (R)-citric synthase.; Citrate
                  condensing enzyme.;
                  Citrate oxaloacetate-lyase, CoA-acetylating.;
                  Oxaloacetate transacetase.
                  [Rickettsia typhi str. Wilmington]</Seq_definition>
  <Seq_source>Rickettsia typhi str. Wilmington</Seq_source>
  <Seq_organism>Rickettsia typhi str. Wilmington</Seq_organism>
  <Seq_sequence>aacactgtgaattaagagcctatcacaacgagccatactacg</Seq_sequence>
</Seq>
```

- here `Seq_locus`, `Seq_length`, . . . , `Seq_sequence` are first level tags
- each tag has a name (e.g. `Seq_locus`) and a value (which is a text in our case, e.g. `AAU04286`)

Parsing a Genbank record in XML format (2/2)

- now parse an XML-file using the `xml.etree.ElementTree`-class
- to simplify referring to the class, we introduce abbreviation `ET`
- we parse the entire tree from the input file, using `ET.parse`
- we extract information from the level below the root node Seq and output tag/text pairs for all tags from a given set `idset`

```
import xml.etree.ElementTree as ET

# show subtags with text/name
def xmlshownextlevel(idset,node):
  for child in node:
    if child.tag in idset:
      print('{} = {}'
            .format(child.tag,
                    child.text))

# specify which tags produce output

idset = ['Seq_locus', 'Seq_sequence',
          'Seq_division']
xml_tree = ET.parse('Record.xml')
root = xml_tree.getroot()
print('# process {}'.format(root.tag))
xmlshownextlevel(idset,root)
```

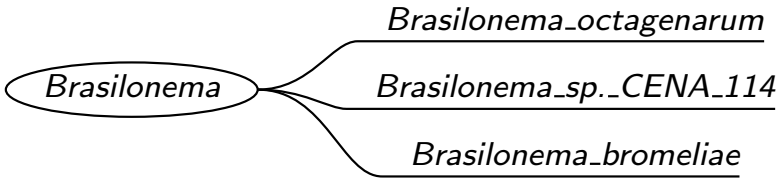
```
$ ./gb_xml_parse.py

# process Seq
Seq_locus=AAU04286
Seq_division=BCT
Seq_sequence=
     aacactgtgaattaagagcctatcacaacgagccatactacg
```

Taxonomy trees as XML-documents

- as XML is a hierarchical format by design, it is natural to output the taxonomic tree in XML-format
- this is easier to read and parsing is simplified
- the XML-formatted tree also contains the taxonomic rank for each node (assuming that it depends on its depth)
- to output this, we introduce the following list, which is stored in a class variable of class `Taxtree`

```
taxonomic_rank = ['domain', 'superkingdom', 'phylum',
                  'subphylum', 'class', 'subclass', 'order',
                  'suborder', 'family', 'subfamiliy', 'tribe',
                  'genus', 'subgenus', 'species', 'subspecies']
```

- in the XML-output, for each node we show its name inside tag-brackets where the tag is the corresponding taxonomic rank
- here is the *Brasilonema*-subtree



in XML-format:

```
<genus> Brasilonema
  <species> Brasilonema_bromeliae </species>
  <species> Brasilonema_sp._CENA_114 </species>
  <species> Brasilonema_octagenarum </species>
</genus>
```

- again we implement a recursive function which traverses the tree
- note that the root of our complete bacteria-tree is at the superkingdom level (which is 1).
- so we have to add 1 to access the correct rank

```
def to_xml_rec(self,fo,depth,node):
  indentation = ' ' * (2 * depth)
  rank = self.taxonomic_rank[depth + 1]
  fo.write(indentation +
           '<{}> {}'.format(rank,self.xmlshow(node.name)))
  if not node.successors:
    fo.write(' </{}>\n'.format(rank))
  else:
    fo.write('\n')
    for subnode in node.successors:
      self.to_xml_rec(fo,depth+1,subnode)
    fo.write(indentation + '</{}>\n'.format(rank))

def to_xml(self,fo,depth = 0):
  self.to_xml_rec(fo,depth,self._root)
```

Handling XML-meta characters

– as some names in the taxonomic tree contain the XML-meta characters <, > or &, we implement a conversion function to correctly show them in XML, using the CDATA-notation.

```
def xmlshow(self,s):
  badchar = '><&'
  if re.search(r'[{}]'.format(badchar),s):
    sl = list()
    for cc in s:
      if cc != '.' and (cc in badchar):
        sl.append('<![CDATA[{}]]>'.format(cc))
      else:
        sl.append('{}'.format(cc))
    return ''.format(sl)
  else:
    return s
```

So, for example, the species `Clostridium_sp._AB&J`, is shown as

```
Clostridium_sp._AB<![CDATA[&]]>J
```

Parsing XML-formatted taxonomic trees (1/3)

- the taxonomic tree stored in XML is much easier to parse than the newick notation
- as before, with a single call to the method `parse` of the `ElementTree`-class we obtain the complete tree structure
- for finding paths in the tree, we want to know the parent of each node
- instead of storing references to a tree node, we map the text of each node (which is unique) to the text of the parent node
- this information is stored in a dictionary `parent_map`
- the root has no parent, and so we store in `parent_map` the value `None` for the text at the root
- all other values are stored in a depth first traversal of the XML-tree, as shown in the `__init__` method of the following class

Parsing XML-formatted taxonomic trees (2/3)

```
import xml.etree.ElementTree as ET

class TaxtreeXML:
  def __init__(self,xmlfilename):
    try:
      xml_tree = ET.parse(xmlfilename)
    except IOError as err:
      sys.stderr.write('{}: {}\n'.format(sys.argv[0],err))
      exit(1)
    self._xml_tree_root = xml_tree.getroot()
    self._parent_map = dict()
    root_name = self._xml_tree_root.text.strip()
    self._parent_map[root_name] = None
    stack = [self._xml_tree_root]

    while stack:
      node = stack.pop()
      assert node.text
      node_text = node.text.strip()
      for child in node:
        stack.append(child)
        self._parent_map[child.text.strip()] = node_text
```

Parsing XML-formatted taxonomic trees (3/3)

- `find_path` is almost identical to `Taxtree.find_path`, except for:
  - the access to the parent is via the dictionary `parent_map` rather than an attribute of an instance of class `Node`
  - the `parent_map` delivers the text at the parent node, while `taxtree.find_path` delivers a reference to the parent

```
def find_path(self,name):
  path = list()
  while name:
    path.append(name)
    name = self._parent_map[name]
  return list(reversed(path))
```

```
def find_path(self,name):
  path = list()
  node = self.find_node(name)
  while node:
    path.append(node.name)
    node = self.find_parent(node.name)
  return list(reversed(path))
```

Taxtree.find_path

List comprehensions (1/16)

- one of the notations in Python we have not yet considered are list comprehensions
- these allow to specify a list by a notation that resembles mathematical notations for sets
- a list comprehension has the general form:

```
[f(x) for x in X if p(x)]
```

where

  - f is a function,
  - X is a set and
  - p is a boolean function (called predicate), which can optionally be used (python-history.blogspot.com/2010/06/from-list-comprehensions-to-generator.html)

List comprehensions (2/16)

Example [https://www.datacamp.com/community/tutorials/python-list-comprehension] List comprehensions on the right construct lists equivalent to the sets on the left. For `listS` and `listV` there are no predicates.

$S = \{x^2 \mid x \in \{0, \ldots, 9\}\}$

$V = \{1, 2, 4, 8, \ldots, 2^{11}\}$

$M = \{m \mid m \in S, m \text{ is even}\}$

```
listS = [x*x for x in range(9+1)]
listV = [2**i for i in range(11+1)]
listM = [m for m in listS \
                 if m % 2 == 0]
```

Let us verify that the appropriate lists are constructed:

```
for name, l in zip(['S','V','M'],[listS,listV,listM]):
  print('{}={}'.format(name,l))
```

```
S=[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
V=[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048]
M=[0, 4, 16, 36, 64]
```

List comprehensions (3/16)

- with a list comprehensions it becomes easy to extract a specific column from a tab separated file (given as a stream):

```
def cut_column(stream,column,sep='\t'):
  return [line.split(sep)[column] for line in stream]
```

- one can e.g. apply the function to the file with data about atoms (see frame 207), this time neglecting appropriate error handling

```
stream = open('../Chemistry/atom-data.tsv')
print(cut_column(stream,2))
stream.close()
```

```
['name', 'Hydrogen', 'Helium', 'Lithium', 'Beryllium', 'Boron', ...]
```

- we can generalize the previous function by allowing to extract those columns specified as numbers in a list `extract_columns`:

```
def cut(stream,extract_columns,sep='\t'):
  return [[line.split(sep)[i] for i in extract_columns]\
          for line in stream]
```

List comprehensions (4/16)

- we use a list comprehension
  - to extract the lines from the stream (outer `for`-loop)
  - to extract the elements from each line (inner `for`-loop)

```
print(cut(stream,[1,3]))
```

```
[['symbol', 'atomicMass'], ['H', '1.00794(4)'], ['He', '4.002602(2)'],
 ['Li', '6.941(2)'], ['Be', '9.012182(3)'], ...]
```

- the next example shows how to extract the elements from a list of lists
- such an operation is often called *flattening*, hence the name of the function
- we use two list comprehensions to flatten the list

```
def flatten(list_of_lists):
  return [x for l in list_of_lists for x in l]

list_of_lists = [[1,2,3],[4,5,6],[7,8]]
print('flatten({})={}'
       .format(list_of_lists,flatten(list_of_lists)))
```

```
flatten([[1, 2, 3], [4, 5, 6], [7, 8]])=[1, 2, 3, 4, 5, 6, 7, 8]
```

List comprehensions (5/16)

- one often implements a matrix by a list of lists, each of which has the same length
- in most cases one uses a row major order, i.e. each list represents a row and its length is the number of columns in the matrix
- the following function checks that the length-constraint holds for a list of lists referred to by `ll`

```
def same_list_length(ll):
  return not ll or all([len(ll[0]) == len(l) for l in ll])
```

- the function `all` returns `True` iff all elements in the list are `True`
- we use this function in an assertion in the following function, which delivers the transposed version of a matrix
- transposing an $m \times n$-matrix A delivers an $n \times m$-matrix B such that $B(j,i) = A(i,j)$ for all i, j, $0 \leq i \leq m-1$, $0 \leq j \leq n-1$

List comprehensions (6/16)

```
def transpose(matrix):
  assert same_list_length(matrix)
  return [[row[j] for row in matrix]\
          for j in range(len(matrix[0]))]
```

- the outer loop iterates over the number of columns: for some *j*th we generate a row with all values from the *j*th column, i.e. the *j*th column is turned in the *j*th row of the transposed matrix
- to test this function, we use a generic function for creating a matrix

```
def matrix_new(rows,columns,init):
  return [[init(i,j) for j in range(columns)] for i in range(rows)]
```

- the parameters of this function specify the number of rows and the number of columns plus a function `init` which delivers the value in the matrix in row *i* and column *j*; the function depends on *i* and *j*
- so we can easily create a matrix whose entries are numbered with increasing row and column index

List comprehensions (7/16)

```
matrix1 = matrix_new(2,4,lambda i,j: i*4 + j)
```

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{pmatrix}$$

– transposing this matrix gives $\begin{pmatrix} 0 & 4 \\ 1 & 5 \\ 2 & 6 \\ 3 & 7 \end{pmatrix}$ see the following:

```
matrix_t = transpose(matrix1)
print('transpose({})={}'.format(matrix1,matrix_t))
```

```
transpose([[0, 1, 2, 3], [4, 5, 6, 7]])=[[0, 4], [1, 5], [2, 6], [3, 7]]
```

List comprehensions (8/16)

- recall that pythagorean triples (i, j, k) satisfy $i^2 + j^2 = k^2$ and $1 \leq i, j, k \leq n$ for some user defined value n
- let us use a list comprehension to construct all such triples

```
def pythagorean_triples(n):
  return [(i,j,k) for i in range(1,n)\
                  for j in range(i,n)\
                  for square_sum in [i**2 + j**2]\
                  for k in [int(sqrt(square_sum))]\
                  if square_sum == k**2]

print('{}'.format(pythagorean_triples(20)))
```

```
[(3,4,5), (5,12,13), (6,8,10), (8,15,17), (9,12,15), (12,16,20)]
```

- the two loops binding `square_sum` and `k` both iterate over lists of length 1; so they basically serve as assignments
- but as assignments are not possible inside a list comprehension, we used `for`-loops

List comprehensions (9/16)

- the next list comprehension has three levels of iteration, each corresponding to a position in a codon
- the function returns a list of all 64 codons

```
def codons():
  bases = 'acgt'
  return [''.join([x,y,z]) for x in bases \
                           for y in bases \
                           for z in bases]

print('codons={}'.format(codons()))
```

```
codons=['aaa', 'aac', 'aag', 'aat', 'aca', 'acc', 'acg', 'act', ..., ]
```

- we have learned that multisets can be represented by sorted lists of non-negative integers
- so we can easily create a list of all multisets of length 3 over an alphabet $\{0, \ldots, \texttt{alpha\_size} - 1\}$ using the following list comprehension:

List comprehensions (10/16)

```
def multisets3(alpha_size):
  return [[i,j,k] for i in range(alpha_size) \
                  for j in range(i,alpha_size)\
                  for k in range(j,alpha_size)]

print('multisets3(2)={}'.format(multisets3(2)))
```

```
multisets3(2)=[[0, 0, 0], [0, 0, 1], [0, 1, 1], [1, 1, 1]]
```

- as another example reconsider the function `map` which takes two arguments:
  - a function `f` and
  - a generator `g`
- `map(f,g)` generates `f(x)` for all elements `x` generated by `g`
- as we have learned earlier, `map` could be implemented using `yield`

```
def my_map_y(f,g):
  for a in g:
    yield f(a)
```

List comprehensions (11/16)

- another implementation uses a generator comprehension, which resembles a list comprehension, except that one uses round brackets

```
def my_map(f,g):
  return (f(a) for a in g)
```

- note that a generator can save space, because it does not construct the list in memory before it is processed
- instead the elements are processed in a loop once they are generated
- we can e.g. use `map` or `my_map_y` or `my_map` to implement a generator which delivers increments of elements generated by `g`

```
def gen_increment_map(g):
  return map(lambda x: x+1, g)
```

- the use of `map` forces us to introduce a function, in this case a nameless function, using a `lambda`-expression

List comprehensions (12/16)

- using a generator comprehension, we can get rid of `map` and the `lambda`-expression

```
def gen_increment(g):
  return (x+1 for x in g)

intlist = [5,3,7,8,10]
print('gen_increment({})={}'
       .format(intlist,list(gen_increment(intlist))))
```

```
gen_increment([5, 3, 7, 8, 10])=[6, 4, 8, 9, 11]
```

- we have not previously considered the function `filter` which has two argument, a predicate `p` and a generator `g`
- it generates all elements generated by `g` satisfying `p`
- like `map` it can easily be implemented by a generator comprehension

```
def my_filter(p,g):
  return (a for a in g if p(a))
```

List comprehensions (13/16)

- as a last example on list comprehensions, consider the following function which delivers a list of primes in the range from 1 to n
- it does so by generating a list `no_primes` of integers j which are multiples of integers in the range from 2 to $\sqrt{n}$.
- the primes are the numbers in the range from 2 to n not occurring in `no_primes`

```
def sieve_lc(n):
  no_primes = [j for i in range(2, int(sqrt(n))+1)\
                 for j in range(i*2, n, i)]
  return [p for p in range(2, n+1) if p not in no_primes]
```

- as 16 is a multiple of 2 and of 4 and of 8, it appears three times in `no_primes`
- for `n=1000`, the list `no_primes` contains 2 978 elements, but only 830 different elements

List comprehensions (14/16)

- as in this context the multiplicity of elements is not relevant and leads to unnecessary computational effort, we would like to get rid of copies of the same element
- we could transform the list into a set in which each elements occurs only once
- or we could use a set comprehension which resembles a list comprehension, except that one uses curly brackets instead of square brackets

```
def sieve_sc(n):
  no_primes = {j for i in range(2, int(sqrt(n))+1)\
                 for j in range(i*2, n, i)}
  return [p for p in range(2, n+1) if p not in no_primes]
```

- this example is from https://www.python-course.eu/python3_list_comprehension.php

List comprehensions (15/16)

- besides the three kinds of comprehensions we have seen (list-, generator-, set-comprehensions) one can also create a dictionary using a dictionary comprehension
- use the curly bracket as delimiters and the colon for separating keys from values
- this is the same syntax we use when defining some key/value pairs of a dictionary

Here is an example of a function which creates a dictionary from two columns of the file `atom-data.tsv`

```
def cut_dict(stream,key_col,value_col,sep='\t'):
  return {split_line[key_col] : split_line[value_col]\
          for line in stream\
          for split_line in [line.split(sep)]}

stream = open('../Chemistry/atom-data.tsv')
print(cut_dict(stream,1,2))
```

List comprehensions (16/16)

```
{'symbol': 'name', 'H': 'Hydrogen', 'He': 'Helium', 'Li': 'Lithium', ... }
```

- there are no checks whether the column given by the first argument has unique values
- also it is not checked that the columns are valid numbers
- in a version of the method which includes such checks we would likely not use a dictionary comprehension

Extracting and visualizing data about Genbank (1/20)

- this section was inspired by the blog titled: *Web Scraping, Regular Expressions, and Data Visualization: Doing it all in Python*

  `https://towardsdatascience.com/`

  `web-scraping-regular-expressions-and-data-visualization-doing-it-all-in-python-37a1aade7924`
- this blog was about data on salaries of different university presidents in the US
- we consider a topic more related to Bioinformatics:

  the growth of the size of Genbank over time
- the data is available on the web at `https://www.ncbi.nlm.nih.gov/genbank/statistics/`
- the next frame contains a screen shot of this page from 2018-01-02.

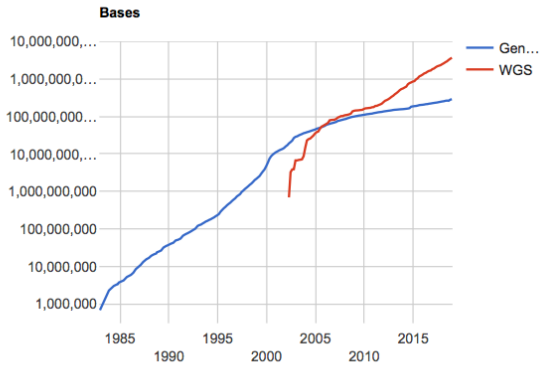
Extracting and visualizing data about Genbank (2/20)

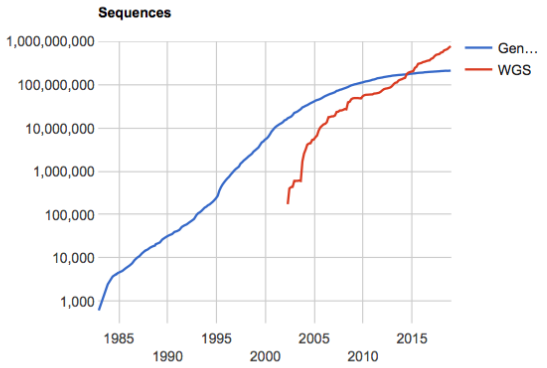
NCBI Resources How To Sign in to NCBI

GenBank Nucleotide Search

GenBank Submit Genomes WGS Metagenomes TPA TSA INSDC Other

GenBank and WGS Statistics





Notes on GenBank statistics

The following table lists the number of bases and the number of sequence records in each release of GenBank, beginning with Release 3 in 1982. CON-division records are not represented in these statistics: because they are constructed from the non-CON records in the database, their inclusion here would be a form of double-counting. From 1982 to the present, the number of bases in GenBank has doubled approximately every 18 months.

Notes on WGS statistics

ftp://ftp.ncbi.nih.gov/ncbi-asn1/wgs ftp://ftp.ncbi.nih.gov/genbank/wgs

GENBANK AND WGS STATISTICS

		GenBank		WGS	
Release	Date	Bases	Sequences	Bases	Sequences
3	Dec 1982	680338	606		
14	Nov 1983	2274029	2427		
20	May 1984	3002088	3665		
24	Sep 1984	3323270	4135		
25	Oct 1984	3368765	4175		
26	Nov 1984	3689752	4393		
32	May 1985	4211931	4954		
36	Sep 1985	5204420	5700		
40	Feb 1986	5925429	6642		

Extracting and visualizing data about Genbank (3/20)

- the web-page is updated several times a year with each release of Genbank
- our task is to automatically perform the following steps in a single Python-script:
  - download the content of the web-page (without opening a browser, of course)
  - extract the relevant data from the content (which is a string formatted in HTML)
  - plot the relevant data
- when you ping a website or portal for information this is called *making a request*
- you usually do this with a web-browser
- but you can also do it with the Python library `requests`
- this allows to make a request inside a Python script

Extracting and visualizing data about Genbank (4/20)

- in our case we want to read the content of a web-page, specified by an URL `urlstring`
- so we make a `GET`-request, followed by the decoding to obtain the content of the web-page as a single string

```
request_genbank_stat = requests.get(urlstring)
genbank_stat_html = request_genbank_stat.content
```

- during the development of the Python-Script shown here, we did not want to make requests a hundred times
- therefore, we use `wget` with the URL given above to download a file named `index.html`
- rename this file into `statistics2019-01-02.html`
- use this snapshot for development

Extracting and visualizing data about Genbank (5/20)

```
import re, argparse, requests

def gb_stat_html_get(from_web):
  if from_web:
    urlstring = 'https://www.ncbi.nlm.nih.gov/genbank/statistics/'
    request_gb_stat = requests.get(urlstring)
    gb_stat_html = request_gb_stat.content
  else:
    stream = open('statistics2019-01-02.html')
    gb_stat_html = stream.read()
    stream.close()
  return gb_stat_html
```

- as the mentioned file is a text file, we can open it with an editor
- we see a lot of data which is not useful for us
- in the middle of the file we see the specification of a table with a caption, table headers and the table data we are looking for (on the next frame we adjusted line breaking and indentation)

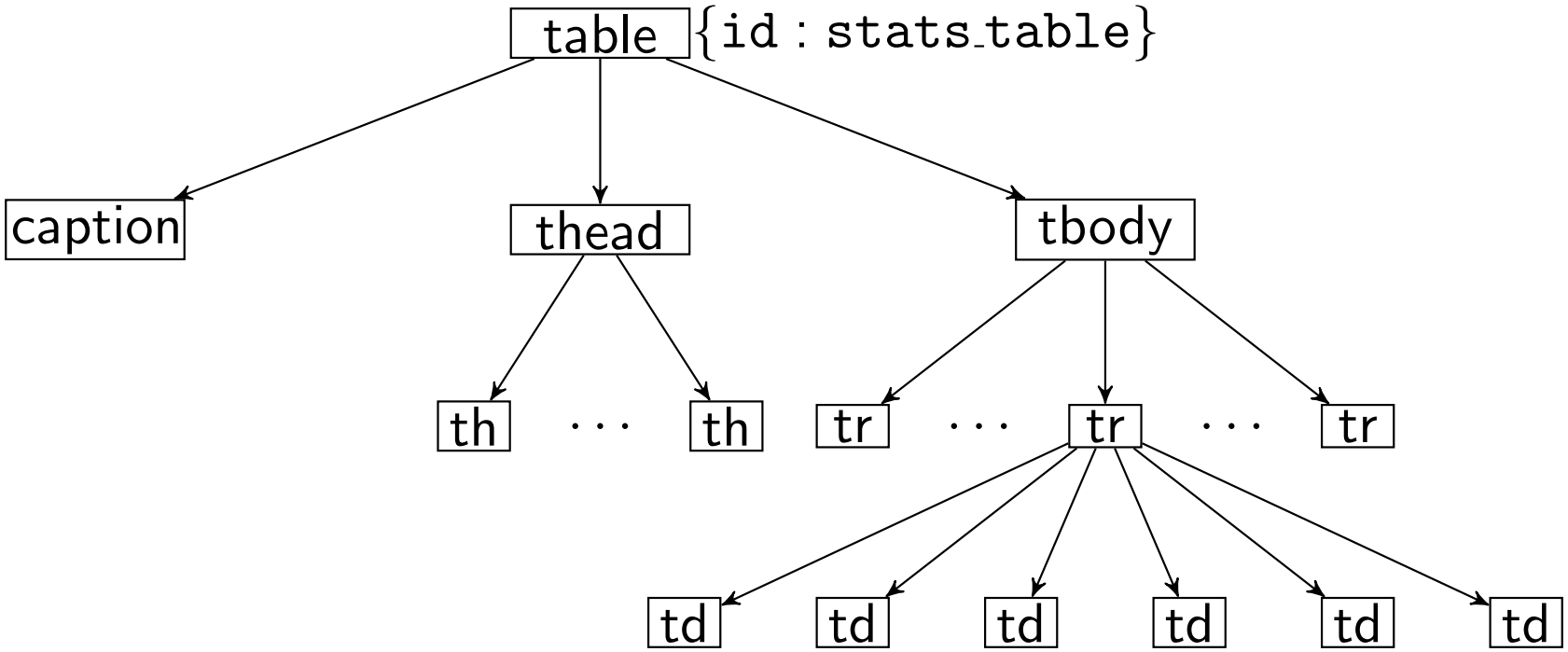
Extracting and visualizing data about Genbank (6/20)

```
<table id="stats_table" summary="GENBANK AND WGS STATISTICS">
<caption>GENBANK AND WGS STATISTICS</caption>
<thead>
    <tr><th colspan="2"></th>
        <th colspan="2">GenBank</th>
        <th colspan="2">WGS</th></tr>
    <tr><th>Release</th>
        <th>Date</th>
        <th>Bases</th>
        <th>Sequences</th>
        <th>Bases</th>
        <th>Sequences</th></tr>
</thead>
<tbody>
    <tr><td>3</td>
        <td>Dec 1982</td>
        <td>680338</td>
        <td>606</td>
        <td></td>
        <td></td></tr>
   ...
</tbody>
</table>
```

- HTML is just a special XML-dialect with predefined tags and attributes

Extracting and visualizing data about Genbank (7/20)

- the document contains statistics on Genbank and WGS, i.e. Whole Genome Sequences
- here is the corresponding tree structure:



- we could use `ElementTree`-class used before to parse the file

Extracting and visualizing data about Genbank (8/20)

- but there is a simpler way to extract the information:

  use the class `BeautifulSoup` from the module `bp4`
- this can be installed using `pip3 install bs4`

```
from bs4 import BeautifulSoup

def gb_stat_table_lines_get(gb_stat_html):
  soup = BeautifulSoup(gb_stat_html,features='html.parser')
  table = soup.find('table', attrs = {'id': 'stats_table'})
  assert table
  thead = table.find('thead') # not used
  tbody = table.find('tbody')
  h_list = ['release','date','gb_bp','gb_seqs','wgs_bp','wgs_seqs']
  gb_stat_table_lines = ['\t'.join(h_list)]
  for six_tup in tbody.find_all('tr'):
    data_fields = [td.text for td in six_tup.find_all('td')]
    assert len(data_fields) == 6
    gb_stat_table_lines.append('\t'.join(data_fields))
  return gb_stat_table_lines
```

Extracting and visualizing data about Genbank (9/20)

- the first thing to do is to create an instance of the class `BeautifulSoup` from the HTML-string using some HTML-parser.
- we use the standard `html.parser`
- we call the instance `soup` and it represents the entire HTML document as a nested structure, i.e. a tree
- the class `BeautifulSoup` provides powerful methods to extract information from a given subtree
- for development purposes we could e.g. show the document line by line with appropriate indentation to improve readability
- we use the method `find` and specify that we look for an HTML-node with tag `table` and appropriate attributes (see HTML string)
- the result of `find` is `None` if the node was not found or otherwise a `BeautifulSoup`-object (which we call `table`)
- this represents the tree structure above

Extracting and visualizing data about Genbank (10/20)

- from our analysis above we know that the table consists of a table head and a table body
- so we extract these using `find` with the tags `thead` and `tbody`, respectively
- we obtain two `BeautifulSoup`-objects named `thead` and `tbody`
- all elements of the header are enclosed in a `th`-tag and we obtain their text content by the following loop:

```
for hsoup in thead.find_all('th'):
  print(hsoup.text)
```

which gives the following output:

```
GenBank
WGS
Release
Date
Bases
Sequences
Bases
Sequences
```

Extracting and visualizing data about Genbank (11/20)

- a look at the screenshot reveals the placement of these table headers:

		Genbank		WGS	
Release	Date	Bases	Sequences	Bases	Sequences

- so in the table body we expect 6 values, a release number, a release date and the size of genbank and WGS in terms of bp. and number of sequences
- such a 6-tuple of values is contained in a `tr`-tag in the table body
- we can extract the corresponding `BeautifulSoup`-objects (named `six_tup`) using the method `find_all` with tag `tr`
- `six_tup` contains the 6 values, each delimited by `td`-tags
- we use a list comprehension and the method `find_all` to obtain the text inside each of the `td`-tags in `six_tup`
- for the early release of Genbank, WGS data was not present, which is expressed by an empty string inside a `td`-tag

Extracting and visualizing data about Genbank (12/20)

- the 6 strings extracted for each line of the table are stored as a single tab-separated string
- we use a tab separated header line with our own identifiers

  release, *date*, *gb_bp*, *gb_seqs*, *wgs_bp*, *wgs_seqs*

  for the 6 columns
- store all tab separated lines in a list `gb_stat_table_lines`
- this is the result of the function `gb_stat_table_lines_get`
- the string `gb_stat_table_lines` is suited as input for our previously developed class `DataMatrix`
- so we create a data matrix for the parsed table as follows:

```
key_col = 0 # the release number
gb_stat_data_matrix = DataMatrix(gb_stat_table_lines,key_col,
                                 sep='\t',ordered=True)
```

- we use the first column, i.e. the release number as a key

Extracting and visualizing data about Genbank (13/20)

- for plotting we need to keep the rows of the matrix in the original order
- we want to plot the size of Genbank/WGS as a function of time, so we use the release date for the X-axes
- but the release date is given as `Mmm YYYY`, like in `Dec 1982` for the first release
- we want to transform the date into a fractional number $\texttt{YYYY} + \frac{m}{12}$ where m is the month number in the range from 0 to 11
- first we implement a function to return a dictionary which allows to transform the month in `Mmm`-notation into such a number

```
def month_dict_get():
  month_list = ['Jan','Feb','Mar','Apr','May','Jun',\
                'Jul','Aug','Sep','Oct','Nov','Dec']
  month_dict = dict()
  for idx, m in enumerate(month_list):
    month_dict[m] = idx
  return month_dict
```

Extracting and visualizing data about Genbank (14/20)

- the next function takes a date string and transforms it into a floating point number as described above (e.g. `Dec 1982` ⇒ 1982.916666)
- due to this transformation the values on the X-axes appear on the correct position (between two consecutive years)

```
def date2float(month_dict,date_string):
  mo = re.search(r'(^[A-Z][a-z]{2}) (\d+)$',date_string)
  assert mo
  month = mo.group(1)
  assert month in month_dict
  return float(mo.group(2)) + month_dict[month]/12.0
```

- the next step is to extract, from the data matrix, the lists of values which are needed for the plotting
- as some WGS-data is not available, we need
  - two lists of dates (one for Genbank and one for WGS) and
  - two lists with the corresponding number of base pairs

Extracting and visualizing data about Genbank (15/20)

- to simplify the extraction, we add a method `attribute_select` to the `DataMatrix`-class
- this allows to select the values of a specific column
- the column is specified by the corresponding column-attribute

```
def attribute_select(self,attribute):
  for key in self._keys:
    yield self._matrix[key][attribute]
```

- in the following function we make heavy use of list comprehensions

Extracting and visualizing data about Genbank (16/20)

```
def prepare_plot_data(gb_stat_matrix):
  month_dict = month_dict_get()
  dates = [date2float(month_dict,date) \
           for date in gb_stat_matrix.attribute_select('date')]
  gb_bp = [int(s)\
           for s in gb_stat_matrix.attribute_select('gb_bp')]
  assert len(dates) == len(gb_bp)
  wgs_bp = [int(s)\
            for s in gb_stat_matrix.attribute_select('wgs_bp')\
            if s != '']
  wgs_dates = dates[len(dates) - len(wgs_bp):]
  assert len(wgs_dates) == len(wgs_bp)
  return dates, gb_bp, wgs_dates, wgs_bp
```

- the first list comprehension extract the date strings from the date-column of the data matrix and converts each such date string into the corresponding floating point number
- the second list comprehension extracts the strings from the gb_bp-column and converts each to an integer

Extracting and visualizing data about Genbank (17/20)

- the `wgs_bp` list is created in an analogous way, but empty strings are ignored
- the corresponding list `wgs_dates` of release dates with WGS data consists of the last `len(wgs_bp)` values of `dates` which we obtain by an appropriate slice operation
- the following function uses `matplotlib` and calls the `plot`-method or `scatter`-method twice
  - once for the genbank data plotted in blue and
  - once for the WGS data plotted in red
- as the number of base pairs has grown exponentially, we use a log-scale for the Y-axes
- the final plot is save as a pdf file with suffix `_scatter.pdf` for the scatter plot and `_plot.pdf` for the continuous plot

Extracting and visualizing data about Genbank (18/20)

```
import matplotlib.pyplot as plt
plt.switch_backend('agg') # to allow remote use

def plot_the_data(scatter, dates, gb_bp, wgs_dates, wgs_bp):
  fig, ax = plt.subplots()
  ax.grid(True)
  ax.set_title('size of Genbank/WGS from {:.0f} to {:.0f}'
           .format(floor(dates[0]),floor(dates[-1])))
  ax.set_xlabel('years')
  ax.set_ylabel('log(size) (bp)')
  ax.set_yscale('log')
  ax.set_xlim(floor(dates[0]),ceil(dates[-1]))
  if scatter:
    ax.scatter(dates, gb_bp, s=0.5, color='blue', label='genbank')
    ax.scatter(wgs_dates, wgs_bp, s=0.5, color='red', label='WGS')
  else:
    ax.plot(dates, gb_bp, color='blue',label='Genbank')
    ax.plot(wgs_dates, wgs_bp, color='red', label='WGS')
  ax.legend(loc='upper left')
  fig.savefig('genbank_{}.pdf'
              .format('scatter' if scatter else 'plot'))
```

Extracting and visualizing data about Genbank (19/20)

- the collection of functions is completed by an option parser which provides two options
- one option to choose extracting the data from an URL and one option to choose a scatter plot rather than a continuous plot

```
def parse_arguments():
  p = argparse.ArgumentParser(description=('plot statistics '
                                            'of Genbank size'))
  p.add_argument('-w','--web',action='store_true',default=False,
                 help='get information from web via URL')
  p.add_argument('-s','--scatter',action='store_true',
                 default=False,help=('show scatter plot rather '
                                     'than continuous plot'))
  return p.parse_args()
```

Extracting and visualizing data about Genbank (20/20)

- finally, we put everything together by combining the different functions in a linear order
- an overview of the different phases (including the corresponding classes and formats involved) is shown on frame 510
- the two resulting plots are shown on frame 511 and frame 512

```
args = parse_arguments()
gb_stat_html = gb_stat_html_get(args.web)
gb_stat_table_lines = gb_stat_table_lines_get(gb_stat_html)
key_col = 0 # the release number
gb_stat_matrix = DataMatrix(gb_stat_table_lines,key_col,
                            sep='\t',ordered=True)
dates, gb_bp, wgs_dates, wgs_bp = prepare_plot_data(gb_stat_matrix)
plot_the_data(args.scatter,dates, gb_bp, wgs_dates, wgs_bp)
```

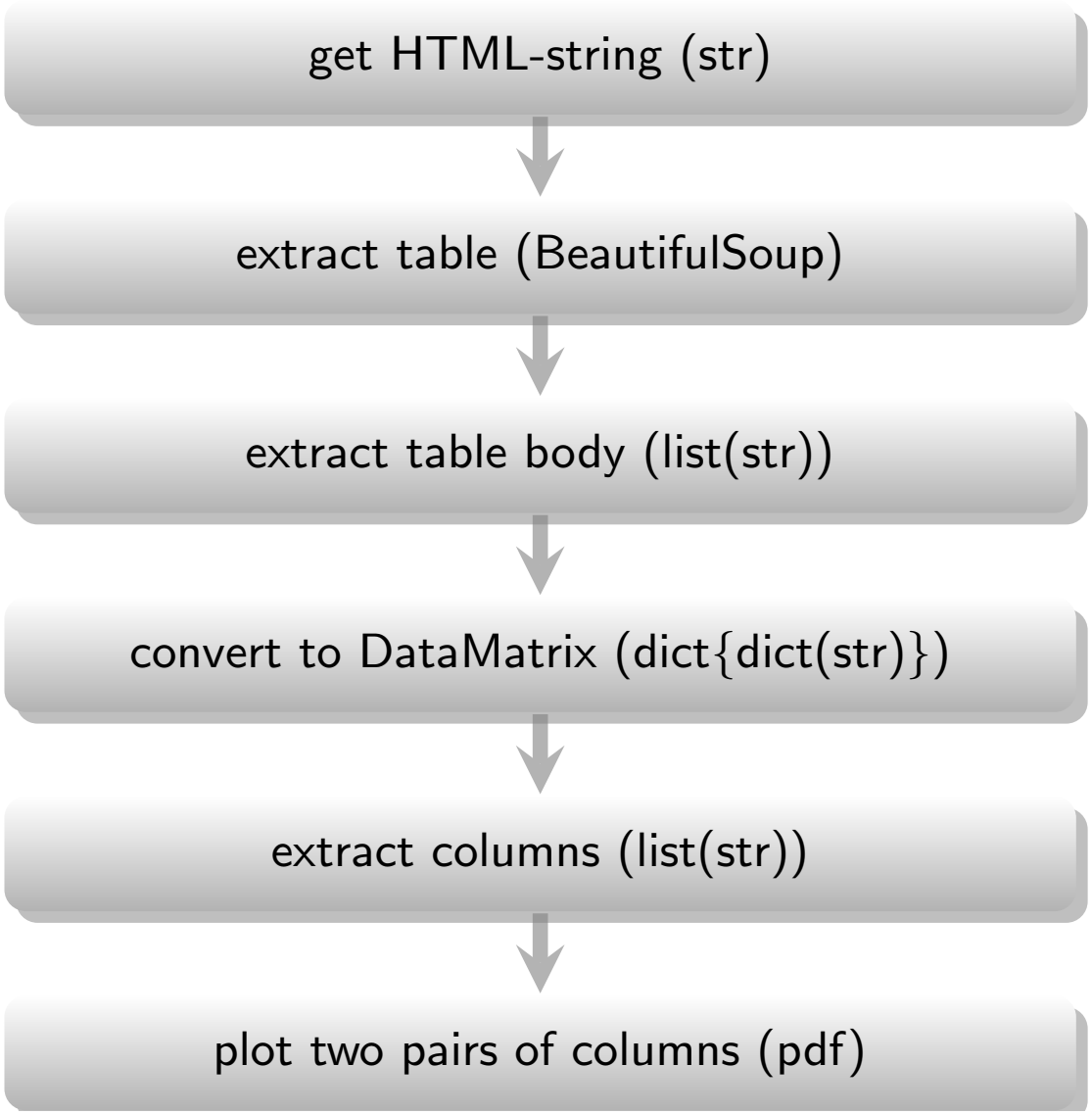


Figure: The different steps of the data extraction and plotting program.

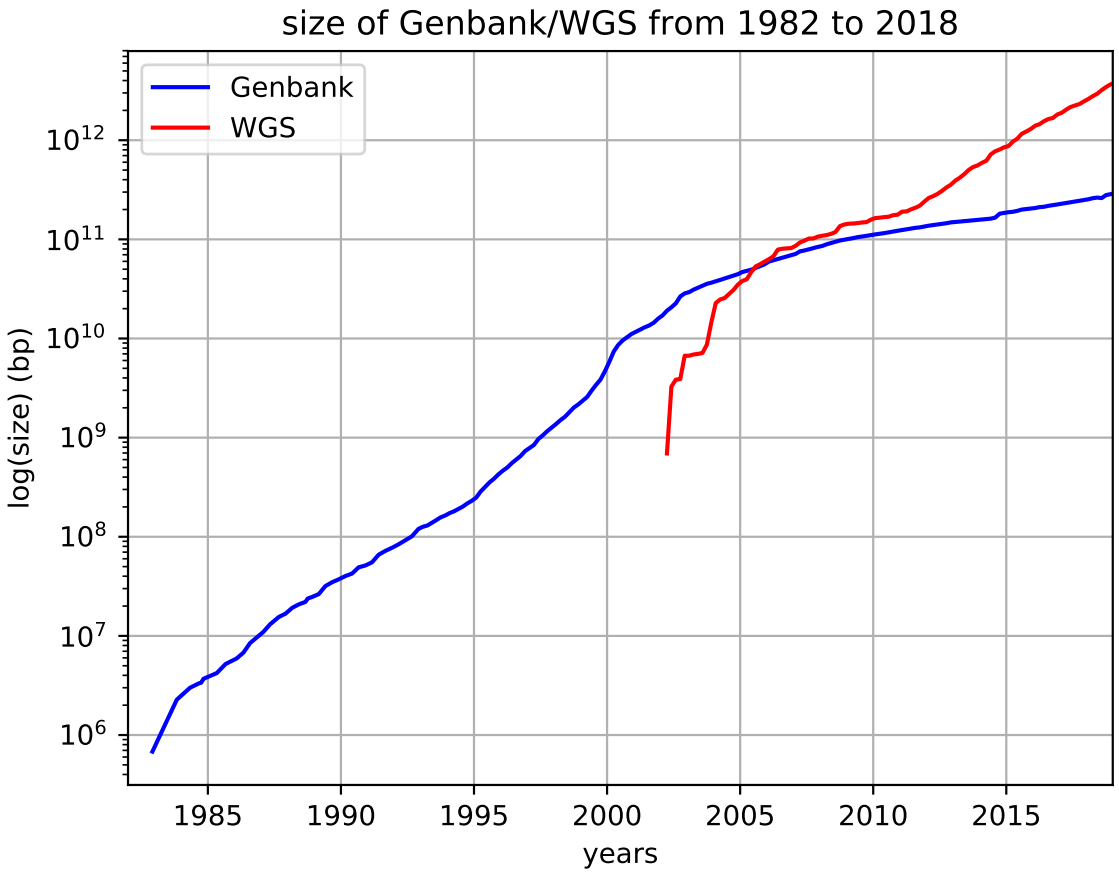


Figure: Continuous plot of Genbank/WGS sizes

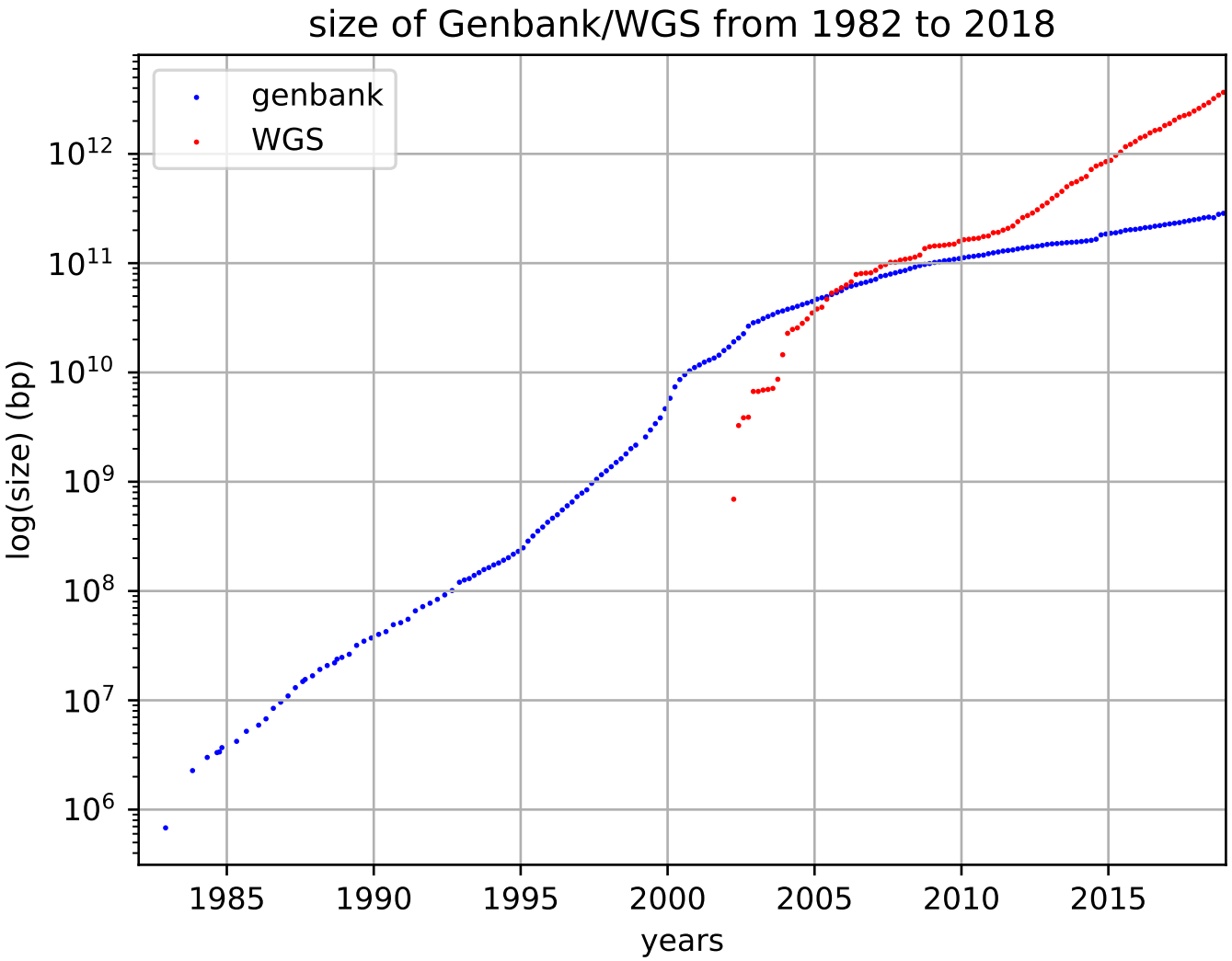


Figure: Scatter plot of Genbank/WGS sizes

Plotting time series (1/33)

- a time series is a series of data points ordered in time
- in all areas of modern societies and especially in the sciences, data often comes inf form of time series:
  - ocean tides
  - ozone concentrations
  - sunshine hours
  - sea ice coverage
  - CO_2-emissions
  - radiation of nuclear material
  - stock values
  - household income
  - sales figures
  - number of births
  - population of species
  - blood pressure of patients
  - expression level of genes
  - size of genbank

Plotting time series (2/33)

- there are many freely available time series data sets available
- some of them are studied in the context of machine learning (see `https://machinelearningmastery.com/time-series-datasets-for-machine-learning/`) where the focus is often on forecasting
- the section was inspired by the blog `https://machinelearningmastery.com/time-series-data-visualization-with-python/` which was based on temperature data from Melbourne, Australia
- our data consists of a time series with daily average temperatures of Hamburg from 2008 to 2017
- the data was downloaded from the ftp site of the *Deutsche Wetterdienst* `ftp://ftp-cdc.dwd.de/pub/CDC/observations_germany/climate/daily/kl/historical/`

Plotting time series (3/33)

- studying the meta data reveals that the weather station number 01975 stands for HH-Fuhlsbüttel, the station in Hamburg with most complete data
- ...and with a search on that page, one identifies the appropriate zip-file whose name includes this station number
- the zip-file contains the file `produkt_klima_tag_19360101_20171231_01975.txt`
- the filename encodes that data is available from 1936 to 2017
- here are the first two lines of this ;-separated 4MB file:

```
STATIONS_ID;MESS_DATUM;QN_3;  FX;  FM;QN_4; RSK;RSKF; SDK;SHK_TAG;  NM; VPM;  PM; TMK; UPM; TXK; TNK; TGK;eor
       1975;19360101;-999;-999;-999;    5;   0.1;   1;-999;   0;   7.7;   8.5;  998.00;    7.3;   82.00;    8.8
```

- studying the corresponding meta data reveals that the date is in columns 2 (shown bold) and the daily average temperature is in column 14 (shown bold)

Plotting time series (4/33)

- some linux-command including `grep` and `cut` allows to extract, select and reformat the data
- to not clutter the plots, we restrict to the data from 2008 to 2017
- we obtain a file with $3654 = 365 \cdot 10 + 3$ lines of data, where the 3 is for the three leap years 2008, 2012 and 2016.
- after including a header we obtain a file with the following head:

```
date temperature (daily mean) in Hamburg
2008-01-01 0.2
2008-01-02 0.6
2008-01-03 -2.7
2008-01-04 -4.4
...
```

- we have reformatted the dates to ease readability
- the long header for the temperature column will be inserted into the title of the plots we generate

Plotting time series (5/33)

- the class `Date` handles the data in the context of a time series
- this uses two functions:
  - a function `leapyear(year)` which returns `True` iff `year` is a leap year
  - a function `daysinmonth(is_leapyear,m)` which returns the number of days in the given month, which depends on whether the corresponding year is a leap year or not
- we do not show the implementation of these functions as they are part of an exercise
- the declaration of the class `Date` starts with the initialization of two class variables which store for each pair of month and day the number of that day in the year
- we need two variables as the calculation is different, depending on whether we have a leap year or not
- the variable `_monthday2daynum` is a dictionaries of dictionaries, such that `_monthday2daynum[m][d]` gives the day number of month `m` and day `d`

Plotting time series (6/33)

- the other variable contains the corresponding data for a leap year

```
class Date:
  _month_day2daynum = dict()
  _month_day2daynum_leap = dict()
  dyear = 1
  dyear_leap = 1
  for m in range(1,12+1):
    _month_day2daynum[m] = dict()
    _month_day2daynum_leap[m] = dict()
    for d in range(1,daysinmonth(False,m)+1):
      _month_day2daynum[m][d] = dyear
      dyear += 1
    for d in range(1,daysinmonth(True,m)+1):
      _month_day2daynum_leap[m][d] = dyear_leap
      dyear_leap += 1
```

- the `__init__`-method of class `Date` extracts the relevant substrings from a given date string
- it converts them to integers which are stored in corresponding member variables

Plotting time series (7/33)

- for the plotting we need to convert each date into a fractional number
- we already did this in the context of plotting the growth of Genbank, where we only had month and no days information

```
def __init__(self,datestring):
  mo = re.search(r'(\d{4})-(\d{2})-(\d{2})',datestring)
  if not mo:
    sys.stderr.write('{}: cannot parse datastring {}\n'
                     .format(sys.argv[0],datestring))
    exit(1)
  self._year = int(mo.group(1))
  self._month = int(mo.group(2))
  self._day = int(mo.group(3))
  if leapyear(self._year):
    divisor = 366
    daynum = Date._month_day2daynum_leap[self._month][self._day]
  else:
    divisor = 365
    daynum = Date._month_day2daynum[self._month][self._day]
  self._fraction = float(self._year) + (daynum - 1)/divisor
```

Plotting time series (8/33)

- the fractional number is obtained by adding to the year the fraction of the day number (counting from 0) and the number of days in that year
- as we need the fraction value more than once, we store it in a member variable
- as we do not want to access the four member variables, we implement corresponding accessors

```
def year(self):
  return self._year
def month(self):
  return self._month
def day(self):
  return self._day
def fraction(self):
  return self._fraction
```

- the implementation of the class is completed by methods overloading the function `str` and the operator <

Plotting time series (9/33)

- we use the latter to verify that the time series is correctly ordered (which is not actually necessary, as we could order them by ourselves)

```
def __str__(self):  # overload str
  return '{}-{}-{}'.format(self._year,self._month,self._day)
def __lt__(self,other):  # overload <
  if self._year < other._year:
    return True
  if self._year <= other._year:
    if self._month < other._month:
      return True
    if self._month == other._month:
      if self._day < other._day:
        return True
  return False
```

- the class `TimeSeries` implements several methods to handle time series
- these are read from a file with at least two columns

Plotting time series (10/33)

```
class TimeSeries:
  def __init__(self,filename,sep='\t'):
    try:
      stream = open(filename)
    except IOError as err:
      sys.stderr.write('{}: {}\n'.format(sys.argv[0],filename))
      exit(1)
    self._tsdata = list()
    self._key_name = self._value_name = previous_date = None
    for line in stream:
      values = line.strip().split(sep)
      assert len(values) >= 2
      if not self._key_name:
        self._key_name, self._value_name = values
      else:
        date = Date(values[0])
        self._tsdata.append((date,float(values[1])))
        assert not previous_date or previous_date < date
        previous_date = date
    self._firstyear = self._tsdata[0][0].year()
    self._lastyear = self._tsdata[-1][0].year()
    stream.close()
```

Plotting time series (11/33)

- the `__init__`-method extracts the attributes from the first line and stores them in two member variables `_key_name` and `_value_name`
- each element of the time series is stored as a pair of a `Date`-object and a `float`-value
- all such pairs are collected in a list `_tsdata`
- for the title of the plots we need the range of years of the data processed and so we store the first and last year in two member variables `_firstyear` and `_lastyear`
- for the methods creating the plots we need to know the number of data points and so we overload the `__len__`-method (next frame)
- moreover, we want to select data points for a specific year and thus add a method `selectby_year` implemented by a list comprehension
- the data we want to plot is used in the several other methods and so a method `data_lists` unifies the extraction

Plotting time series (12/33)

```
  def __len__(self):
    return len(self._tsdata)
  def selectby_year(self,year):
    return [(d,v) for d, v in self._tsdata if d.year() == year]
  def data_lists(self):
    x_list = [d.fraction() for d,v in self._tsdata]
    y_list = [v for d,v in self._tsdata]
    return x_list, y_list
```

- the methods for plotting time series data relies on the module `matplotlib,pyplot`
- this is imported and used as the abbreviation `plt`

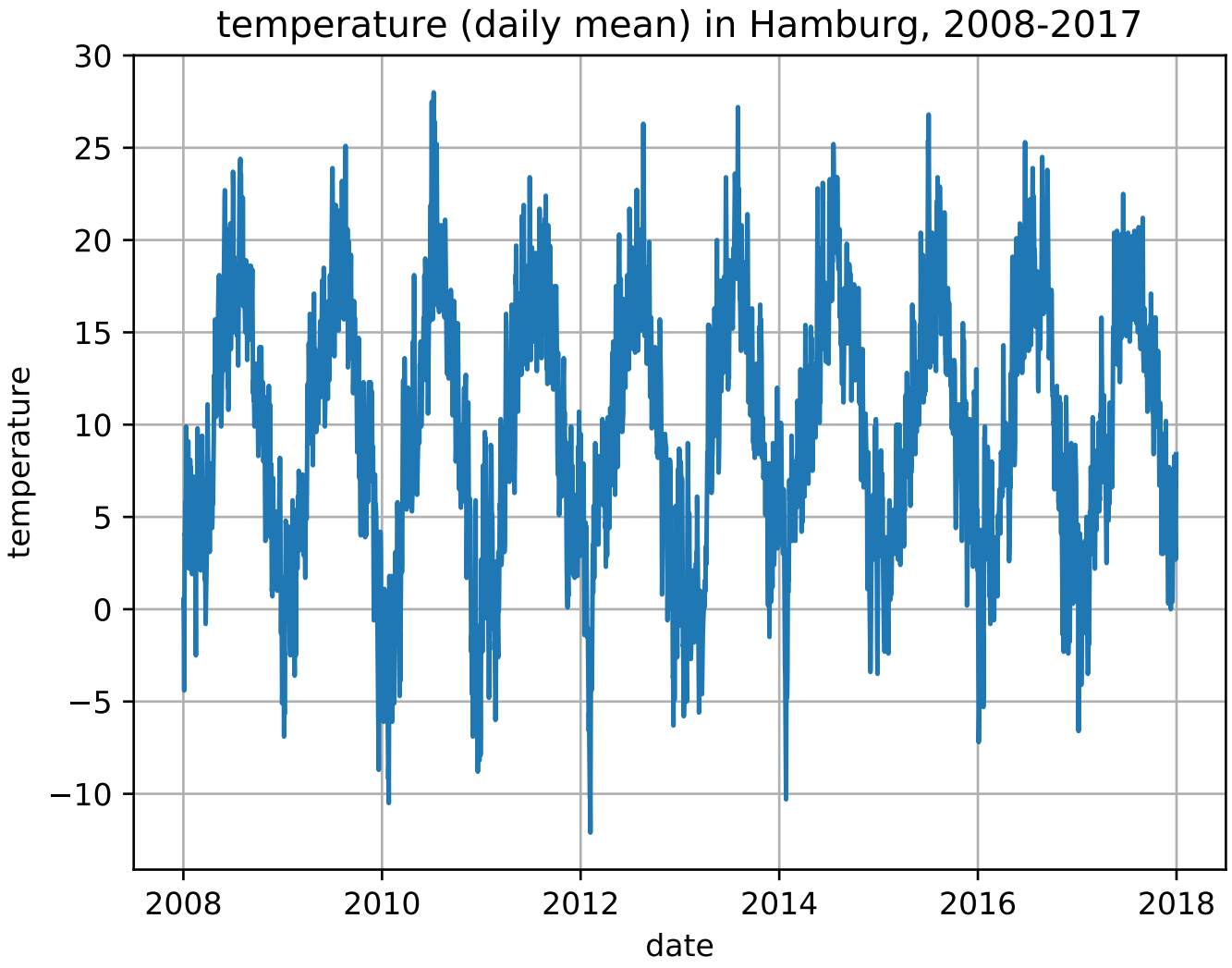
```
import matplotlib.pyplot as plt
plt.switch_backend('agg')
```

Plotting time series (13/33)

- for the first plotting method `TimeSeries.plot` we define appropriate labels for the axes and a title
- we also request a grid as a background of the plot
- we call `ax.plot` to which we provide the two lists delivered by the `data_list`-method from above
- to allow remote use of the program via ssh, we add the `switch_backend` command (you can leave it out, if you use matplotlib locally)

```
def plot(self):
  x_list, y_list = self.data_lists()
  fig, ax = plt.subplots()
  ax.set_xlabel(self._key_name)
  ax.set_ylabel(self._value_name.split()[0])
  ax.set_title('{}, {}-{}'.format(self._value_name,
                                  self._firstyear,self._lastyear)
                                      )
  ax.grid(True)
  ax.plot(x_list,y_list)
  fig.savefig('temp_plot.pdf')  # save figure as pdf file
```

Plotting time series (14/33)

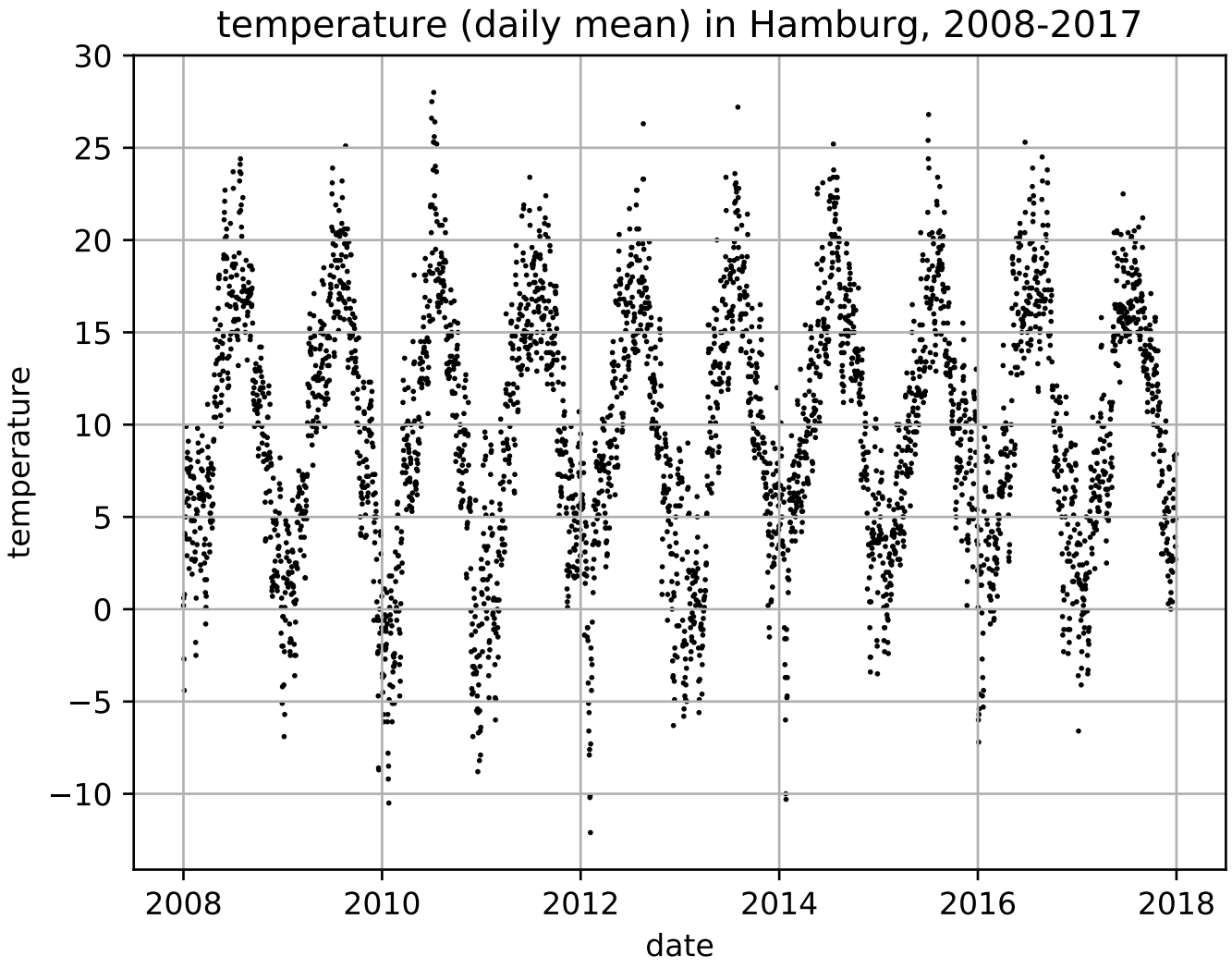


Plotting time series (15/33)

- as the temperatures are discrete data, it would be more appropriate to create a scatter plot
- this is obtained by `TimeSeries.scatter` which employs `ax.scatter`
- the rest of the function is identical to `TimeSeries.plot`

```
def scatter(self):
  x_list, y_list = self.data_lists()
  fig, ax = plt.subplots()
  ax.set_xlabel(self._key_name)
  ax.set_ylabel(self._value_name.split()[0])
  ax.set_title('{}, {}-{}'.format(self._value_name,
                                  self._firstyear,self._lastyear)
                                      )
  ax.grid(True)
  ax.scatter(x_list,y_list,s=0.5,color='black')
  fig.savefig('temp_scatter.pdf')  # save figure as pdf file
```

Plotting time series (16/33)



Plotting time series (17/33)

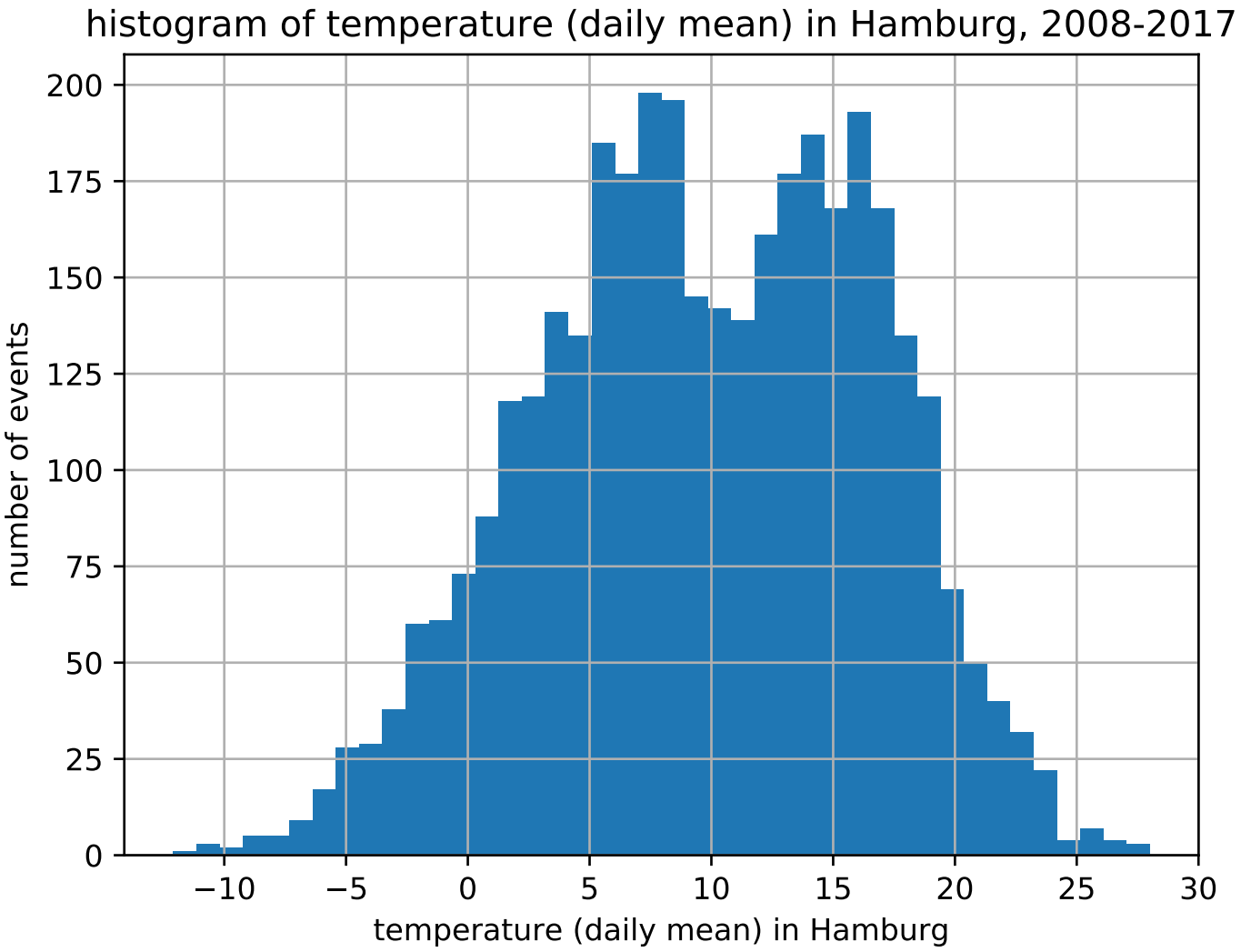
- one is often interested in the distribution of values in a time frame
- such a distribution is usually plotted as a histogram
- there is an appropriate method `ax.hist` to draw histograms
- this has to be provided with the list of values from which the distribution is to be computed
- the granularity of the histogram is defined by the number of bins, specified by the `bins`-parameter
- in our case we use one bin for each rounded temperature value (i.e. -12, -11, . . . , 26, 27 for the chosen time interval)
- in general the number of bins is determined from the ceiled/floored minimum and maximum temperature
- the temperature bins are shown on the X-axes
- the number of data points falling into the corresponding bins are shown on the Y-axes

Plotting time series (18/33)

```
def histogram(self):
  def temp_bound(t):
    return math.ceil(t) if t > 0.0 else math.floor(t)
  _, value_list = self.data_lists()
  min_value = temp_bound(min(value_list))
  max_value = temp_bound(max(value_list))
  fig, ax = plt.subplots()
  ax.set_xlabel(self._value_name)
  ax.set_ylabel('number of events')
  ax.set_title('histogram of {}, {}-{}'
               .format(self._value_name,self._firstyear,
                                        self._lastyear))
  ax.grid(True)
  ax.hist(value_list,bins=max_value - min_value + 1)
  fig.savefig('temp_hist.pdf')  # save figure as pdf file
```

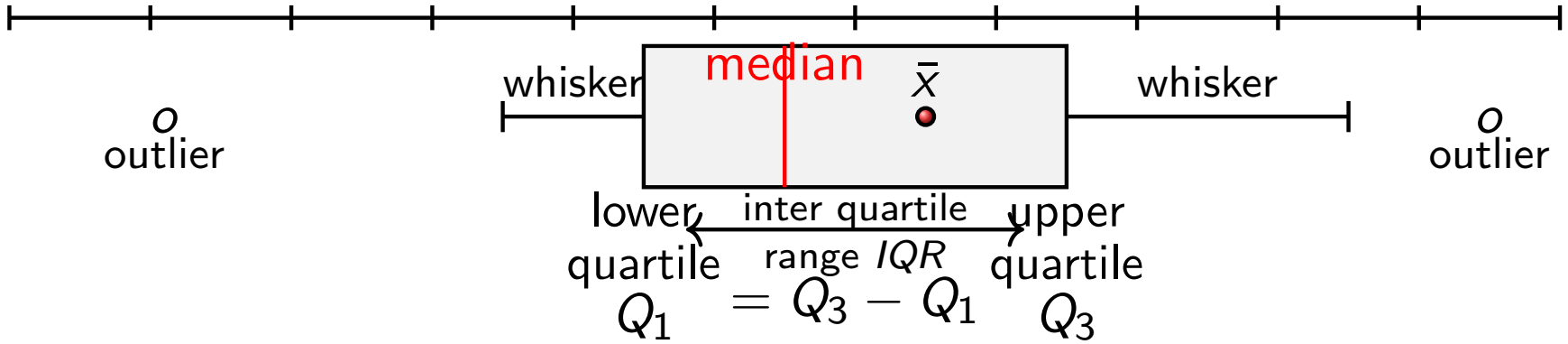
- the histogram is shown on the next frame
- note the unexpected bimodal distribution of the temperatures, due to the decrease of the number of events in the range from 8-12°C
- for an explanation we probably would have to ask a meteorologist

Plotting time series (19/33)



Plotting time series (20/33)

- another means of displaying a distribution is to use a box and whisker plot (boxplot, for short)
- a boxplot shows the shape of the distribution, its central value, and its variability including outliers
- here is a schematic representation of a horizontally displayed boxplot



Q_1 = median of values smaller than median
Q_3 = median of values larger than median
left whisker: $< Q_1$; right whisker: $> Q_3$, both except outliers
outlier: $< Q_1 - (1.5 \cdot IQR)$ or $> Q_3 + (1.5 \cdot IQR)$

adapted from http://www.texample.net/tikz/examples/box-and-whisker-plot/

Plotting time series (21/33)

- in a similar way as before, we call `ax.boxplot` in our method `TimeSeries.boxplot`
- it would not be too interesting to create a single boxplot for all data points
- instead we group the data by year
- for this step we use a function `groupby` implemented outside of the class
- this is applied to a list of key/value-pairs
- the parameter `select_func` allows to select an arbitrary value from the key
- the selected value serves as a key for the ordered dictionary created

Plotting time series (22/33)

```
from collections import OrderedDict

def groupby(data_list,select_func):
  groups = OrderedDict()
  for key, value in data_list:
    y = select_func(key)
    if not y in groups:
      groups[y] = list()
    groups[y].append(value)
  return groups
```

- inside the class `TimeSeries` we implement the specialization `groupby_year` to group the data points by their year

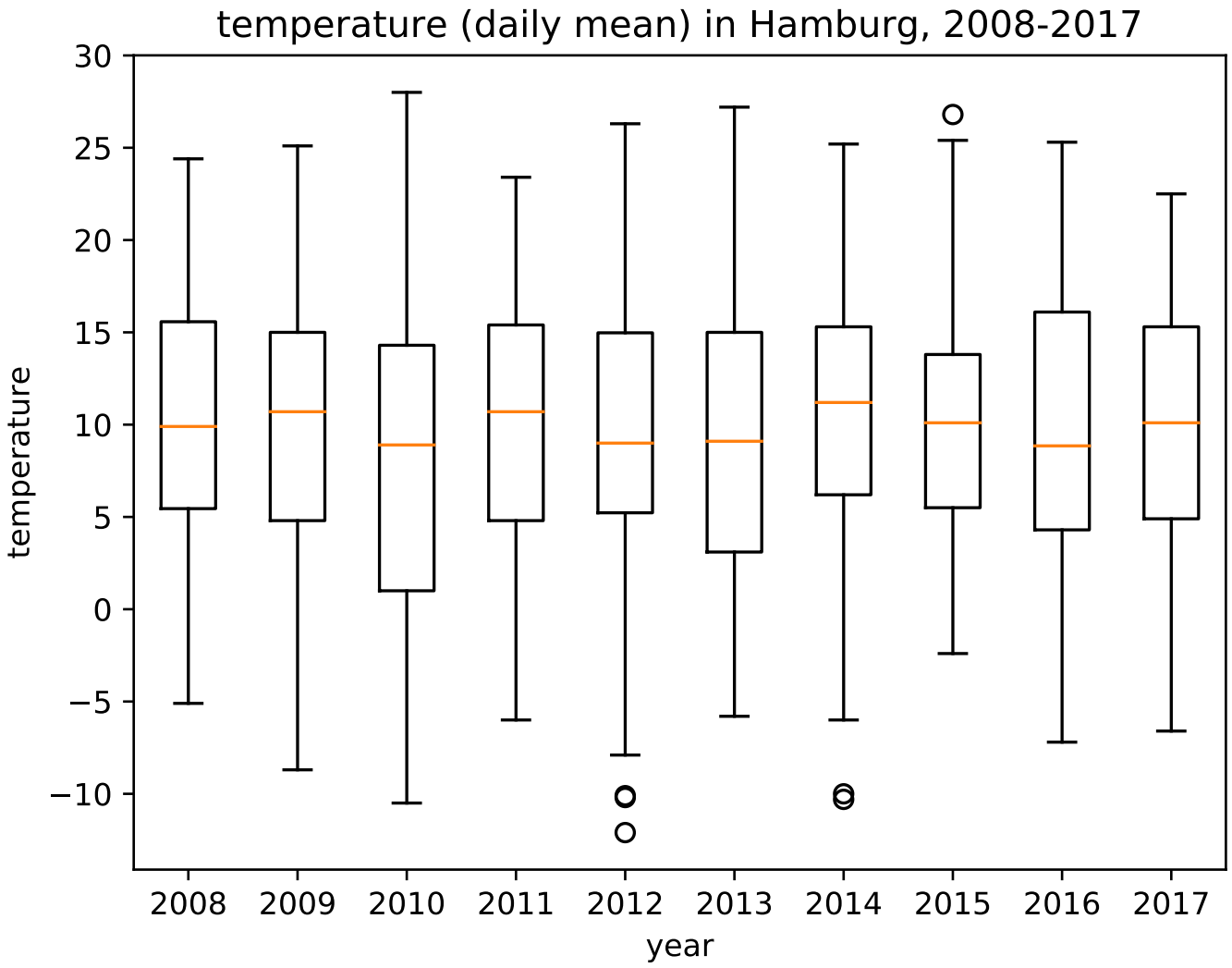
```
def groupby_year(self):
  return groupby(self._tsdata,lambda d: d.year())
```

Plotting time series (23/33)

- the implementation of `TimeSeries.boxplot` is simple
- we only need to provide `ax.boxplot` with the list of lists of values, i.e. one list for each year
- these are obtained by calling `groups.values()`
- we also define the labels using the parameter `labels`
- these are the keys of the dictionary `groups`

```
def boxplot(self):
  groups = self.groupby_year()
  fig, ax = plt.subplots()
  ax.set_title('{}, {}-{}'
               .format(self._value_name,min(groups.keys()),
                                        max(groups.keys())))
  ax.set_xlabel('year')
  ax.set_ylabel(self._value_name.split()[0])
  ax.boxplot(groups.values(),labels=groups.keys())
  fig.savefig('temp_boxplot.pdf')  # save figure as pdf file
```

Plotting time series (24/33)

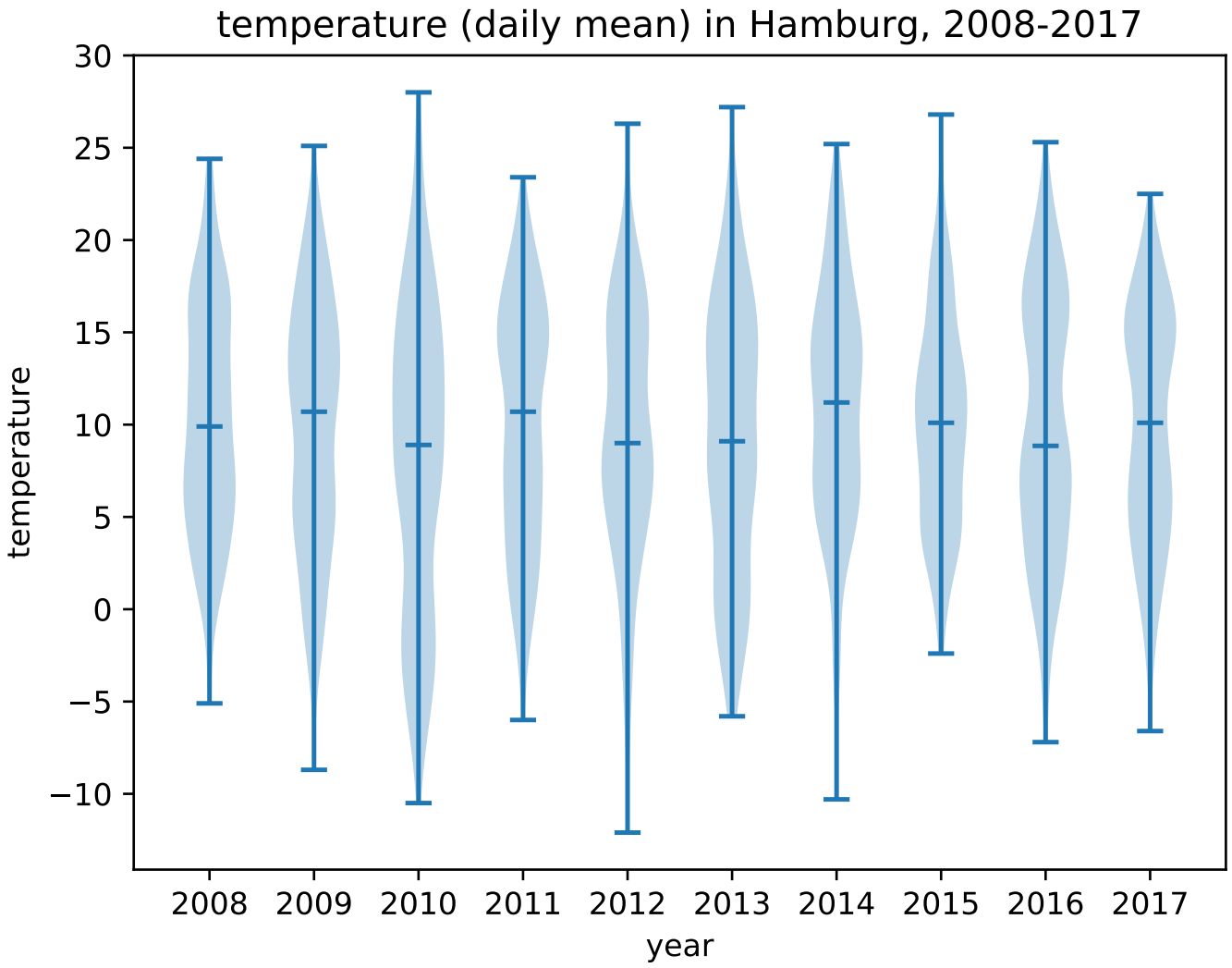


Plotting time series (25/33)

- a similar and often better way is to use a violin plot instead of a boxplot, as the former adds a second dimension represented by the width of the violin at some y-coordinate

```
def violinplot(self):
  groups = self.groupby_year()
  fig, ax = plt.subplots()
  ax.set_title('{}, {}-{}'
               .format(self._value_name,min(groups.keys()),
                                        max(groups.keys())))
  ax.set_xlabel('year')
  ax.set_ylabel(self._value_name.split()[0])
  ax.violinplot(groups.values(),showmedians=True)
  keys = list(groups.keys())
  ax.set_xticks(list(range(1,len(keys)+1)))
  ax.set_xticklabels(keys)
  fig.savefig('temp_violinplot.pdf')  # save figure as pdf file
```

Plotting time series (26/33)

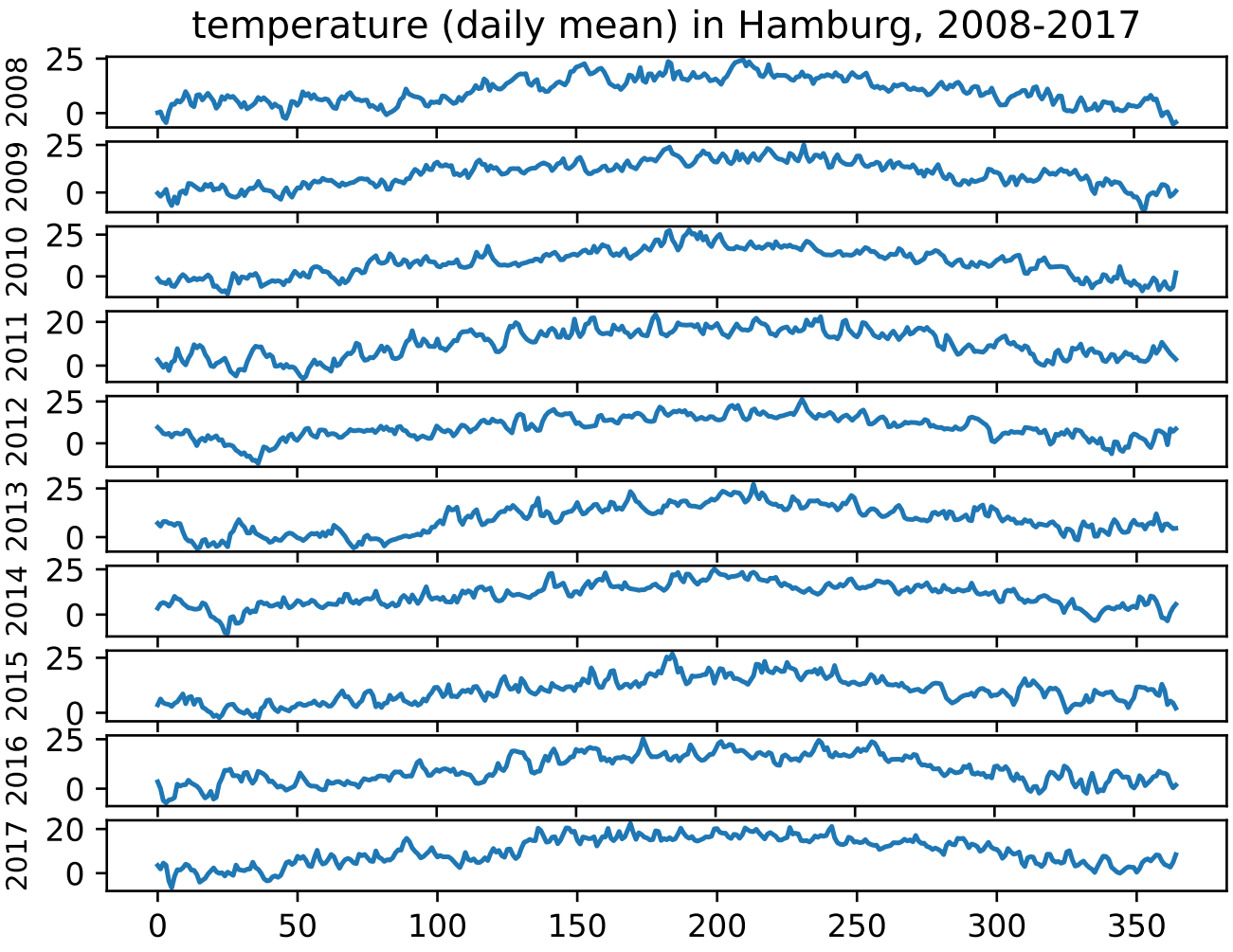


Plotting time series (27/33)

- we now want to plot the temperatures for each year and each day in that year in a stacked form
- so we call `plt.subplots` such that we get an array `ax` of `rows = len(groups)` axes-objects
- we set the title for `ax[0]` and the ticks for the `ax[rows-1]`
- for each `year` indexed at index `idx` we plot the values for each day in `ax[idx]` and set the corresponding label

```
def subplot(self):
  groups = self.groupby_year()
  rows = len(groups)
  fig, ax = plt.subplots(rows,1)
  ax[0].set_title('{}, {}-{}'.format(self._value_name,
                                     min(groups.keys()),
                                     max(groups.keys())))
  ax[rows-1].set_xticks([1] + list(range(25,365,25)))
  for idx, year in enumerate(groups.keys()):
    ax[idx].set_ylabel('{}'.format(year),fontsize=9)
    ax[idx].plot(groups[year])
  fig.savefig('temp_subplot.pdf')  # save figure as pdf file
```

Plotting time series (28/33)

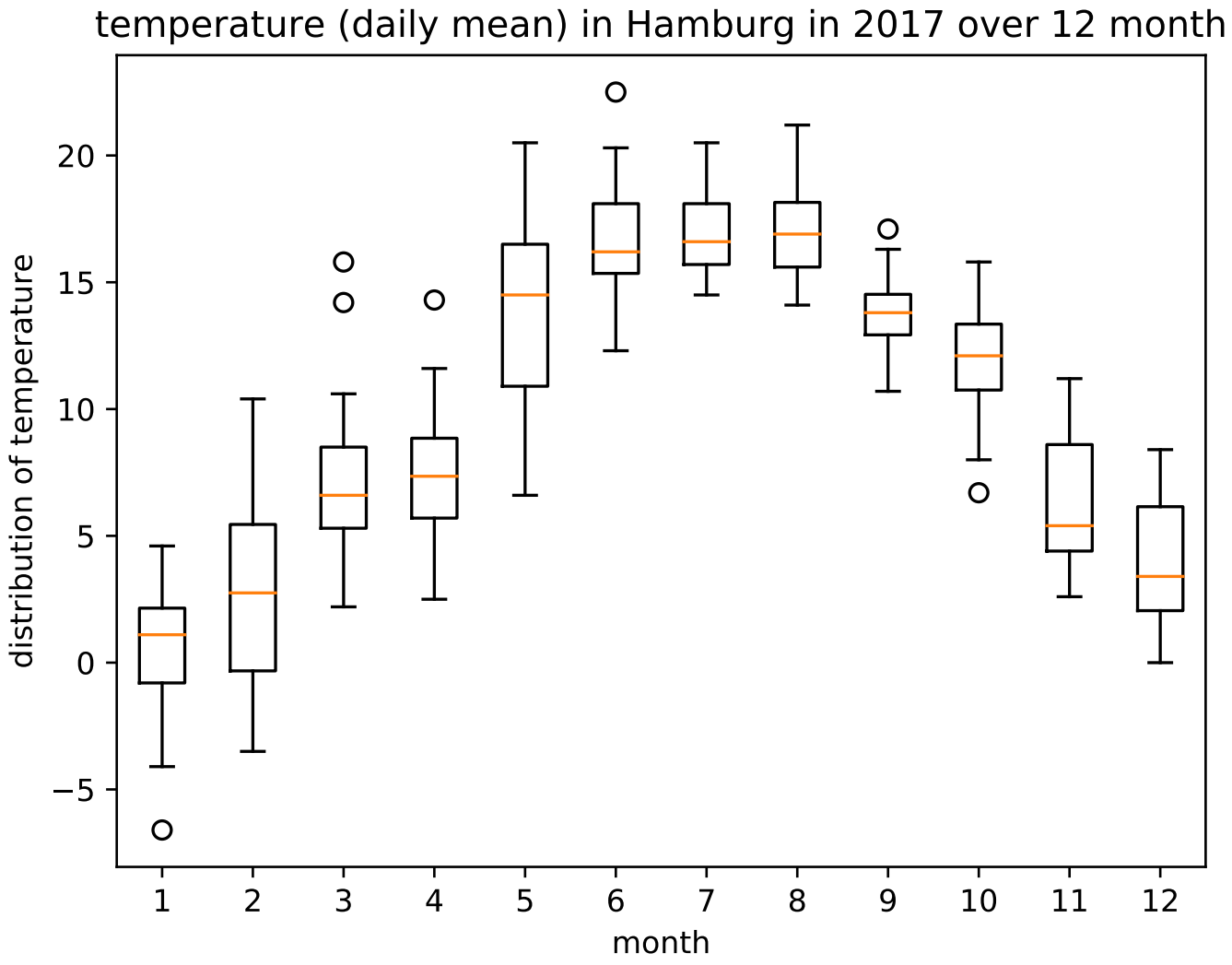


Plotting time series (29/33)

- the final plot is based on the temperature values of a single year
- for each month of the chosen year we want a boxplot of the temperatures of that month
- for this we first select all values for a specified year, using `selectby_year` declared above
- the resulting data, i.e. a list of key/value pair is then grouped according to the month using the function `groupby`
- the resulting dictionary is then fed into `ax.boxplot` as done before

```
def boxplot_monthly(self,year):
  date_one_year = self.selectby_year(year)
  groups = groupby(date_one_year,lambda d: d.month())
  fig, ax = plt.subplots()
  ax.set_title('{} in {} over 12 month'
               .format(self._value_name,year))
  ax.set_xlabel('month')
  ax.set_ylabel('temperature (C)')
  ax.boxplot(groups.values(),labels=groups.keys())
  fig.savefig('temp_monthly_{}.pdf'.format(year))  # save as pdf
```

Plotting time series (30/33)



Plotting time series (31/33)

- the remaining code of the time series plotting program first calls an option parser
- the four `with_`-options (see next frame) are set to `False` to tell the option parser to exclude options that are only relevant for another more powerful plotting program
- we do not show the option parser here, but only the corresponding help text:

```
usage: time_series.py [-h]
                      (--std | --scatter | --hist | --boxp | --ysub | --year YEAR)

plot time series

optional arguments:
  -h, --help   show this help message and exit
  --std        standard plot
  --scatter    show scatter plot
  --hist       show histograms black dots for plot
  --boxp       show boxplots for each year
  --ysub       show subplots for each year
  --year YEAR  show boxplots for month of the given year
```

Plotting time series (32/33)

- the next step creates a `TimeSeries`-instance from a given file with the data shown on frame 516
- the number of data points is reported
- depending on the options set, the corresponding `TimeSeries`-method is called

```
from temp_args import temp_parse_arguments
args = temp_parse_arguments(\
            with_hsmth=False,with_heat=False,
            with_lag=False,with_acor=False)
tseries = TimeSeries('temperature.tsv')
print('{} data points in time series'
       .format(len(tseries)))
if args.std:
  tseries.plot()
elif args.scatter:
  tseries.scatter()
elif args.hist:
  tseries.histogram()
elif args.boxp:
  tseries.boxplot()
elif args.violinp:
  tseries.violinplot()
elif args.ysub:
  tseries.subplot()
elif args.year:
  tseries.boxplot_monthly(args.year)
else:
  assert False
```

Plotting time series (33/33)

Final remark

- the plots above can also be created using the Python-module `pandas`
- this provides a class `Series`
- this is much more general than our `TimeSeries`-class
- it provides predefined methods like `groupby` or `selectby`
- and other kinds of plots like lag-plots and auto-correlation plots
- the blog cited above is based on `pandas.Series`
- but some non-trivial modifications of the code presented there were necessary since `pandas` has undergone many changes since the blog was posted in early 2017

Interactive Plots (1/14)

- suppose we have a function g with $n \geq 2$ real valued parameters $x_1, x_2, \ldots, x_{n-1}, x_n$ and a real valued return value $y = g(x_1, x_2, \ldots, x_{n-1}, x_n)$
- suppose we need to plot this function in two dimensions
- this requires that we can vary only one parameter of g
- so we assign concrete values to $n-1$ parameters, say, v_i to x_i for $1 \leq i \leq n-1$ and plot the function

$$f(x) = g(v_1, v_2, \ldots, v_{n-1}, x)$$

for varying x.

Interactive Plots (2/14)

Example

The function $g(a, b, x) = ax + b$ is the linear function with slope a and shift b. Omitting the constants on the left hand side, we get a function $f(x) = ax + b$ for constants a and b.

Goal

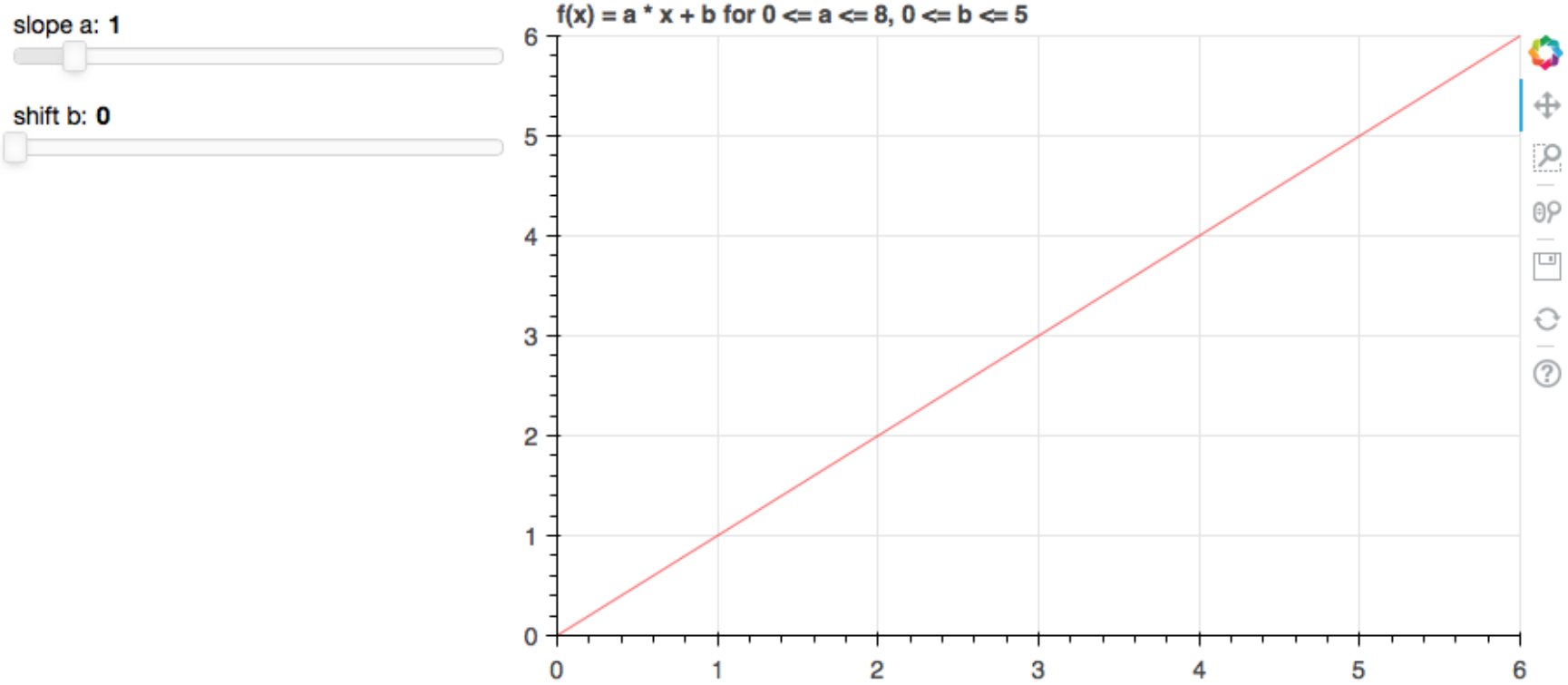
- create a two dimensional plot of a function f depending on one parameter and $n - 1$ constants
- interactively modify the values of the constants by corresponding sliders such that the plot is simultaneously modified accordingly

Example

For the linear function $f(x) = ax + b$ we would need two sliders, one for a and one for b and changing their value would modify the plot of f.

Interactive Plots (3/14)

- `https://bokeh.pydata.org/en/latest/docs/gallery/slider.html` provides an example how to implement such an interactive plot in *Bokeh* (see next frame)
- here is a static screenshot, but the lecture will of course show how it works interactively



Interactive Plots (4/14)

Bokeh* ...

- is an interactive visualization library that targets modern web browsers for presentation.
- provides methods for elegant and concise construction of versatile graphics,
- extends this capability with high-performance interactivity over very large or streaming datasets,
- allows to quickly create interactive plots, dashboards, and data applications,
- can be used from Python (or Scala, or R, or...)
- does not require knowledge of JavaScript

*: `https://bokeh.pydata.org/en/latest/`

Interactive Plots (5/14)

- the original Bokeh-code referenced above is a single module for plotting a concrete function within fixed range of values, all of which are specified as global objects
- we want to abstract from this and implement an own class `InteractivePlot`
- we first need to import several Bokeh-methods and classes
- some methods are renamed to more easily identify them as Bokeh-methods

```
import numpy as np
from bokeh.plotting import figure as bokeh_plot
from bokeh.layouts import row as bokeh_row
from bokeh.layouts import column as bokeh_column
from bokeh.models import ColumnDataSource
from bokeh.models.widgets import Slider
from bokeh.io import curdoc
```

- the following function is implemented outside of class `InteractivePlot`

Interactive Plots (6/14)

- it takes a function `func` as arguments
- the constants (to be varied interactively) are provided as a list `args`, used as first argument
- `func` is applied to all values in the numpy array `np_x_vec` and the result is a new `numpy`-vector `np_y_vec` of the corresponding function values
- from the two numpy-vectors a dictionary is created which maps the keys `'x_vec'` and `'y_vec'` to the two numpy arrays
- the dictionary is stored in the `data`-field of the object `cd_source` which is an instance of the Bokeh-class `ColumnDataSource`
- the data for a Bokeh graph is usually provided via a `ColumnDataSource`-object and the above initialization with a dictionary is one way of storing the data in such an object

```
def update_cd_source(cd_source,func,args,np_x_vec):
  np_y_vec = np.array([func(args,x) for x in np_x_vec])
  cd_source.data = {'x_vec' : np_x_vec, 'y_vec' : np_y_vec}
```

Interactive Plots (7/14)

- the class provides only the `__init__`-Function and has no instance variables
- the `__init__`-Function has (besides `self`) the following parameters:
  - `plt_title`: title of plot
  - `x_min`, `x_max`: minimum/maximum value on the X-axis
  - `color`: the color of the plotted function
  - `func`: the function itself
  - `sliders_spec`: specification of the sliders as a list of 5-tuples of the description of the constant (e.g. slope), the name (e.g. a), the minimum and maximum of the range of values controlled by the slider and its initial value
- from the specification we create three lists:
  - `sliders` contains Bokeh `Slider`-objects initialized appropriately
  - `init_args` holds the initial values of each constant
  - `consts` stores strings with the name of the constant and its range, to be used for the title of the plot

Interactive Plots (8/14)

```
class InteractivePlot:
  def __init__(self,plt_title,x_min,x_max,color,func,sliders_spec):
    sliders = list()
    init_args = list()
    consts = list()
    for s_desc, s_name, s_min, s_max, s_init in sliders_spec:
      sliders.append(Slider(title='{}{}'.format(s_desc,s_name),
                            value=s_init,start=s_min,end=s_max,
                            step=(s_max-s_min)/100))
      init_args.append(s_init)
      consts.append('{} <= {} <= {}'.format(s_min,s_name,s_max))
    num_points = 200
    np_x_vec = np.linspace(x_min,x_max,num_points)
    np_y_vec = np.array([func(init_args,x) for x in np_x_vec])
    y_min, y_max = min(np_y_vec), max(np_y_vec)
    plot = bokeh_plot(plot_width=600,
                      plot_height=400,
                      title='{} for {}'
                             .format(plt_title,', '.join(consts)),
                      x_range=[x_min,x_max],
                      y_range=[y_min,y_max])
```

Interactive Plots (9/14)

- after filling the three lists, one creates a `numpy`-vector of 200 entries stretched on the range between `x_min` and `x_max`
- the range of values for the Y-axis is determined by the minimum/maximum function-value with initial constant values and for all values in `np_x_vec`
- an even better way would be to also vary the constants to determine the minimum and maximum possible Y-values, but this would lead to a combinatorial explosion and possibly long runtimes
- the next step is to generate the plot (not yet including the line plot of the function) using the method `bokeh_plot`
- this part of the plot is fixed, i.e. not affected by the sliders

Interactive Plots (10/14)

- the line-plot of the function is created by the method `line` for which we specify what to show on the X-axis and the Y-axis using assignments of keys to the parameters `x` and `y`
- the keys `'x_vec'` and `'y_vec'` refer to the instance `cd_source` of class `ColumnDataSource`
- `cd_source` is initialized by the function explained above and assigned to the parameter `source`
- so the value for key `'x_vec'` in `cd_source` (i.e. `np_x_vec`) is shown on the X-axis and
- the value for key `'y_vec'` in `cd_source` (i.e. `np_y_vec` in `update_cd_source`) is shown on the Y-axis

```
cd_source = ColumnDataSource()
update_cd_source(cd_source,func,init_args,np_x_vec)
plot.line(x='x_vec', y='y_vec', source=cd_source, line_width=1,
          line_color=color,line_alpha=0.6)
```

Interactive Plots (11/14)

- for each slider we need to specify what to do when the value controlled by it changes
- so we iterate over the sliders and apply the Bokeh-method `on_change` to the slider
- whenever, the value controlled by a slider changes the supplied function `update_data` is called
- this function has three unused parameters
- it first collects the current value of each sliders in a list and calls `update_cd_source` with this list of values
- as the function is local to the function `__init__` it can access the local variables `cd_source` and `np_x_vec` as well as the parameter `func`

```
def update_data(attr,new,old):
  slider_values = [slider.value for slider in sliders]
  update_cd_source(cd_source,func,slider_values,np_x_vec)
for slider in sliders:
  slider.on_change('value',update_data)
```

Interactive Plots (12/14)

- the final step of the method `__init__` in class `InteractivePlot` concerns the layout of the sliders within the current document
- this can be specified using the methods `bokeh_column` (to arrange elements on top of each other) and `bokeh_row` (to arrange elements beside each other)
- in our case we arrange the sliders (provided as a list) in the left row and the plot to the right of this column
- this column- and row-wise layout is added as a root to the current document (where the latter is obtained by the method `curr_doc`)

```
first_column = bokeh_column(sliders)
curdoc().add_root(bokeh_row(first_column, plot, width=800))
```

Interactive Plots (13/14)

- the implementation in form of a class simplifies reusing it
- the first application is to create an interactive plot of linear functions:

```
from interactive_plot import InteractivePlot

def linear_function(args,x):
  return args[1] + args[0] * x

sliders_spec = [('slope ','a',0,8,1),('shift ','b',0,5,0)]
title = 'f(x) = a * x + b'
InteractivePlot(title,0,6,'red',linear_function,sliders_spec)
```

- recall that we need two sliders, one for the slope a and one for the shift b
- so we have a list of two 5-tuples with appropriate values for the range of the slider and its initial value
- suppose this code is stored in a file `linear_func.py`

Interactive Plots (14/14)

- we now want to run a Bokeh-web server hosting the application we have implemented
- this is achieved by executing
  `bokeh serve linear_func.py`
  in the Linux shell
- once the server has started, we paste the following URL in the web-browser so that it communicates with the server:
  `http://localhost:5006/linear_func`
- this then creates the display in the browser's window and interactively changing the slider value using the mouse leads to a modified plot
- here is another example for quadratic functions:

```
def quadratic_function(args,x):
  return args[0] * x * x + args[1] * x + args[2]

sliders_spec = [('','a',0,4,1),('','b',0,16,1),('','c',0,64,1)]
title = 'f(x) = a * x^{2} + b * x + c'
InteractivePlot(title,0,6,'blue',quadratic_function,sliders_spec)
```

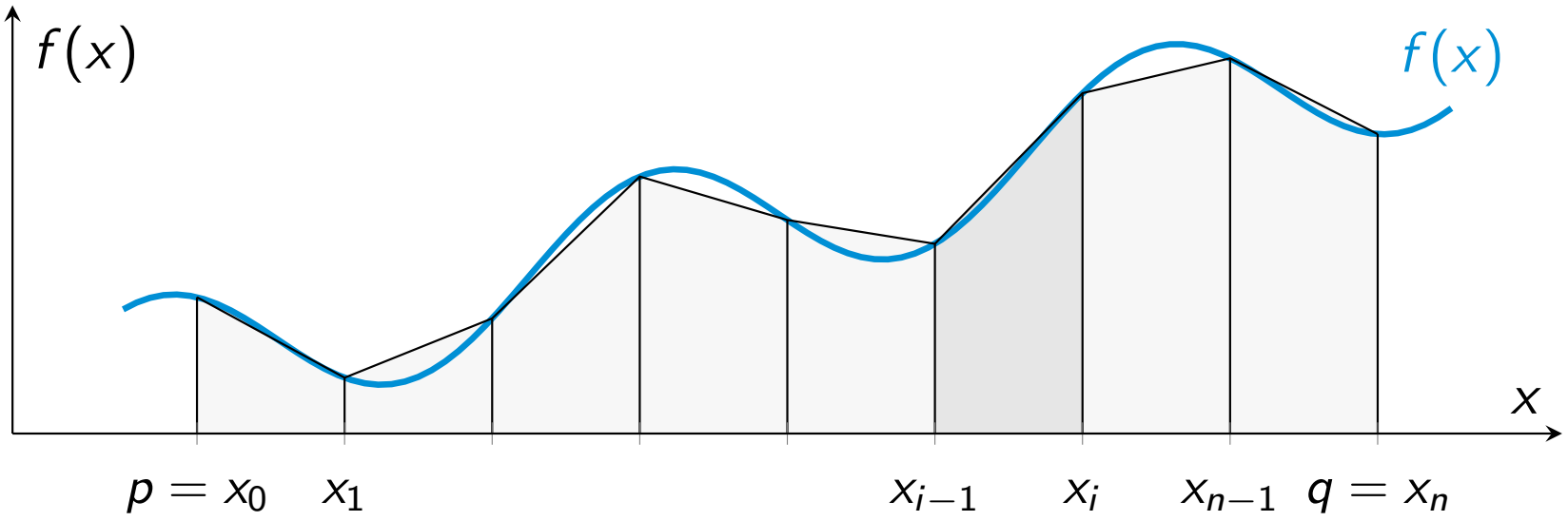
A Numpy Application: Numerical integration with trapezoids (1/1)

- we now want to determine the integral of a function f i.e. compute

$$\int_p^q f(x)dx$$

  in the range from p to q
- for some functions one can apply standard mathematical techniques to solve the problem
- but in many cases these techniques can be very tedious and so it is easier to use a numerical method, executed by a program
- the idea is to approximate the integral of f by many trapezoids for which we can determine an integral and to sum these up
- the trapezoidal regions are placed at even distance, say d, on the X-axes

Numerical integration with trapezoids (1/12)

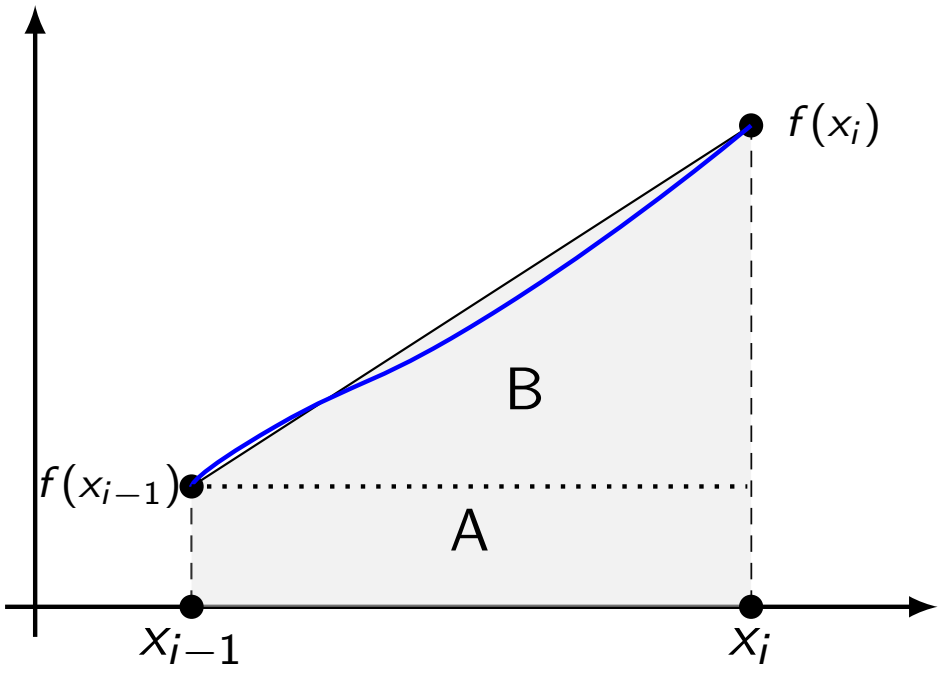


- the grey trapezoid limited on the X-axes by x_{i-1} and x_i already well approximates $\int_{x_{i-1}}^{x_i} f(x)dx$
- this is also true for the trapezoid starting with $p = x_0$, or with x_2, or with x_{n-1}, but not for the others
- obviously, when we shorten the interval size, each (very thin) trapezoid well approximates f in the corresponding interval

figure from `https://tex.stackexchange.com/questions/110598/trapezium-rule-for-integration-using-tikz`

Numerical integration with trapezoids (2/12)

- so we compute the integral as $\sum_{i=1}^{n} \underbrace{\int_{x_{i-1}}^{x_i} f(x)dx}_{\text{approx. by trapezoid}}$,
- have a closer look at a single trapezoid and determine its size when $f(x_{i-1}) \leq f(x_i)$:

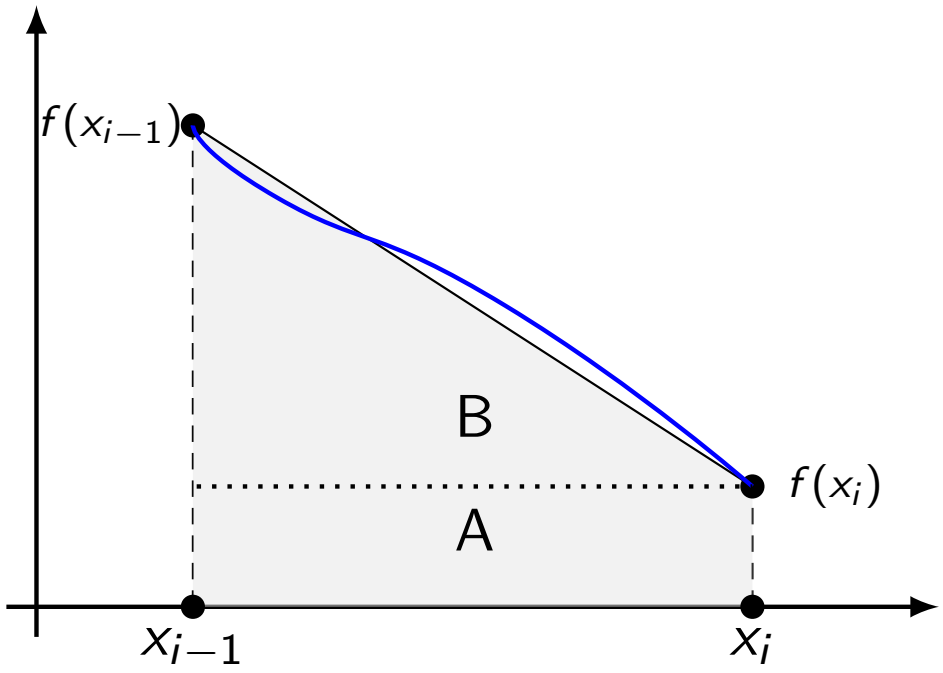


size of grey area

$$
\begin{aligned}
& \overbrace{(x_i - x_{i-1}) \cdot f(x_{i-1})}^{\text{rectangle A}} + \overbrace{\tfrac{1}{2} \cdot (x_i - x_{i-1}) \cdot (f(x_i) - f(x_{i-1}))}^{\text{triangle B}} \\
= \; & (x_i - x_{i-1}) \cdot (f(x_{i-1}) + \tfrac{1}{2} \cdot (f(x_i) - f(x_{i-1}))) \\
= \; & (x_i - x_{i-1}) \cdot (f(x_{i-1}) + \tfrac{1}{2} \cdot f(x_i) - \tfrac{1}{2} \cdot f(x_{i-1})) \\
= \; & (x_i - x_{i-1}) \cdot \tfrac{1}{2} \cdot (f(x_{i-1}) + f(x_i))
\end{aligned}
$$

Numerical integration with trapezoids (3/12)

– in the case
$f(x_{i-1}) > f(x_i)$,
we obtain:



size of grey area

$$
\begin{aligned}
& \overbrace{(x_i - x_{i-1}) \cdot f(x_i)}^{\text{rectangle A}} + \overbrace{\tfrac{1}{2} \cdot (x_i - x_{i-1}) \cdot (f(x_{i-1}) - f(x_i))}^{\text{triangle B}} \\
= \; & (x_i - x_{i-1}) \cdot (f(x_i) + \tfrac{1}{2} \cdot (f(x_{i-1}) - f(x_i))) \\
= \; & (x_i - x_{i-1}) \cdot (f(x_i) + \tfrac{1}{2} \cdot f(x_{i-1}) - \tfrac{1}{2} \cdot f(x_i)) \\
= \; & (x_i - x_{i-1}) \cdot \tfrac{1}{2} \cdot (f(x_{i-1}) + f(x_i))
\end{aligned}
$$

Numerical integration with trapezoids (4/12)

– to summarize the method, we

1. take many points at even distance, say d, on the X-axes:

$$p, p+d, p+2d, p+3d, \ldots, p+n \cdot d = q$$

2. compute $(x_i - x_{i-1}) \cdot \frac{1}{2} \cdot (f(x_{i-1}) + f(x_i))$ for two consecutive points x_{i-1} and x_i on the X-axes, where

$$\begin{aligned} x_{i-1} =& p + (i-1) \cdot d \text{ and} \\ x_i =& p + i \cdot d \end{aligned}$$

3. and sum this up:

Numerical integration with trapezoids (5/12)

$$\sum_{i=1}^{n}(x_i - x_{i-1}) \cdot \frac{1}{2} \cdot (f(x_{i-1}) + f(x_i)) = \frac{1}{2} \cdot d \cdot \sum_{i=1}^{n}(f(x_{i-1}) + f(x_i))$$

$$= \frac{1}{2} \cdot d \left(\sum_{i=1}^{n} f(x_{i-1}) + \sum_{i=1}^{n} f(x_i) \right)$$

$$= \frac{1}{2} \cdot d \left(\sum_{i=0}^{n-1} f(x_i) + \sum_{i=1}^{n} f(x_i) \right)$$

$$= \frac{1}{2} \cdot d \left(f(x_0) + \sum_{i=1}^{n-1} f(x_i) + \sum_{i=1}^{n-1} f(x_i) + f(x_n) \right)$$

$$= \frac{1}{2} \cdot d \left(f(x_0) + f(x_n) + 2 \cdot \sum_{i=1}^{n-1} f(x_i) \right) = d \left(\frac{1}{2} (f(p) + f(q)) + \sum_{i=1}^{n-1} f(x_i) \right)$$

Numerical integration with trapezoids (6/12)

- the sum $d \cdot \left(\frac{1}{2}\left(f(p) + f(q)\right) + \sum_{i=1}^{n-1} f(x_i)\right)$ (also called composite trapezoidal rule) can now be turned into a Python-function

```
def approx_integral_trpz(f,p,q,n):
  fsum = 0.0
  d = (q - p)/n
  for i in range(1,n):
    fsum += f(p + i * d)
  return d * (0.5 * (f(p) + f(q)) + fsum)
```

- note that the function `f` we integrate is given as a parameter to `approx_integral_trpz`
- this abstraction allows to use `approx_integral_trpz` for any other function mapping floating point values to floating point values

Numerical integration with trapezoids (7/12)

- we now want to apply the approximation to function `curvedM` in the interval from 0 to 4 (as in the plot on frame 68)
- determine the effect of the number n of steps
- we start with $n = 10$ and multiply n by 10 in each iteration, while we have not exceeded a defined number `maxsteps`
- as we want to apply the same scheme to other functions, we implement a corresponding function

```
def approx_integral_trpz_print(f,p,q,maxsteps,expected = None):
  n = 10
  numerical = None # as we use it after the loop
  while n <= maxsteps:
    numerical = approx_integral_trpz(f,p,q,n)
    print('n = {:7d}, integral({},{:.1f},{:.1f}) = {:.6f}'
          .format(n,f.__name__,p,q,numerical))
    n *= 10
  if expected and numerical:
    equality = '==' if expected == numerical else '!='
    print('numerical = {:.8f} {} {:.8f} = expected'
          .format(numerical,equality,expected))
```

Numerical integration with trapezoids (8/12)

- note that we use the name of the function `f` as part of the output
- we have an optional parameter `expected` (to be used later)
- if it is used and not `None`, we compare it with the numerically determined integral

```
approx_integral_trpz_print(curvedM,0.0,4.0,maxsteps)
```

```
n =      10, integral(curvedM,0.0,4.0) = 10.151812
n =     100, integral(curvedM,0.0,4.0) = 10.324944
n =    1000, integral(curvedM,0.0,4.0) = 10.326215
n =   10000, integral(curvedM,0.0,4.0) = 10.326228
n =  100000, integral(curvedM,0.0,4.0) = 10.326228
```

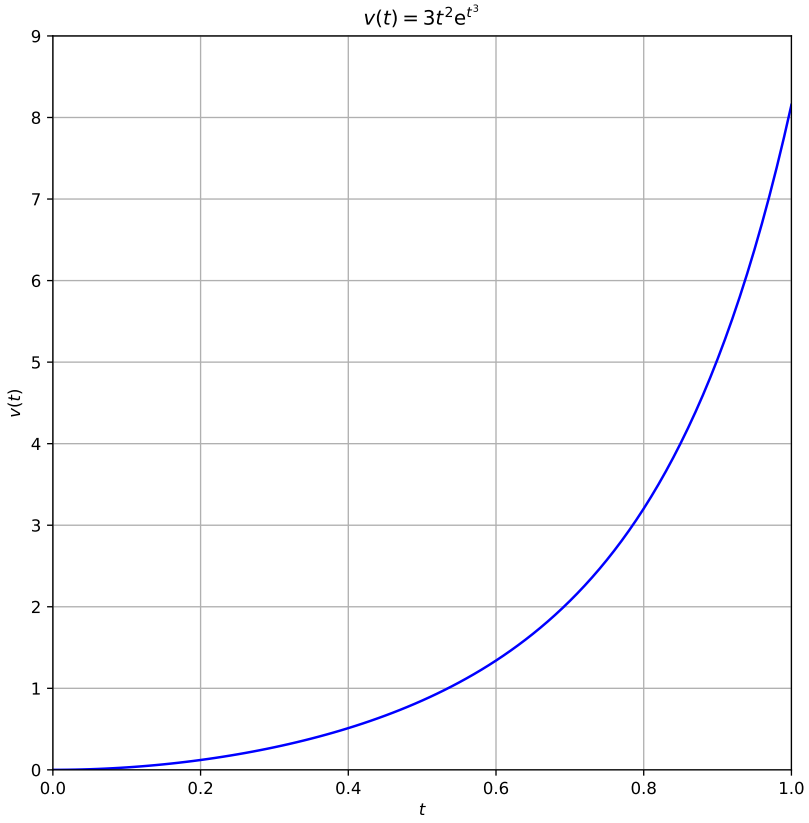
- program runs in about 0.6 seconds on a Intel Core 5 with 2.7 Ghz.

Numerical integration with trapezoids (9/12)

- how can we be sure that the result is correct?
- we should test the program for functions for which we can analytically determine the integral

- example from physics:
- you speed up your vehicle from rest and wonder how far you get in n seconds
- distance is given by integral $\int_0^n v(t)dt$ where $v(t)$ is the velocity as a function of time
- a rapidly increasing velocity function is $v(t) = 3t^2 \mathrm{e}^{t^3}$ where $\mathrm{e} \approx 2.718$ is the Euler number

- we should first plot v:



example from Linge & Langtangen, Programming for Computations, Springer Verlag 2016

Numerical integration with trapezoids (10/12)

- the anti-derivative (Stammfunktion) of v is $V(t) = \mathrm{e}^{t^3}$ (according to `https://www.integral-calculator.com/`), and so

$$\int_0^1 v(t)dt = V(1) - V(0) = \mathrm{e}^{1^3} - \mathrm{e}^{0^3} = \mathrm{e}^1 - \mathrm{e}^0 = \mathrm{e} - 1 \approx 1.718$$

- verify that the numerical integration method delivers this result
- implement v and its anti-derivative as the following Python-functions (assuming that the module `math` was imported):

```
def velocity(t):
  return 3 * t * t * (math.pow(math.e,t * t * t))

def velocity_anti_derivative(t):
  return math.pow(math.e,t * t * t)
```

Numerical integration with trapezoids (11/12)

- the latter function allows us to compute the expected value in the interval from 0 to 1:

```
expected =  velocity_anti_derivative(1.0) - \
            velocity_anti_derivative(0.0)
```

- as previous functions `approx_integral_trpz` and `approx_integral_trpz_print` abstract from the concrete function to integrate, we can reuse the latter here, this time additionally providing the expected value

```
approx_integral_trpz_print(velocity,0.0,1.0,maxsteps,expected)
```

```
n =      10, integral(velocity,0.0,1.0) = 1.752043
n =     100, integral(velocity,0.0,1.0) = 1.718622
n =    1000, integral(velocity,0.0,1.0) = 1.718285
n =   10000, integral(velocity,0.0,1.0) = 1.718282
n =  100000, integral(velocity,0.0,1.0) = 1.718282
numerical = 1.71828183 != 1.71828183 = expected
```

Numerical integration with trapezoids (12/12)

- so the numbers look like they are identical, but the comparison tells us the opposite
- of course, the numerical method only delivers an approximation and it is very likely that the value is not identical to the correct value
- in fact, if we would show the floating point values with 9 instead of just 8 digits, the difference would become visible
- but there is a general problem here: as the correct integral is $\mathrm{e} - 1$ and e is an irrational number, so is $\mathrm{e} - 1$
- that is, we cannot represent the result by a fraction of integers

section from Linge & Langtangen, Programming for Computations, Springer Verlag 2016

Finite Precision of Floating-Point Numbers (1/7)

- one key result of the previous section was that the integral value $\mathrm{e} - 1$ is irrational, i.e. we cannot represent it by a fraction of integers

$\Rightarrow$ we will not be able to represent it as a floating point number in Python, since this only has a limited number of digits

- so does this feature hold only for irrational numbers?

  NO, as shown in the following example

- consider the addition `1 + 2` with a comparison of the expected result (executed in the Python-shell):

```
>>> a = 1; b = 2; expected = 3;
>>> a + b == expected
```

$\Rightarrow$ `True`

- now consider a similar evaluation for floating point numbers:

```
>>> a = 0.1; b = 0.2; expected = 0.3
>>> a + b == expected
```

$\Rightarrow$ `False`

Finite Precision of Floating-Point Numbers (2/7)

- to understand what happened, print out the four values `0.1`, `0.2`, `0.1+0.2` and `0.3` with a precision of 17 digits:

```
>>> fs = '{:.17f}\n' * 4
>>> print(fs.format(0.1,0.2,0.1 + 0.2,0.3))
```

```
0.10000000000000001
0.20000000000000001
0.30000000000000004
0.29999999999999999
```

- obviously, the four real numbers are only approximated by a floating point value
- this holds in general, because
  - there are infinitely many different real numbers, but
  - only a finite number of floating point values, as each value can only store a limited amount of information (i.e. bits)

$\Rightarrow$ so each floating point number must represent infinitely many different real numbers

Finite Precision of Floating-Point Numbers (3/7)

- this means to approximate the real numbers by floating point numbers
- in general, in Python floating point values have (at most) 16 correct digits
- so what happens if we apply arithmetic operations to real numbers that are inaccurately represented?
- such operations lead to small rounding errors
- these rounding errors may or may not accumulate, so the result may become incorrect
- this becomes apparent in the output above with the inaccurate digit in the 17th decimal place, leading to differences in the (very simple) calculation
- if tests like `0.1 + 0.2 == 0.3` do not lead to mathematically correct results, what should we do then?

Finite Precision of Floating-Point Numbers (4/7)

- answer: accept some small inaccuracy in floating point calculation and make a test with a tolerance, like this:

```
>>> a = 0.1; b = 0.2; expected = 0.3; computed = a + b
>>> diff = abs(expected - computed)
>>> tol = 1e-15
>>> diff < tol
```

$\Rightarrow$ `True`

Finite Precision of Floating-Point Numbers (5/7)

- the actual value of `diff` is around $5.55112 \cdot 10^{-17}$, so the tolerance value of $1.0 \cdot 10^{-15}$ is conservative
- however, in general an appropriate choice of the tolerance is difficult, as the absolute differences depend on the magnitude of the numbers involved in the calculations, see the following example:

```
def difference(k):
  return 10**k + 0.3 - (10**k + 0.1 + 0.2)

for k in range(1,10+1):
  print('{:>2d}\t{:>+.17f}\t{:>.17f}'
        .format(k,difference(k),10.0 ** (k-16)))
```

```
 1 +0.00000000000000178 0.00000000000000100
 2 +0.00000000000000000 0.00000000000001000
 3 -0.00000000000011369 0.00000000000010000
 4 -0.00000000000181899 0.00000000000100000
 5 +0.00000000000000000 0.00000000001000000
 6 +0.00000000011641532 0.00000000010000000
 7 +0.00000000186264515 0.00000000100000000
 8 +0.00000000000000000 0.00000001000000000
 9 -0.00000011920928955 0.00000010000000000
10 -0.00000190734863281 0.00000100000000000
```

- the mathematically correct result is 0
- for all $k \notin \{2, 5, 8]$ there is a difference of $\approx 10^{k-16}$
- difference increases with $k \Rightarrow$ increase tolerance with larger values of k

Finite Precision of Floating-Point Numbers (6/7)

- so to determine the tolerance we would have to know the magnitude of the numbers we compare, which is not always easy to determine
- a possible solution of this problem is to consider the absolute difference in relation to the expected value, i.e. to compute a relative difference, as shown in the following example:

```
for k in range(1,10+1):
  d = difference(k)
  v = 10**k + 0.3
  print('{:>2d}\t{:.2g}'
        .format(k,d/v))
```

```
 1 1.7e-16
 2 0
 3 -1.1e-16
 4 -1.8e-16
 5 0
 6 1.2e-16
 7 1.9e-16
 8 0
 9 -1.2e-16
10 -1.9e-16
```

- so the relative difference (if any) is $\approx 10^{-16}$, i.e. independent of k

Finite Precision of Floating-Point Numbers (7/7)

Synopsis on floating point values

- when working with real numbers, always keep in mind, that they are approximated by floating point values
- this is due to the fact, that finitely many floating point numbers must represent infinitely many real numbers
- calculations on floating point numbers lead to small differences (compared to the exact mathematical calculation)
- when comparing floating point numbers allow a small difference, according to a tolerance value
- the allowed tolerance can be absolute or relative

Generalizing the numerical integration code framework (1/2)

this section is subject to self study

- we now apply these insights to our integration example, and print the differences to the expected result (this time also interested in the sign)
- as we want to do this for other integration methods considered later, we write a generic function, which takes the method as a parameter

```
def approx_integral_generic_print(imethod,f,p,q,maxsteps,expected):
  print('method: {}, integral({},{},{}):'
        .format(imethod.__name__,f.__name__,p,q))
  n = 10
  while n <= maxsteps:
    numerical = imethod(f,p,q,n)
    diff = expected - numerical
    print('n = {:7d}, numerical = {:.6f}, diff={:.2e}'
          .format(n,numerical,diff))
    n *= 10
```

Generalizing the numerical integration code framework (2/2)

```
method: approx_integral_trpz, integral(velocity,0.0,1.0):
n =      10, numerical = 1.752043, diff=-3.38e-02
n =     100, numerical = 1.718622, diff=-3.40e-04
n =    1000, numerical = 1.718285, diff=-3.40e-06
n =   10000, numerical = 1.718282, diff=-3.40e-08
n =  100000, numerical = 1.718282, diff=-3.40e-10
n = 1000000, numerical = 1.718282, diff=-3.37e-12
               expected = 1.718282
```

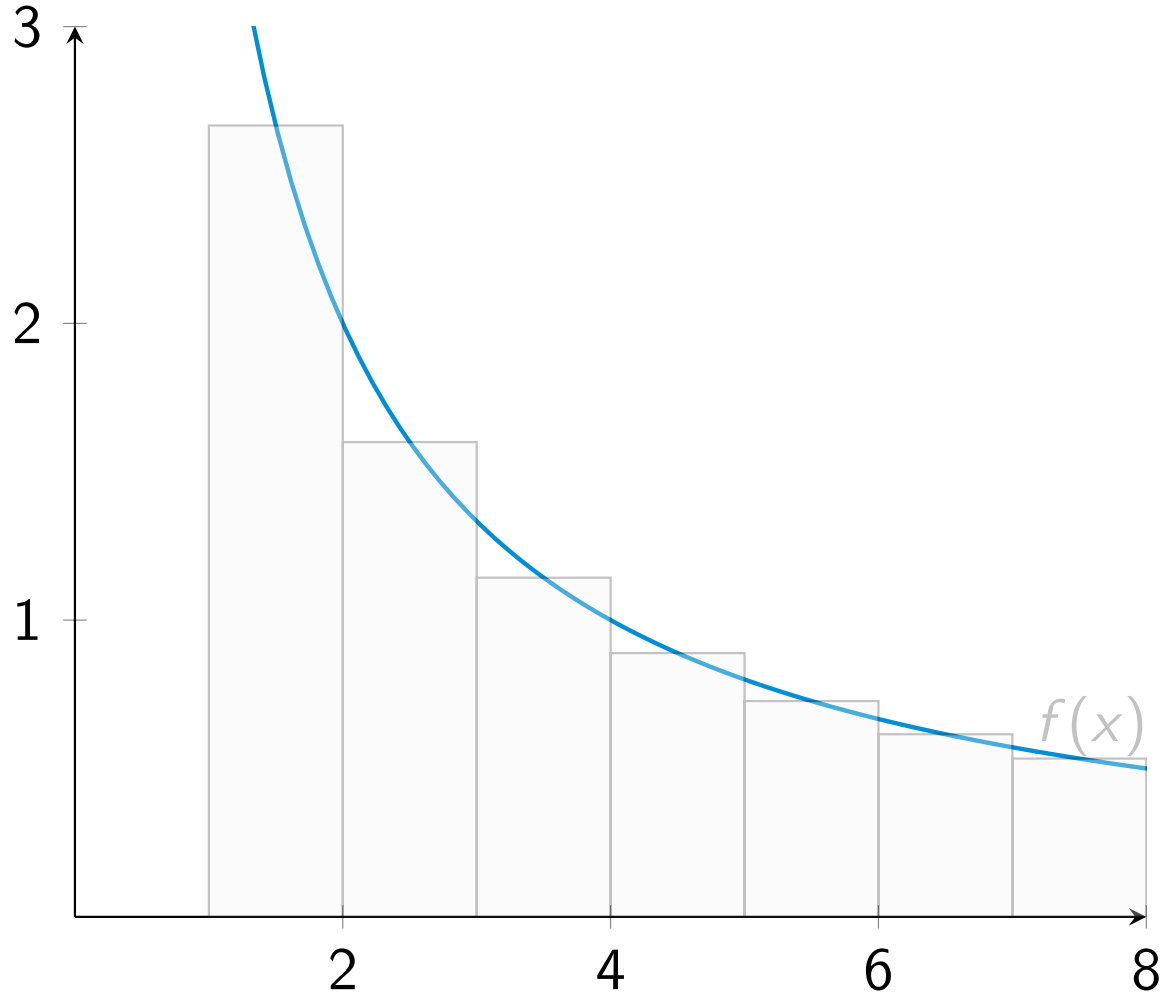
- the output shows that numerical integration with the composite trapezoid method
  - only needs 10 000 intervals (i.e. steps) to compute the correct solution up to a precision of 6 digits
  - for $n = 100\,000$, a small difference of 10^{-10} remains
  - the difference is negative, i.e. the numerical method slightly underestimates the integral
  - increasing the number of steps by a factor of 10 reduces the difference by a factor of ≈ 100

Numerical integration using midpoints (1/4)

this section is subject to self study

- instead of trapezoids, use plain rectangles to approximate the integral

- idea: take the function value at the midpoint of an interval and multiply it by the interval width
- this gives integral of grey rectangles, and these are summed to obtain $\int_p^q f(x)dx$



section follows Linge & Langtangen, Programming for Computations, Springer Verlag 2016

Numerical integration using midpoints (2/4)

- for a distance d of consecutive interval boundaries and the midpoint $m_i = p + i \cdot d + \frac{d}{2}$ of the ith interval, for $0 \leq i \leq n-1$, we compute

$$\sum_{i=0}^{n-1} \underbrace{d}_{\substack{\text{rectangle}\\ \text{width}}} \cdot \underbrace{f(m_i)}_{\substack{\text{rectangle}\\ \text{height}}} = d \cdot \sum_{i=0}^{n-1} f(m_i)$$

- this is easily turned into a Python-function:

```
def approx_integral_mid(f,p,q,n):
  fsum = 0.0
  d = (q - p)/n
  for i in range(0,n):
    fsum += f(p + i * d + d/2)
  return d * fsum
```

Numerical integration using midpoints (3/4)

- use the same scheme as before to evaluate the integral for an increasing number of steps:

```
approx_integral_generic_print(approx_integral_mid,velocity,
                              0.0,1.0,maxsteps,expected)
print('{}expected = {:.6f}'.format(' ' * 14,expected))
```

- here is the result, when using the same value for the variable `expected` as before

```
method: approx_integral_mid, integral(velocity,0.0,1.0):
n =      10, numerical = 1.701483, diff=1.68e-02
n =     100, numerical = 1.718112, diff=1.70e-04
n =    1000, numerical = 1.718280, diff=1.70e-06
n =   10000, numerical = 1.718282, diff=1.70e-08
n =  100000, numerical = 1.718282, diff=1.70e-10
              expected = 1.718282
```

- the output shows that numerical integration with the midpoint method

Numerical integration using midpoints (4/4)

- only needs 10 000 intervals (i.e. steps) to compute the correct solution up to a precision of 6 digits
- for $n = 100\,000$, a small difference of 10^{-10} remains
- the error is positive, i.e. the midpoint method slightly overestimates the integral
- increasing the number of steps by a factor of 10 reduces the difference by a factor of $\approx 100 = 10^2$, i.e. it the method has a convergence rate of 2

– in comparison with the trapezoid method, the absolute value of the difference to the expected value is smaller by a factor of ≈ 2

Speeding up numerical integration using numpy's vectorization (1/9)

- numerical integration methods (even if implemented in Python) are already very fast for a single problem instance
- but in some applications (e.g. navigation systems) numerical integration problems have to be solved for many instances with varying functions and interval boundaries
- so speeding up numerical integration is important and we will see that it is not difficult to achieve
- the two methods for numerical integration have a very simple structure
- view them as methods which, for a vector of evenly spaced points x_i, computes a vector of function values $f(x_i)$ which are summed up

Speeding up numerical integration using numpy's vectorization (2/9)

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
$f(x_0)$	$f(x_1)$	$f(x_2)$	$f(x_3)$	$f(x_4)$	$f(x_5)$	$f(x_6)$	$f(x_7)$	$f(x_8)$	$f(x_9)$	$f(x_{10})$	$f(x_{11})$	$f(x_{12})$	$f(x_{13})$	$f(x_{14})$	$f(x_{15})$

$$\sum$$

- this is a typical structure of numerical algorithms
- it is amendable to vectorization, which compute such vectors and their sum extremely fast
- the speedup is achieved by using instructions of the processor which can compute a constant number (usually 4) of function values and their sum in a single CPU-cycle
- such instructions are available and easy to use via `numpy`, a very widely used module of Python

Speeding up numerical integration using numpy's vectorization (3/9)

- as we want to apply `numpy` to the velocity function v, we first have to implement a version which uses `numpy`-methods instead of methods from the module `math`:

```
import numpy as np

def np_velocity(t):
  return 3 * t * t * (np.power(np.e,t * t * t))
```

- the midpoint method can be implemented in four lines of Python code

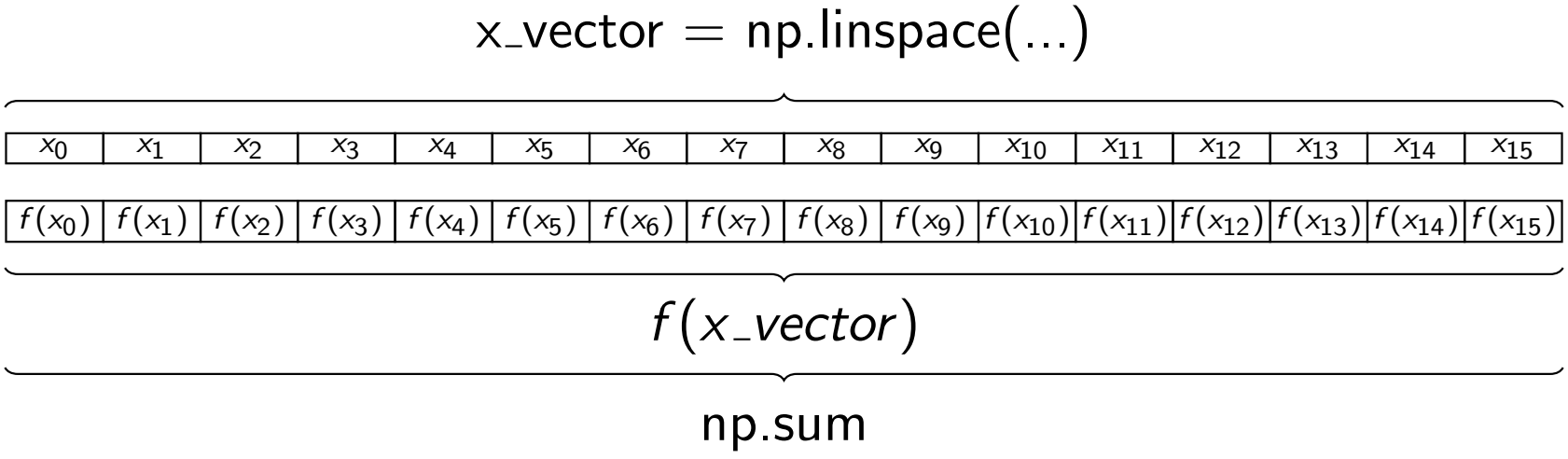
```
def np_approx_integral_mid(f, p, q, n):
  d = (q-p)/n
  x_array = np.linspace(p + d/2, q - d/2, n)
  return d * np.sum(f(x_array))
```

- the interface is the same as before, but requires that the function to integrate only uses `numpy`-methods besides basic arithmetic operations

Speeding up numerical integration using numpy's vectorization (4/9)

- the first step is to determine (as done previously) the distance of two consecutive interval boundaries inside the integration interval $[p, q]$,
- the second step computes an array of n evenly spaced points using the `linspace`-method of `numpy`
- besides n, this method requires the specification of the
  - first value $p + \frac{d}{2}$ of the vector (the midpoint of the first interval),
  - last value $q - d/2$ of the vector (the midpoint of the last interval)
- applying the function `f` to each value of this array `x_array` is expressed by applying `f` to `x_array`
- this gives a new array of function values which are summed up using the `sum`-method from `numpy`
- the structure can be depicted as follows:

Speeding up numerical integration using numpy's vectorization (5/9)



- we now want to measure the runtime of the different methods to verify if the effort was worth it
- we use the class `Timer` from module `timeit` and the `partial`-method from `functools`
- additionally we have to import the integration methods and the functions we want to integrate

Speeding up numerical integration using numpy's vectorization (6/9)

```
from timeit import Timer
from approx_integral import approx_integral_mid, \
                            np_approx_integral_mid
from funcdefs import velocity, np_velocity
from functools import partial
```

- we cannot directly supply the timer with a function call
- instead we need to create a partial object, that behaves like the corresponding function call, when actually called
- such a partial object is created by the method `partial`, which takes a function and a list of its arguments as parameter
- to reuse it, we encapsulate the creation of the partial object and the call to the timer in the following function

Speeding up numerical integration using numpy's vectorization (7/9)

```
def runtime_get(func,*args):
  partial_object = partial(func,*args)
  times = Timer(partial_object).repeat(3,1)
  return min(times)
```

- it returns the minimum of the runtime of three repetitive calls to the given function with the given argument
- for the runtime measurement, we specify the concrete boundaries, the number of steps, and provide `runtime_get` with the function, for which we want to measure the runtime

Speeding up numerical integration using numpy's vectorization (8/9)

```
p = 0.0
q = 1.0
n = 10000000
t = runtime_get(approx_integral_mid,
                velocity,p,q,n)
print('runtime for approx_integral_mid: {:.2f} s'
      .format(t))
t = runtime_get(np_approx_integral_mid,
                np_velocity,p,q,n)
print('runtime for np_approx_integral_mid: {:.2f} s'
      .format(t))
```

- for the chosen value of $n = 10\,000\,000$ we see that the `numpy`-based integration method is faster by a factor of ≈ 14 compared the direct implementation using its own for-loop

```
runtime for approx_integral_mid: 6.97 s
runtime for np_approx_integral_mid: 0.50 s
```

runtime for pure C-version of approx_integral_mid: 0.3 s

Speeding up numerical integration using numpy's vectorization (9/9)

- from the vectorization, we would expect a speedup of a factor of at most 4 (because the vectorization handles four floating point values in one computation cycle)
- the additional speedup comes from the fact that the entire iterations of `np_approx_integral_mid` are performed inside the methods `linspace` and `sum`
- their calls are executed very fast by corresponding library functions not implemented in Python
- the Python interpreter is not involved in the execution of these methods, except that it provides the methods with their arguments and receives their results

Integration: final remark

- the two methods shown here are just two examples of many possible numerical integration rules
- other methods are Simpson's method and Gauss quadrature method
- they are all based on the same principle:

$$\int_p^q f(x)dx \approx \sum_{i=0}^{n} w_i f(x_i)$$

- that is, the integral is approximated by a sum of function values for some x_i which get a weight w_i.
- the methods differ in the way they construct the evaluation points x_i and the weights w_i
- some methods use equally spaced points x_i, but higher accuracy can be obtained by varying the spacing between consecutive points.

section follows Linge & Langtangen, Programming for Computations, Springer Verlag 2016

Basics of Numpy (1/1)

Numpy

- extends Python by 1-dim arrays and multidimensional arrays of fixed base type
- is closer to hardware, and thus more efficient than equivalent Python code that uses lists
- is designed for scientific computation and provides many convenient features for this area

- for all code we show here, we assume that the statement

  ```
  import numpy as np
  ```

  appears before it

section is a condensed version of `http://www.scipy-lectures.org/intro/numpy/numpy.html`

Creating Numpy arrays (1/10)

- creating a 1-dim array (also called array from now on) or a multidimensional array (also called matrix from now on), requires little more syntax than introducing a corresponding list:

```
one_dim_array = np.array([0.0, 1.0, 2.0, 3.0])
two_dim_array = np.array([[0, 1, 2], [3, 4, 5]])
print('one_dim_array=\n{}'.format(one_dim_array))
print('two_dim_array=\n{}'.format(two_dim_array))
```

```
one_dim_array=
[ 0.  1.  2.  3.]
two_dim_array=
[[0 1 2]
 [3 4 5]]
```

- the method which converts an array to a string uses spaces as separators and indentation for ease of readability

Creating Numpy arrays (2/10)

- while lists allow elements of different type, the elements of an array all have the same type
- the name of the type is stored in the `dtype`-attribute of the array

```
print('one_dim_array.dtype={}'.format(one_dim_array.dtype))
print('two_dim_array.dtype={}'.format(two_dim_array.dtype))
```

```
one_dim_array.dtype=float64
two_dim_array.dtype=int64
```

- the attribute `ndim` stores the number of dimensions of a numpy array

```
print('one_dim_array.ndim={}'.format(one_dim_array.ndim))
print('two_dim_array.ndim={}'.format(two_dim_array.ndim))
```

```
one_dim_array.ndim=1
two_dim_array.ndim=2
```

Creating Numpy arrays (3/10)

- the attribute `shape` stores the tuple of array dimensions of the corresponding array
- for the previous arrays `one_dim_array = [1 2 3 4]` and `two_dim_array = [[0 1 2],[3 4 5]]` we get:

```
print('one_dim_array.shape={}'.format(one_dim_array.shape))
print('two_dim_array.shape={}'.format(two_dim_array.shape))
```

```
one_dim_array.shape=(4,)
two_dim_array.shape=(2, 3)
```

- `len(a)` is the size of the first dimension of the array:

```
print('len(one_dim_array)={}'.format(len(one_dim_array)))
print('len(two_dim_array)={}'.format(len(two_dim_array)))
```

```
len(one_dim_array)=4
len(two_dim_array)=2
```

Creating Numpy arrays (4/10)

- a 1-dim array of base type `int` is often constructed using `arange`
- and a 1-dim array of base type `float64` is often constructed using `linspace`:

```
one_dim_array_int = np.arange(10)
one_dim_array_float = np.linspace(0,1,6) # first, last, num entries
print('one_dim_array_int=\n{}'.format(one_dim_array_int))
print('one_dim_array_float=\n{}'.format(one_dim_array_float))
```

```
one_dim_array_int=
[0 1 2 3 4 5 6 7 8 9]
one_dim_array_float=
[ 0.   0.2  0.4  0.6  0.8  1. ]
```

Creating Numpy arrays (5/10)

- a matrix initialized with 1 can be created with the method `ones`

```
onesmatrix = np.ones((3, 3))
print('onesmatrix=\n{}'.format(onesmatrix))
```

```
onesmatrix=
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
```

- similarly, method `zeros` creates a matrix initialized with 0

```
zerosmatrix = np.zeros((2, 3))
print('zerosmatrix=\n{}'.format(zerosmatrix))
```

```
zerosmatrix=
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
```

Creating Numpy arrays (6/10)

- method `eye` creates a square matrix in which the main diagonal is initialized to 1 and all other values are 0, a unit matrix

```
maindiag1matrix = np.eye(3)
print('maindiag1matrix=\n{}'.format(maindiag1matrix))
```

```
maindiag1matrix=
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

Creating Numpy arrays (7/10)

- method `diag` is a little more general, as it allows to specify the values on the main diagonal

```
maindiagsetmatrix = np.diag(np.array([1, 2, 3, 4]))
print('maindiagsetmatrix=\n{}'.format(maindiagsetmatrix))
```

```
maindiagsetmatrix=
[[1 0 0 0]
 [0 2 0 0]
 [0 0 3 0]
 [0 0 0 4]]
```

Creating Numpy arrays (8/10)

- here is a little task: create the following 4×4-matrix:

```
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 2]
 [1 6 1 1]]
```

- as most values are 1, we could use `np.ones`
- values $\neq 1$ can explicitly be set in two assignment statements
- this is possible, since `numpy`-arrays are mutable

```
matrix1 = np.ones((4,4),
                  dtype=int)
matrix1[2,3] = 2
matrix1[3,1] = 6
```

- note the slight difference in the notation: while in a 2-dim list m element (i, j) would be addressed by `m[i][j]`, in a `numpy`-array m we use `m[i,j]`, thus saving one symbol

Creating Numpy arrays (9/10)

- now create the following 4×3-matrix

```
[[ 0.  0.  0.]
 [ 2.  0.  0.]
 [ 0.  3.  0.]
 [ 0.  0.  4.]]
```

- it is similar to one we have seen before, but with the non-zero values shifted below the main diagonal
- if we would add an additional first column to the target matrix, the non-zero values would be back on the main diagonal
- to create this matrix, we use the same matrix as before (with floats instead of ints) and just delete the first column of the matrix

```
maindiagonal = np.linspace(1,4,4)
matrix_main_diag = np.diag(maindiagonal)
matrix_main_diag_shift = np.delete(matrix_main_diag,axis=1,obj=0)
```

- for the delete method we specify:
  - the array in which we want to delete something
  - the axis the deletion refers to: axis 1, i.e. the column in our case
  - the column number to delete (0 in our case)

Creating Numpy arrays (10/10)

- matrices often consist of repetitive parts, like this matrix:

```
[[4 3 4 3 4 3]
 [2 1 2 1 2 1]
 [4 3 4 3 4 3]
 [2 1 2 1 2 1]]
```

  - we see two rows appearing twice and each of the rows consists of a subarray appearing three times

- to construct this matrix, we can apply `np.tile`, for which we specify, the repetitive element and how many times it is repeated

```
firstline = np.tile([4,3],3)
secondline = np.tile([2,1],3)
matrix_tiled = np.array(np.tile([firstline,secondline],(2,1)))
print(matrix_tiled)  # outputs matrix shown at top left
```

- in the last application of `np.tile` (line 3) we specify that we want to copy the rows twice (first parameter 2), but do not want to copy the columns (parameter 1)

synopsis: methods for creating numpy arrays

`np.array([1,2])`	create a 1-dim array
`np.array([[1,2],[3,4]])`	create a 2-dim array (matrix)
`np.arange(n)`	create 1-dim `int`-array with `n` elements
`np.linspace(i,j,k)`	create 1-dim `float`-array with `k` elements evenly spread on the interval from `i` to `j`
`np.ones((r,c))`	create 2-dim `float`-array with `r` rows and `c` columns, initialized to 1.0
`np.zeros((r,c),dtype=int)`	create 2-dim `int`-array with `r` rows and `c` columns, initialized to 0
`np.eye(r)`	create 2-dim `float`-array with `r` rows and columns; main diagonal is 1, other are 0
`np.diag(md)`	create 2-dim array; main-diagonal consists of values in `md`; other values are 0
`m[i,j] = v`	update matrix-value of row `i` and column `j` to `v`
`np.delete(source,obj=j,axis=i)`	in matrix `source` delete object number `j` on axis `i`; `i=1` ⇒ columns, `i=0` ⇒ rows
`np.tile(rep,n)`	create `n` copies of `rep`

Resizing Numpy arrays (1/1)

- in some cases we have to modify the size of an array
- most often we increase the size to accommodate more elements, as in the following examples
- in the first case we add four elements to the array, in the second one additional row of two elements, all initialized to 0

```
arr = np.array([1,2,3,4])
arr.resize((8,))
matrix = np.ones((2,2),dtype=int)
matrix.resize(3,2)
print('arr={}'.format(arr))
print('matrix=\n{}'.format(matrix))
```

```
arr=[1 2 3 4 0 0 0 0]
matrix=
[[1 1]
 [1 1]
 [0 0]]
```

Slicing Numpy arrays (1/5)

- slicing is a powerful operation and can also be applied to 1-dim arrays and matrices
- for a matrix `m` the slice `m[a:b:c,d:e:f]` means the following:
  - obtain slice of matrix `m` starting with row `a`, ending (exclusive) with row `b` and with steps of size `c`
  - all three values can be omitted, in which case they have default values `0,numrows,1`, where `numrows` is the number of rows
  - `d:e:f` specify the corresponding values for columns
  - for 1-dim array omit `,d:e:f`

Slicing Numpy arrays (2/5)

```
onedimarray = np.arange(10)    # [0 1 2 3 4 5 6 7 8 9]
onedimarray_sliced = onedimarray[2:9:3] # start:end(exclusive):step
print(onedimarray_sliced)
```

```
[2 5 8]
```

- for matrices, we specify the slicing coordinates for both dimensions
- consider this in a series of examples, all referring to the following matrix stored in the variable `square_matrix`

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
```

Slicing Numpy arrays (3/5)

- the first slicing extracts from the first row (at index 0) the elements in the fourth and fifth column (beginning at index 3)

```
onfirstrow = square_matrix[0,3:5]
print(onfirstrow)
```

```
[3 4]
```

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 & 24 \end{pmatrix}$$

- let us now slice the south east part of the matrix beginning in the fourth row and fourth column (both at index 3)

```
south_east22 = square_matrix[3:,3:]
print(south_east22)
```

```
[[18 19]
 [23 24]]
```

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 & 24 \end{pmatrix}$$

Slicing Numpy arrays (4/5)

- next slice all elements of the third column (at index 2):

```
column3 = square_matrix[:,2]
print(column3)
```

```
[ 2  7 12 17 22]
```

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 & 24 \end{pmatrix}$$

- finally slice every second element of the third and fifth row, i.e. we begin at row 2 and use a step width of 2

```
scattered = square_matrix[2::2,::2]
print(scattered)
```

```
[[10 12 14]
 [20 22 24]]
```

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 & 24 \end{pmatrix}$$

Slicing Numpy arrays (5/5)

- it is important to note that slicing does not create a copy of the elements in the original array, but only a view
- this saves time, but also means that an update on the sliced array affects the original, see the following example

```
arr = np.arange(8)
arr_step2 = arr[::2]
arr_step2[0] = 12
print('arr={}'.format(arr))
print('arr_step2={}'.format(arr_step2))
```

```
arr=[12  1  2  3  4  5  6  7]
arr_step2=[12  2  4  6]
```

Numerical operations on Numpy arrays (1/17)

- adding some scalar value (like an integer or a floating point number) to all array elements is very easy:

```
arr = np.arange(4)
arr_plus1 = arr + 1
print(arr_plus1)
```

```
[1 2 3 4]
```

- we can even use an array on the right hand side of an exponentiation operator to obtain a list of exponents of 2:

```
arr = np.arange(9)
twoexponents = 2 ** arr
print(twoexponents)
```

```
[  1   2   4   8  16  32  64 128 256]
```

Numerical operations on Numpy arrays (2/17)

– the most common arithmetic operations we usually apply to scalars are available for arrays, too, provided the arrays are of the same dimensionality and length:

```
summand1 = np.array([3,2,4,7])
summand2 = np.array([5,4,4,1])
sumof = summand1 + summand2
difference = summand1 - summand2
simpleproduct = summand1 * summand2
equality = summand1 == summand2
print('sumof={}'.format(sumof))
print('difference={}'.format(difference))
print('simpleproduct={}'.format(simpleproduct))
print('equality={}'.format(equality))
```

```
sumof=[8 6 8 8]
difference=[-2 -2  0  6]
simpleproduct=[15  8 16  7]
equality=[False False  True False]
```

Numerical operations on Numpy arrays (3/17)

- note that $*$ is not matrix multiplication, which is expressed by the method `dot`:

```
matrix1 = np.array([[3,2,1],[1,0,2]])
matrix2 = np.array([[1,2],[0,1],[4,0]])
matrixproduct = matrix1.dot(matrix2)
print('matrixproduct=\n{}'.format(matrixproduct))
```

```
matrixproduct=
[[7 8]
 [9 2]]
```

- let us verify that this is correct
- we want to compute the product $C = A \cdot B$, where A is an $m \times n$-matrix and B is an $n \times \ell$-matrix.
- in our case, we have $m = 2$, $n = 3$, $\ell = 2$.

Numerical operations on Numpy arrays (4/17)

$\Rightarrow$ C is an $m \times \ell$-matrix C defined by

$$C(i,j) = \sum_{k=0}^{n-1} A(i,k)B(k,j)$$

for all i, j, $0 \leq i \leq m-1$, $0 \leq j \leq n-1$.

$$\begin{pmatrix} 3 & 2 & 1 \\ 1 & 0 & 2 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 \\ 0 & 1 \\ 4 & 0 \end{pmatrix} = \begin{pmatrix} 3 \cdot 1 + 2 \cdot 0 + 1 \cdot 4 & 8 \\ 9 & 2 \end{pmatrix} \; i = 0, \; j = 0$$

$$\begin{pmatrix} 3 & 2 & 1 \\ 1 & 0 & 2 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 \\ 0 & 1 \\ 4 & 0 \end{pmatrix} = \begin{pmatrix} 7 & 3 \cdot 2 + 2 \cdot 1 + 1 \cdot 0 \\ 9 & 2 \end{pmatrix} \; i = 0, \; j = 1$$

Numerical operations on Numpy arrays (5/17)

$$\begin{pmatrix} 3 & 2 & 1 \\ 1 & 0 & 2 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 \\ 0 & 1 \\ 4 & 0 \end{pmatrix} = \begin{pmatrix} 7 & 8 \\ 1 \cdot 1 + 0 \cdot 0 + 2 \cdot 4 & 2 \end{pmatrix} \; i = 1, \; j = 0$$

$$\begin{pmatrix} 3 & 2 & 1 \\ 1 & 0 & 2 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 \\ 0 & 1 \\ 4 & 0 \end{pmatrix} = \begin{pmatrix} 7 & 8 \\ 9 & 1 \cdot 2 + 0 \cdot 1 + 2 \cdot 0 \end{pmatrix} \; i = 1, \; j = 1$$

Numerical operations on Numpy arrays (6/17)

- another important operation on 2-dim arrays is transposition, which turns the columns of a matrix into rows
- it is implemented by the method `T`
- we exemplify it by first computing the upper triangular matrix initialize to 1, using the method `np.triu`
- to the resulting matrix we apply `.T`

```
uppertriangular = np.triu(np.ones((3, 3),dtype=int), 1)
lowertriangular = uppertriangular.T
print('uppertriangular=\n{}'.format(uppertriangular))
print('lowertriangular=\n{}'.format(lowertriangular))
```

```
uppertriangular=
[[0 1 1]
 [0 0 1]
 [0 0 0]]
```

```
lowertriangular=
[[0 0 0]
 [1 0 0]
 [1 1 0]]
```

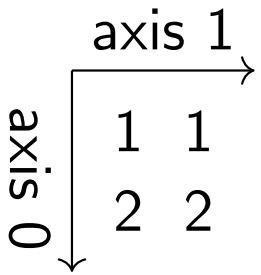
Numerical operations on Numpy arrays (7/17)

- in the context of numerical integration we have seen that we can easily determine the sum of all elements in a 1-dim array

```
arr = np.arange(1,6)
print('sum(arr)={}'.format(np.sum(arr)))
```

```
sum(arr)=15
```

- when we apply `np.sum` to a matrix we have to specify the axis over which we want to sum: `axis=0` for sum of column; `axis=1` for sum of rows




```
matrix = np.array([[1,1],[2,2]])
print('col: sum(matrix,axis=0)={}'.format(np.sum(matrix,axis=0)))
print('row: sum(matrix,axis=1)={}'.format(np.sum(matrix,axis=1)))
```

```
col: sum(matrix,axis=0)=[3 3]
row: sum(matrix,axis=1)=[2 4]
```

Numerical operations on Numpy arrays (8/17)

- another way of aggregating an array is to determine the minimum or the maximum:

```
arr = np.array([2,3,2,8,7,3,4,1])
print('min(arr)={}'.format(np.min(arr)))
print('max(arr)={}'.format(np.max(arr)))
```

```
min(arr)=1
max(arr)=8
```

- sometimes we want to know the index of the minimum/maximum which is delivered by `np.argmin` and `np.argmax`:

```
arr = np.array([2,3,2,8,7,3,4,1])
print('argmin(arr)={}'.format(np.argmin(arr)))
print('argmax(arr)={}'.format(np.argmax(arr)))
```

```
argmin(arr)=7
argmax(arr)=3
```

Numerical operations on Numpy arrays (9/17)

- of course, we can compute the arithmetic mean, or the median or the standard deviation

```
arr = np.array([2,3,2,8,7,3,4,1])
print('mean(arr)={}'.format(np.mean(arr)))
print('median(arr)={}'.format(np.median(arr)))
print('std(arr)={:.2f}'.format(np.std(arr)))
```

```
mean(arr)=3.75
median(arr)=3.0
std(arr)=2.33
```

- let us consider an application to real data, namely population counts of different species in northern Canada at the beginning of the 20th century
- data is available at `http://www.scipy-lectures.org/_downloads/populations.txt` and consists of tab-separated values in 4 columns:

Numerical operations on Numpy arrays (10/17)

```
# year  hare    lynx    carrot
1900    30e3    4e3     48300
1901    47.2e3  6.1e3   48200
...
```

hare	Hase
lynx	Luchs

- `np.loadtext` allows to read such data into a matrix
- as we want to access the data column by column, we transpose the matrix, to make the columns to rows
- the four resulting rows can easily be assigned to four variables, giving a meaning to each

```
population = np.loadtxt('Math/population.tsv')
years, hares, lynxes, carrots = population.T
```

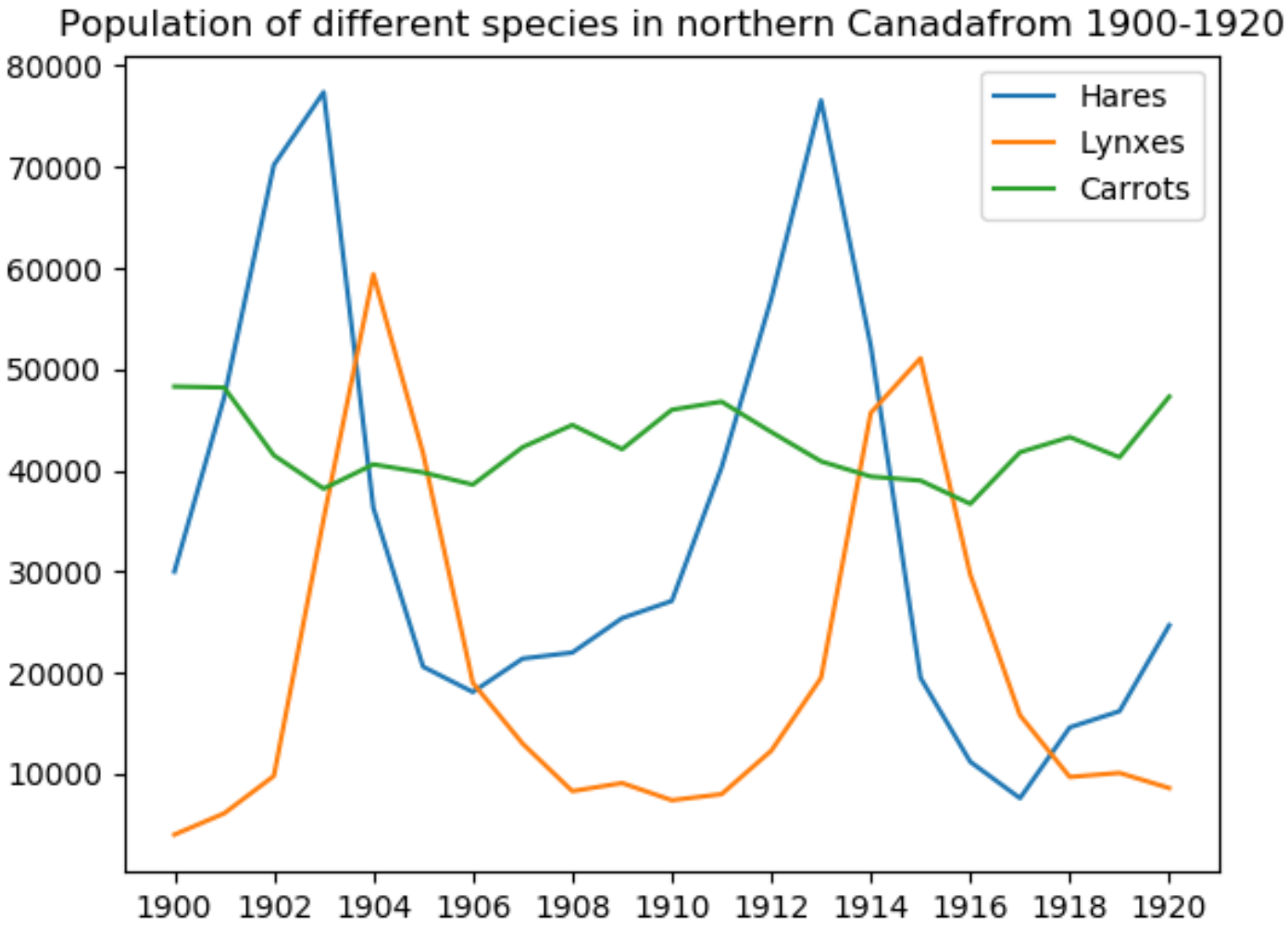
Numerical operations on Numpy arrays (11/17)

- to obtain an overview, we plot the data using `matplotlib`

```
import matplotlib.pyplot as plt
plt.switch_backend('agg')
fig, ax = plt.subplots()
ax.set_xticks(np.arange(min(years), max(years)+1, 2))
ax.plot(years, hares, years, lynxes, years, carrots)
ax.set_title(('Population of different species in northern Canada'
              'from {}-{}').format(int(min(years)),int(max(years)))
                  )
ax.legend(('Hares', 'Lynxes', 'Carrots'), loc='upper right')
fig.savefig('population.png')
```

- instead of calling `ax.plot` three times, we call it once with three pairs of 1-dim arrays

Numerical operations on Numpy arrays (12/17)



Numerical operations on Numpy arrays (13/17)

- we want to determine the mean, the median and the standard deviation of the population of each of the three species
- so we extract the submatrix consisting of the last three matrix columns
- and then determine these values over the years, i.e. along each of the three columns (`axes=0`), each with population counts for a species
- this gives three arrays, each of length 3 whose values are displayed in the for loop

```
species_pop = population[:,1:]
popmean = np.mean(species_pop,axis=0)  # arr of 3 floats, 1 per col
popmed = np.median(species_pop,axis=0) # arr of 3 floats. 1 per col
popstd = np.std(species_pop,axis=0)    # arr of 3 floats, 1 per col
print('# species\tpmean\tpmedian\tpstd')
for species,pmean,pmed,pstd in zip(['hares','lynxes','carrots'],
                                   popmean,popmed,popstd):
  print('{}\t{:.0f}\t{:.0f}\t{:.0f}'
        .format(species,pmean,pmed,pstd))
```

Numerical operations on Numpy arrays (14/17)

```
# species   pmean    pmedian   pstd
hares       34081   25400     20898
lynxes      20167   12300     16255
carrots     42400   41800     3323
```

- to simplify iterating over all matrix elements, one can flatten a matrix, using `np.ravel`

```
matrix = np.array([[1,2,3],[4,5,6]])
print('elements={}'.format(matrix.ravel()))
```

```
elements=[1 2 3 4 5 6]
```

Numerical operations on Numpy arrays (15/17)

- the inverse function to `np.ravel` is `np.reshape`, which turns a 1-dim array into a matrix whose shape is given

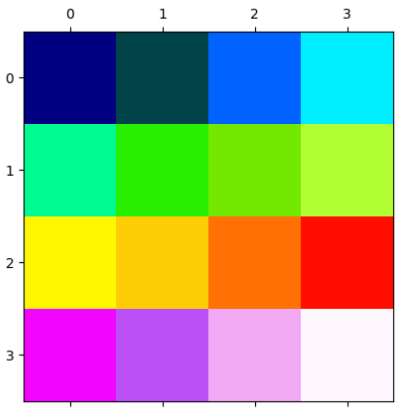
```
width = height = 4
arr = np.arange(width * height)
matrix = arr.reshape(width,height)
print('matrix=\n{}'.format(matrix))
```

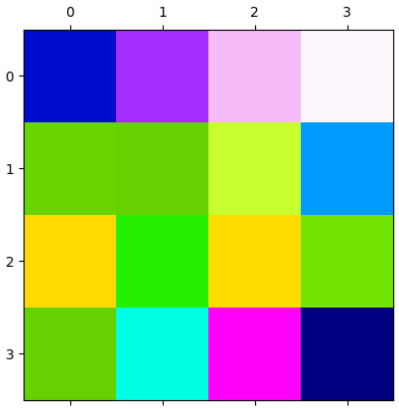
```
matrix=
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

- a common way to visualize the values of a matrix is to associate each value with a color from some spectrum and display the color in a grid superimposed on the matrix
- this can be done using `ax.matshow` where `ax` is an axes object

Numerical operations on Numpy arrays (16/17)

```
fig, ax = plt.subplots()
ax.matshow(matrix,cmap='gist_ncar')
fig.savefig('colorgrid.png')
fig, ax = plt.subplots()
ax.matshow(np.random.rand(4,4),cmap='gist_ncar')
fig.savefig('rcolorgrid.png')
```





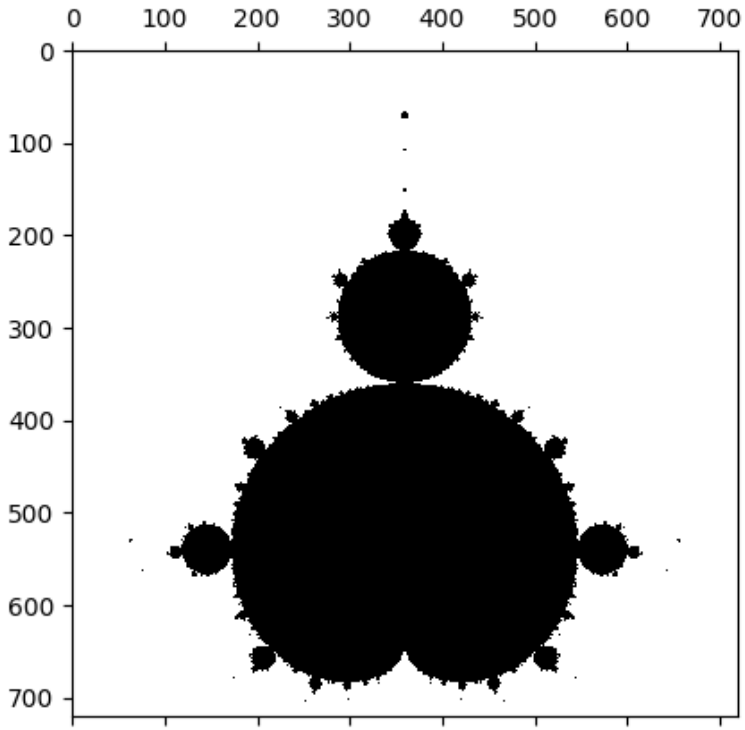
- from previous matrix with continuous values from 0 to 15: 0 for dark blue and 15 for white
- right: visualization of matrix of same size consisting of random values; some colors appear twice
- the `cmap` parameter specifies a color spectrum named `gist_ncar` (whatever that means) covering the colors of a rainbow

Numerical operations on Numpy arrays (17/17)

- as suggested by this example, we can create an image from a matrix of numeric values
- in fact, from a computational point of view, images can be considered as matrices with numeric values interpreted as colors
- the larger the matrix, the sharper the image
- we will consider another application in this direction and construct and visualize a famous mathematical set

A Numpy Application: Mandelbrot sets (1/12)

– this is a well known visualization of a specific mathematical set:



– it visualizes a Mandelbrot set, named after the french mathematician Benoit Mandelbrot, who first described these in 1980

– the visualization was created by a Python program explained later

– but first define Mandelbrot sets

A Numpy Application: Mandelbrot sets (2/12)

Definition

Consider a complex number c and the series z_0, z_1, z_2, ... of complex numbers, where

$$z_0 = 0$$
$$z_{n+1} = z_n \cdot z_n + c$$

The Mandelbrot set $\mathcal{M}$ is the set of all complex numbers c such that $\limsup_{n\to\infty} |z_{n+1}| \leq 2$.

- So: for a complex number c it holds $c \in \mathcal{M}$, iff the absolute values of the elements of the series z_0, z_1, z_2, ... depending on c are ≤ 2
- recall definitions of relevant operators for complex number $z = a + bi$:

$$\begin{aligned} z \cdot z &= (a+bi)\cdot(a+bi) = aa + abi + bia + bi \cdot bi \\ &= a^2 + b^2 i^2 + 2abi = a^2 - b^2 + 2abi \end{aligned} \quad \Bigg| \quad |z| = \sqrt{a^2 + b^2}$$

A Numpy Application: Mandelbrot sets (3/12)

– As $z_0 = 0 = 0 + 0i$, we get

$$z_1 = z_0 \cdot z_0 + c = (0 + 0i) \cdot (0 + 0i) + c = 0^2 - 0^2 - 2 \cdot 0 \cdot 0i + c = c$$

Example

We consider two complex numbers and show the relevant values:

$c = -1.5 + 1i$

n	z_n	$\lvert z_n \rvert$
1	$-1.5 + 1i$	1.8028
2	$-0.25 - 2i$	2.0156

$\lvert z_2 \rvert > 2 \Rightarrow c \notin \mathcal{M}$

$c = -1 + 0i$

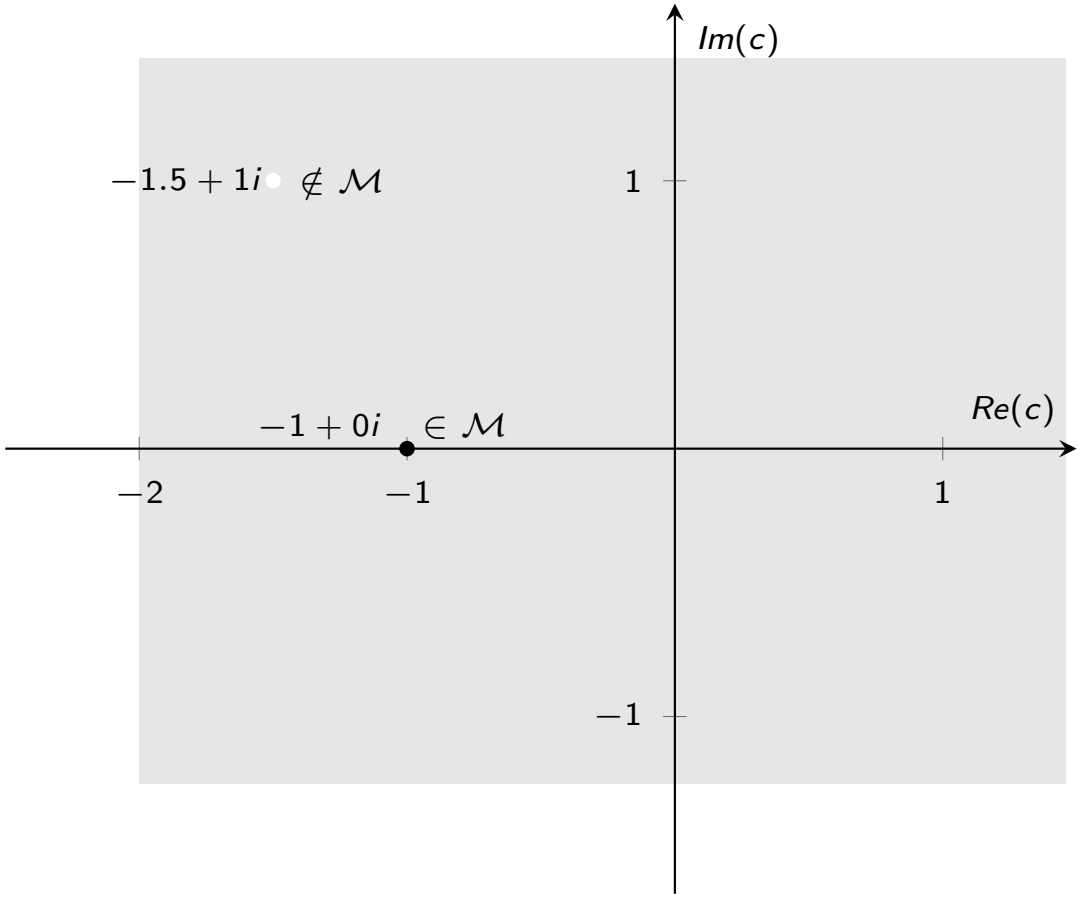
n	z_n	$\lvert z_n \rvert$
1	$-1 + 0i$	1.00
2	$0 + 0i$	0.00
3	$-1 + 0i$	1.00
4	$0 + 0i$	0.00

periodic and $\lvert z_n \rvert \leq 2 \Rightarrow c \in \mathcal{M}$

A Numpy Application: Mandelbrot sets (4/12)

- one approximates $\mathcal{M}$ by sampling a finite number of points c in complex space
- for each such sample point c
  - compute only up to a constant number of values $z_1, z_2, \ldots, z_{n_{\max}}$ for some constant $n_{\max}$, independent of c
  - if $|z_i| \leq 2$ for all i, $1 \leq i \leq n_{\max}$, then c belongs to $\widehat{\mathcal{M}}$
  - otherwise $|z_i| > 2$ for some i, $1 \leq i \leq n_{\max}$, and so c does not belong to $\widehat{\mathcal{M}}$.
- the larger $n_{\max}$, the better the approximation of $\mathcal{M}$ by $\widehat{\mathcal{M}}$
- for ease of notation, from now on, $\mathcal{M}$ denotes the approximated set $\widehat{\mathcal{M}}$.
- consider the two complex numbers of previous example in complex plane

A Numpy Application: Mandelbrot sets (5/12)



- slice of complex plane:
- on X-axes: real part $Re(c)$ of c
- on Y-axes: imaginary part $Im(c)$ of c
- grey part: space for sampling complex numbers for construction of $\mathcal{M}$
- sample many complex numbers (from ∞ many) and test if $\in \mathcal{M}$

- visualization: interpret complex number c as coordinate and draw point at this coordinate as black if $c \in \mathcal{M}$, otherwise draw it as white

$\Rightarrow$ image of mandelbrot set

A Numpy Application: Mandelbrot sets (6/12)

- to implement and visualize Mandelbrot sets in Python, we first implement a function to generate the series of z-values for some given complex number c and $n_{\max}$
- the function keeps track of the number of iterations and returns the minimal value $n < n_{max}$ such that $|z_n| > 2$
- if such a value does not exist, it returns n_{max}

```
def mandelbrot_iter(c,nmax):
  z = c
  for n in range(nmax):
    if abs(z) > 2:
      return n
    z = z * z + c
  return nmax
```

- the multiplication, additions and absolute values computed in this function are on complex numbers which are supported in Python
- so $c \in \mathcal{M}$ if and only if `mandelbrot_iter(c,nmax)== nmax`

A Numpy Application: Mandelbrot sets (7/12)

- we want to draw a Mandelbrot set as an image of a given width and height with a certain number of pixels $\Rightarrow$ discrete sample
- the boundaries are given by minimum and maximum values for the real and imaginary coordinates
- for this reason we first generate arrays `repart` and `impart` for the real and imaginary part of the complex numbers
- we combine all pairs of values from `repart` and `impart`, say at index `i` and `j`, respectively
- then we construct the corresponding complex number
- this requires to multiply imaginary part by the symbol `1j`
- we represent the pixels as entries of a `width` $\times$ `height`-matrix
- when creating a black and white image (`bw` is `True`):
  - for the complex number we test whether it is in $\mathcal{M}$, as described above
  - if the complex number is in $\mathcal{M}$, then we store the integer 1 in `matrix[i,j]`, otherwise we store 0

A Numpy Application: Mandelbrot sets (8/12)

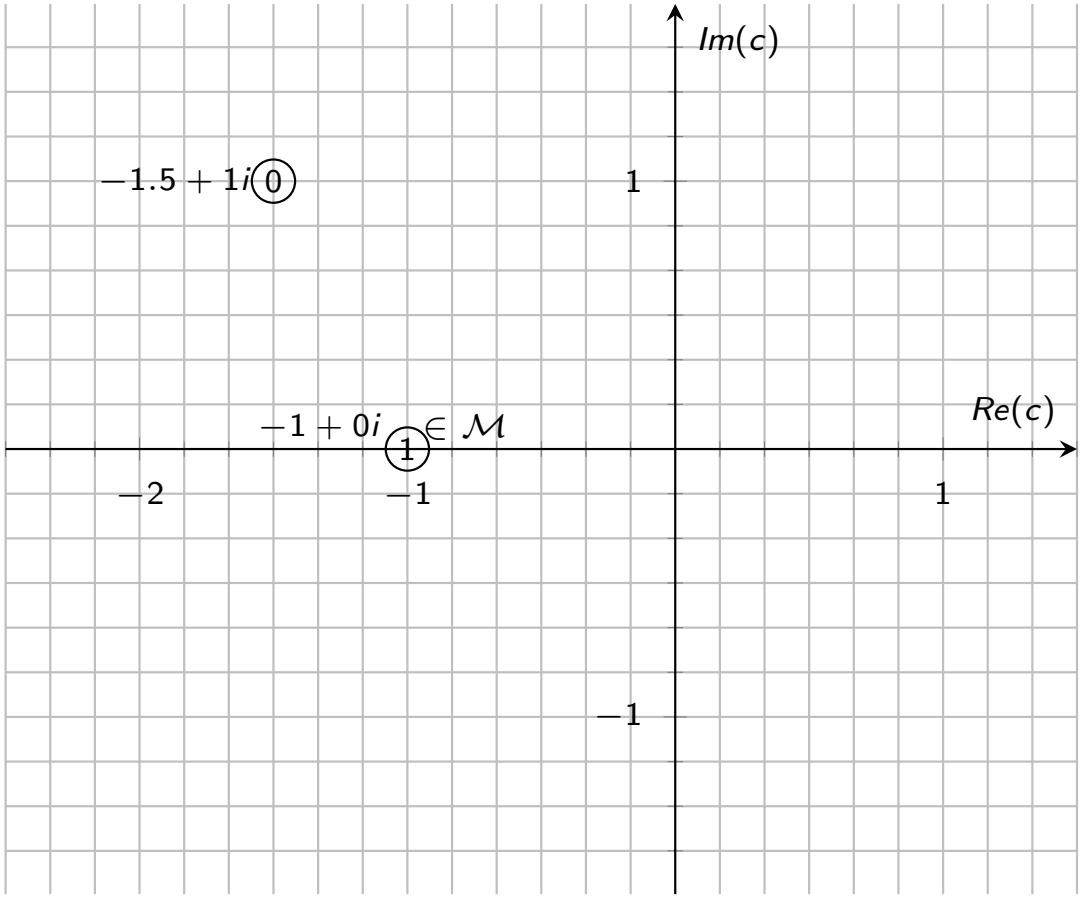
- when creating a colored image, we simply interpret the iteration value delivered by `mandelbrot_iter` as a color in some appropriate color map

```
def mandelbrot_set(bw,rmin,rmax,imin,imax,imgwidth,imgheight,nmax):
  repart = np.linspace(rmin, rmax, imgwidth)
  impart = np.linspace(imin, imax, imgheight)
  matrix = np.empty((imgwidth,imgheight),dtype=int)
  for i in range(imgwidth):
    for j in range(imgheight):
      c = repart[i] + impart[j] * 1j# imaginary const => complex n.
      if bw:
        matrix[i,j] = mandelbrot_iter(c,nmax) == nmax
      else: # colored
        miter = mandelbrot_iter(c,nmax)
        matrix[i,j] = 0 if miter == nmax else (miter + 1)
  return matrix
```

- so the matrix can be considered as a discrete grid of sampled values superimposed on the complex plane, see the following illustration
- the matrix values for the two complex numbers used in the previous example are circled (and shown for the bw-case):

A Numpy Application: Mandelbrot sets (9/12)

- $-1.5 + 1i \notin \mathcal{M} \Rightarrow$ `matrix[i,j]` $= 0$, where -1.5 corresponds to *column* i and 1 corresponds to row j of the grid



- $-1 + 0i \in \mathcal{M} \Rightarrow$ `matrix[i,j]` $= 1$, where -1 corresponds to *column* i and 0 corresponds to row j of the grid
- note that here we use i as the imaginary constant and `i` as index

A Numpy Application: Mandelbrot sets (10/12)

- once we have the matrix, we can use the `matshow`-method from `matplotlib` to visualize it
- this is performed in a function `mandelbrot_image` to which we provide minimum and maximum values for the real and imaginary parts, respectively, of the complex numbers of the sample
- we also provide a height and a width of the image which are multiplied by a constant `dpi` (dots per inch) to obtain the number of sample values on both dimensions, the image height and image width
- finally we provide the `nmax` parameter

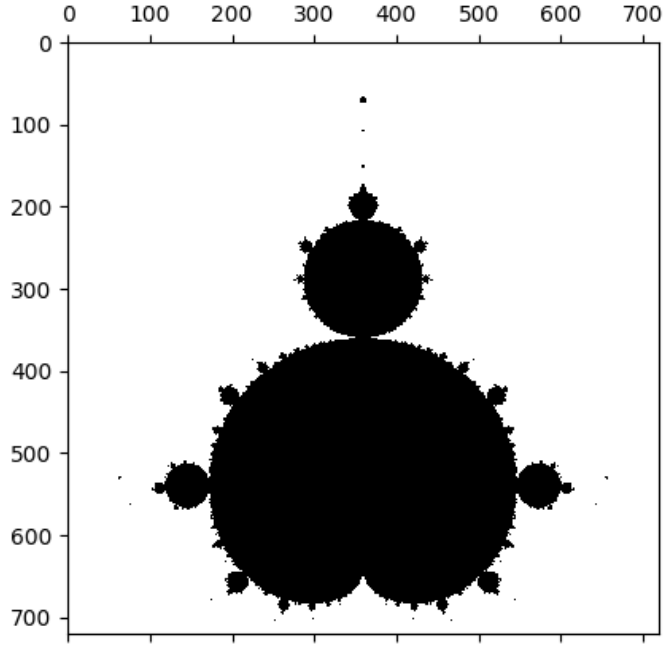
A Numpy Application: Mandelbrot sets (11/12)

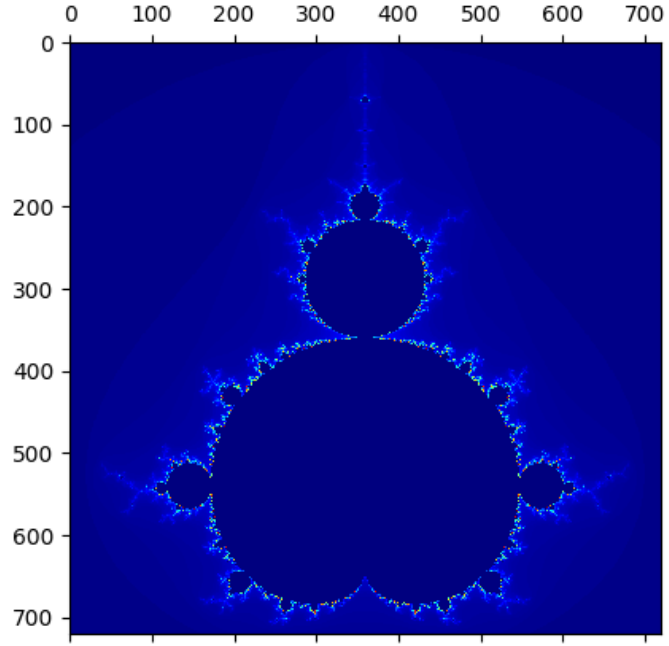
```
from matplotlib import pyplot as plt
plt.switch_backend('agg') # disable X-server
# needed on Linux when running program via remote ssh login

def mandelbrot_image(bw,rmin,rmax,imin,imax,nmax,width,height):
  dpi = 72 # dots per inch
  imgwidth = dpi * width
  imgheight = dpi * height
  matrix = mandelbrot_set(bw,rmin,rmax,imin,imax,
                          imgwidth,imgheight,nmax)
  this_cmap = 'gist_yarg' if bw else 'jet'
  fig, ax = plt.subplots()
  ax.matshow(matrix,cmap=this_cmap)
  fig.savefig('mandelbrot-{}.png'.format('bw' if bw else 'color'))

mandelbrot_image(False,-2.0,0.5,-1.25,1.25,256,10,10)
```

- note that we use two different colormaps which turn the matrix value into colors: `gist_yarg` for a bw-image, and `jet` for the colored image
- here are the two images obtained

A Numpy Application: Mandelbrot sets (12/12)

– the ticks on the axes refer to the indexes of the matrix in the corresponding dimension





– the web provides a lot of material on Mandelbrot sets, e.g.

`https://www.youtube.com/watch?v=2JUAojvFpCo`

Classification and ROC-curves (1/20)

- classification of objects (any kind of items of interest, e.g. sequences, customers, atoms, molecules ..) means to group them into different disjoint classes
- each object gets a label, usually 0 and 1 for binary classification, or green and red in our example below
- classification is often difficult and one cannot always uniquely assign an object to one of the possible classes
- so, often a classification method delivers a score for each object
- in binary classification one then defines a threshold:
  - score $<$ threshold $\Rightarrow$ assign object to class 0
  - score $\geq$ threshold $\Rightarrow$ assign object to class 1
- ideal case: score leads to assignment corresponding to the real class

this section is inspired by the Blog *Receive Operating Characteristic Curves Demystified (in Python)* by Syed Sadat Nazrul

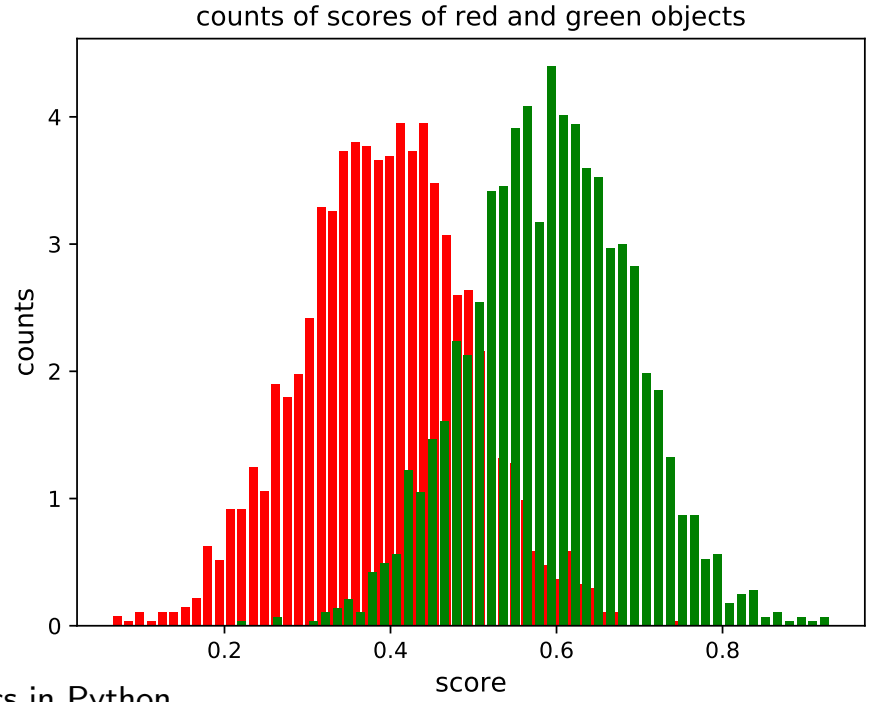
`https://towardsdatascience.com/receiver-operating-characteristic-curves-demystified-in-python-bd531a4364d0`

Classification and ROC-curves (2/20)

- suppose we have a training set of 2 000 red and 2 000 green objects and our classification method computes a score for each object
- count c of score range x $\Rightarrow$ bar at x of height c
- compute the distribution of the scores for the red and green objects and plot a corresponding histogram:

```
ax.bar(scores,counts,width=0.01,color=color)
```

- where `scores` is the vector of score ranges and `counts` is the vector of counts for each such score range



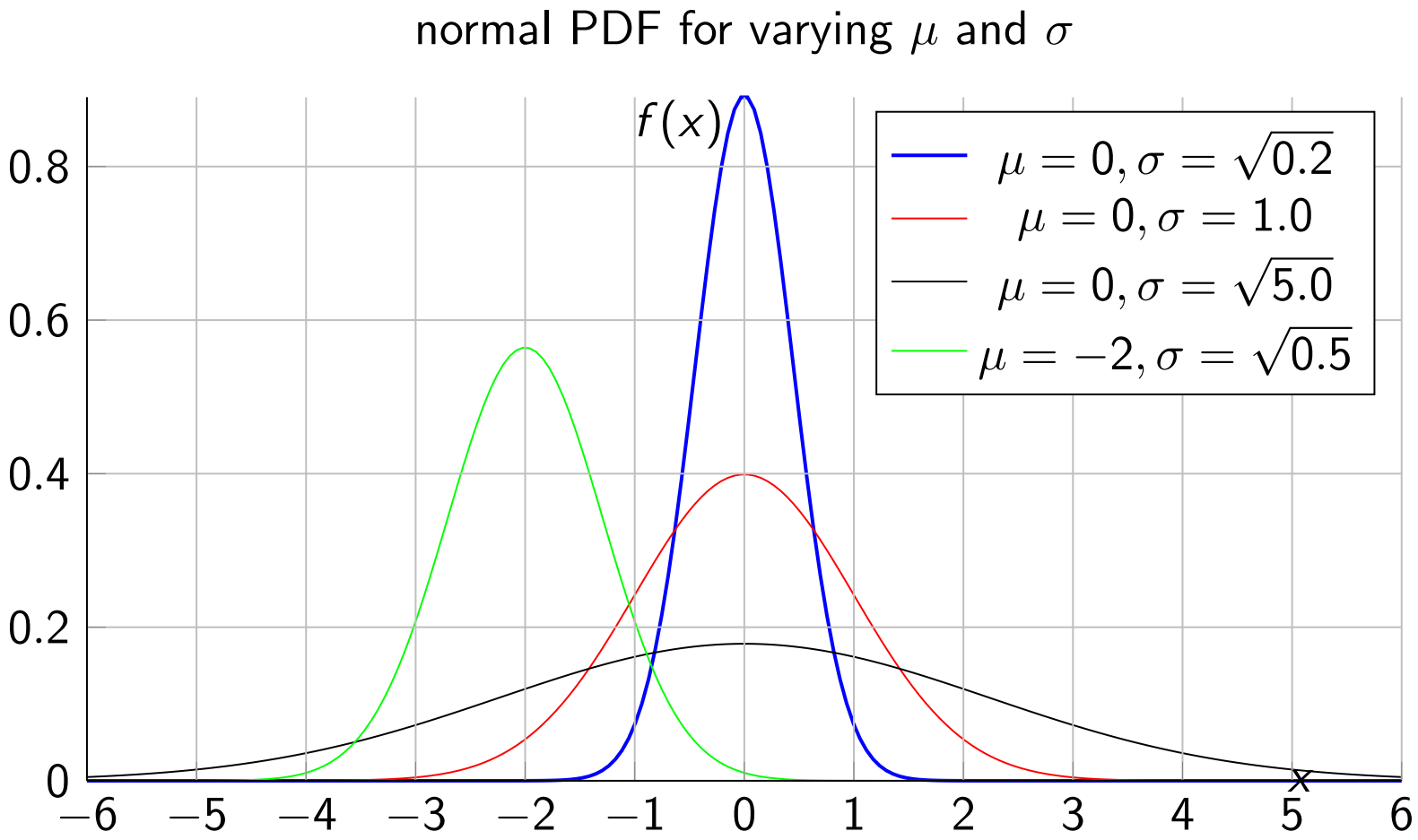
Classification and ROC-curves (3/20)

- in the region where the two histograms overlap, the assignment to one of the two classes (red or green) is not always uniquely possible
- whichever threshold for the score we use, some objects will not be correctly assigned to one of the two classes
- to develop a method quantifying classification performance (i.e. the ability to separate classes), one often turns the discrete distribution into a continuous function, the probability density function (PDF)
- so the counts per bin are approximated by a smooth function
- in our case, the discrete distributions were randomly generated according to a normal distribution (also called gaussian distribution) with standard deviation $\sigma = 0.5$ and mean $\mu = 0.4$ (red) and $\mu = 0.6$ (green)
- the PDF of a normal distribution with known μ and σ is the function

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3}$$

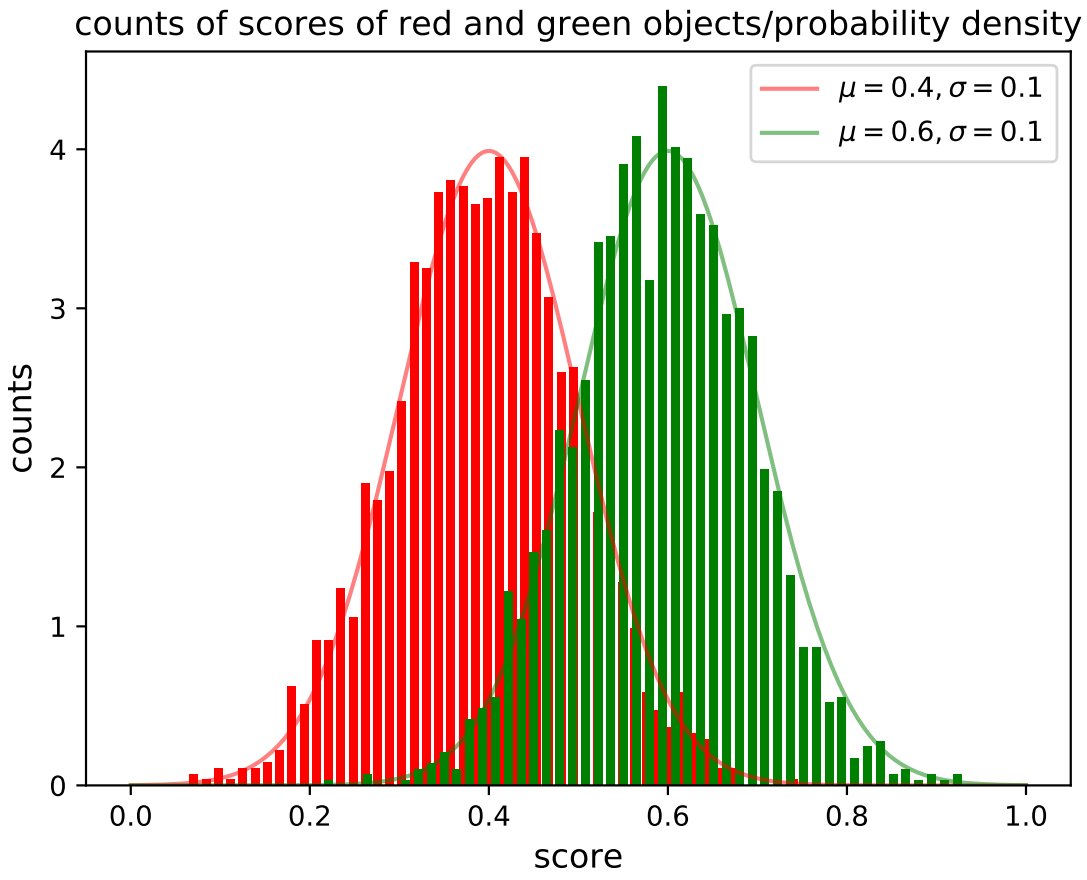
Classification and ROC-curves (4/20)

– the following figure shows how the values of μ and σ shape the plot of the probability density function of the normal distribution



Classification and ROC-curves (5/20)

– here is the plot of the previous discrete distribution augmented with the corresponding PDF in the same colors



Classification and ROC-curves (6/20)

- to generate such a plot of the PDF we first implement a function corresponding to Equation (3):

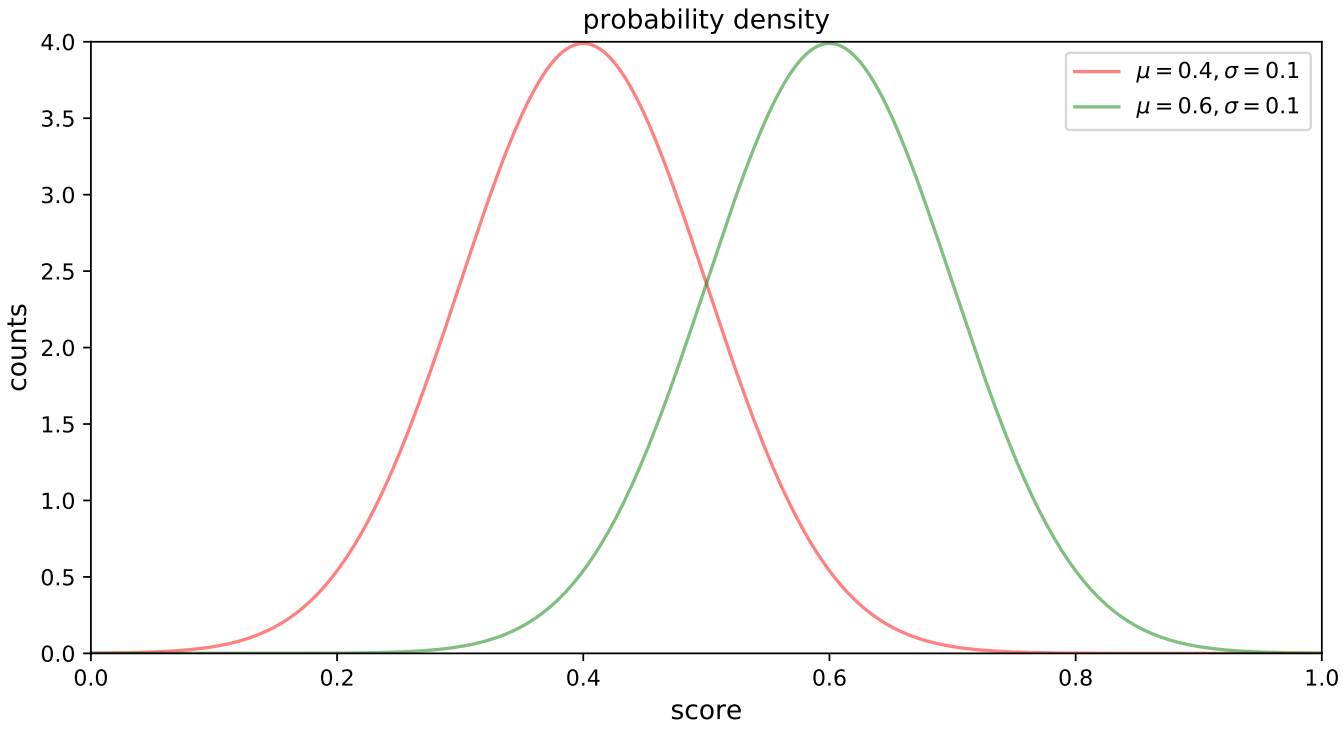
```
def normalProbabilityDensity(mean,stddev,x):
  stddev_sq = stddev ** 2
  const = 1.0 / np.sqrt(2 * np.pi * stddev_sq)
  return const * np.exp(-((x - mean) ** 2)/(2.0 * stddev_sq))
```

- as we use `numpy`-functions, the parameter `x` can be a single value or an array of values, as used in the following code (where `color` is `'r'` or `'g'` for red or green, respectively)

```
x_vec = np.linspace(0, 1, num=num_points)
y_vec = normalProbabilityDensity(x_vec,mean,stddev)
ax.plot(x_vec, y_vec,color,alpha=0.5,
        label='$\mu={}, \sigma={}$'.format(mean,stddev))
```

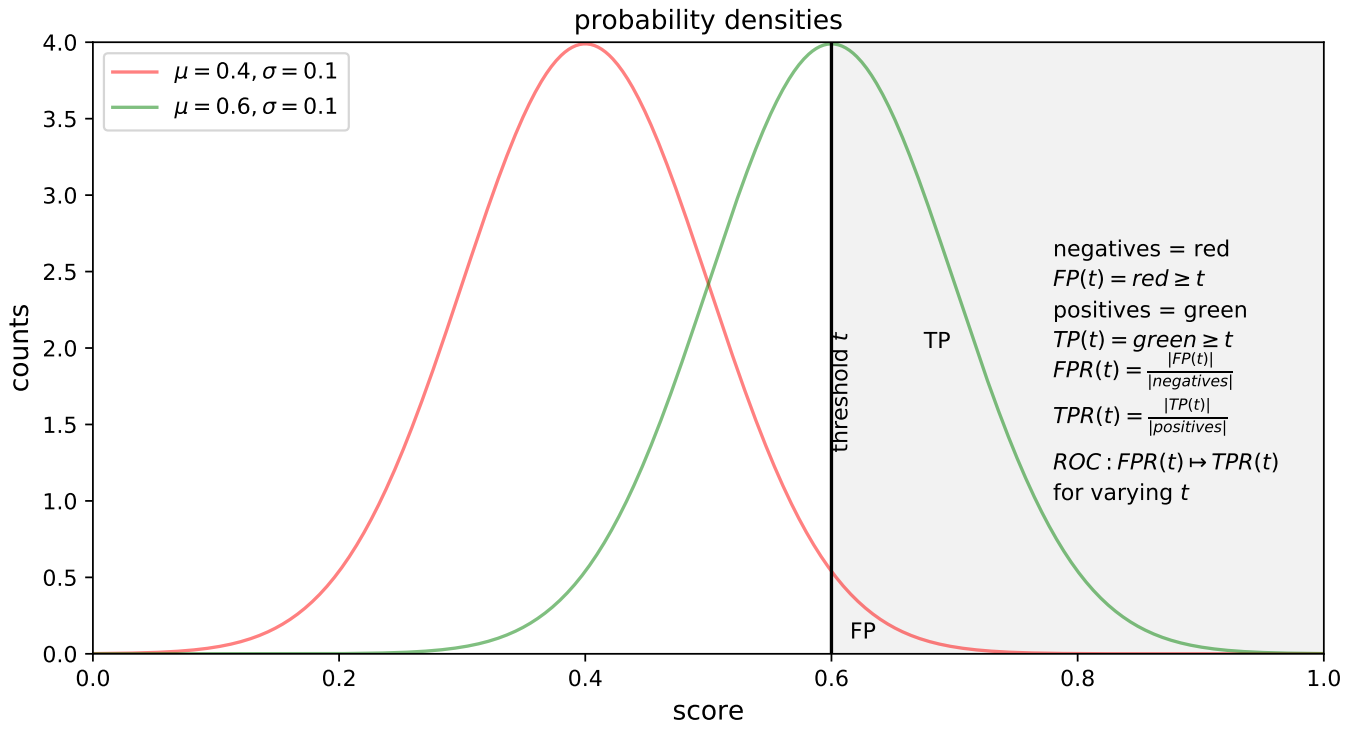
Classification and ROC-curves (7/20)

- omitting the original discrete distributions gives a less cluttered plot showing the relevant information
- always keep in mind, that the integral of the functions (in a certain interval) represents the counts of the distribution



Classification and ROC-curves (8/20)

- we need a threshold to turn the score of an object into class membership
- threshold is depicted as vertical line; region right of this line is grey
- we used `matplotlib`-commands `ax.text`, `ax.axvline` and `ax.axvspan` to obtain the annotation of the plot

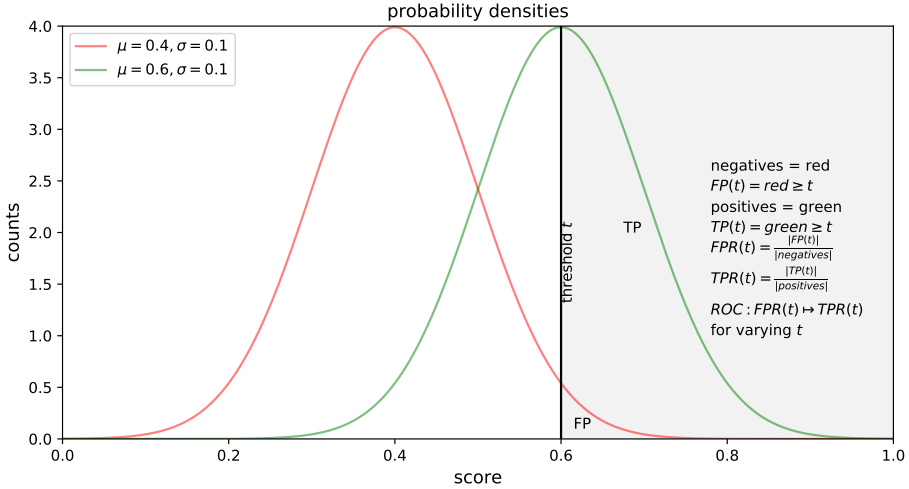


Classification and ROC-curves (9/20)

- all elements with score $\geq t$ are considered to be in the green class, all others in the red class
- in this context, one often denotes the two classes of objects as *negatives* (for red) and *positives* (for green).
- the objects with scores in the grey area below the red curve are termed false positives (FP), as these are falsely classified as positives (green) for the given threshold t
- the objects with scores in the grey area below the green curve are termed true positives (TP), as these are truly classified as positives (green) for the given threshold t
- the ratio $\frac{|FP(t)|}{|negatives|}$ of the number of false positives and all negatives is the false positive rate, denoted $FPR(t)$
- the ratio $\frac{|TP(t)|}{|positives|}$ of the number of true positives and all positives is the true positive rate, denoted $TPR(t)$

Classification and ROC-curves (10/20)

- if we increase the value of t, $FP(t)$ and $FPR(t)$ as well as $TP(t)$ and $TPR(t)$ decreases
- decreasing the value of t would lead to the opposite result



- so with a varying t we obtain pairs $(FPR(t), TPR(t))$ and these can be considered as a function termed *receiver operator characteristic* (ROC, for short).
- of course we want to plot this function to obtain a ROC-curve
- we next consider how to do this, starting with a class representing a normal distribution

Classification and ROC-curves (11/20)

```
class NormalDist:
  def __init__(self,num_points,mean,stddev,color):
    self._mean = mean
    self._stddev = stddev
    self._color = color
    self._x_vec = np.linspace(0, 1, num=num_points)
    self._y_vec = normalProbabilityDensity(mean,stddev,self._x_vec)
    self._tail = list()
    tail_sum = 0
    for y in reversed(self._y_vec):
      tail_sum += y
      self._tail.append(tail_sum)
  def domain(self):
    return self._x_vec
  def __getitem__(self,idx):
    assert idx >= 0 and idx < len(self._y_vec)
    return self._y_vec[idx]
  def tail(self):
    return iter(self._tail)
  def sum(self):
    return np.sum(self._y_vec)
  def plot(self,ax):
    ax.plot(self._x_vec, self._y_vec,self._color,alpha=0.5,
            label='$\mu={}, \sigma={}$'.format(self._mean,self._stddev))
    ax.legend()
```

Classification and ROC-curves (12/20)

- the class is fairly standard keeping track of the mean and standard deviation of the normal distribution
- besides these, it maintains the color, as we refer to a distribution by its color
- besides the arrays of x- and y-values there is an extra array `tail` s.t. $tail[i]$ is the sum of all y-values in the entries at an index $j \geq i$
- we access the elements of the array via an iterator `tail()`
- to access a y-value, we overload the operator `[]` by declaring the method `__getitem__`
- for a given `matplotlib`-axes `ax`, the method `plot` creates a plot of the x- versus the y-values appropriately colored and supplemented with a legend in LaTeX-format.

Classification and ROC-curves (13/20)

- the following function plots the two PDFs in one figure

```
def plot_pdfs(d_red,d_green,ax):
  ax.set_title('probability densities',size = 12)
  ax.set_ylabel('counts', size = 12)
  ax.set_xlabel('score', size = 12)
  ax.set_xlim(0,1)
  ax.set_ylim(0,4)
  d_red.plot(ax)
  d_green.plot(ax)
```

- we use it here for creating the plots we have seen previously

```
num_points = 1000
stddev = 0.1
d_red = NormalDist(num_points,0.4,stddev,'r')
d_green = NormalDist(num_points,0.6,stddev,'g')

fig, ax = plt.subplots(figsize=(10, 5))
plot_pdfs(d_red,d_green,ax)
```

Classification and ROC-curves (14/20)

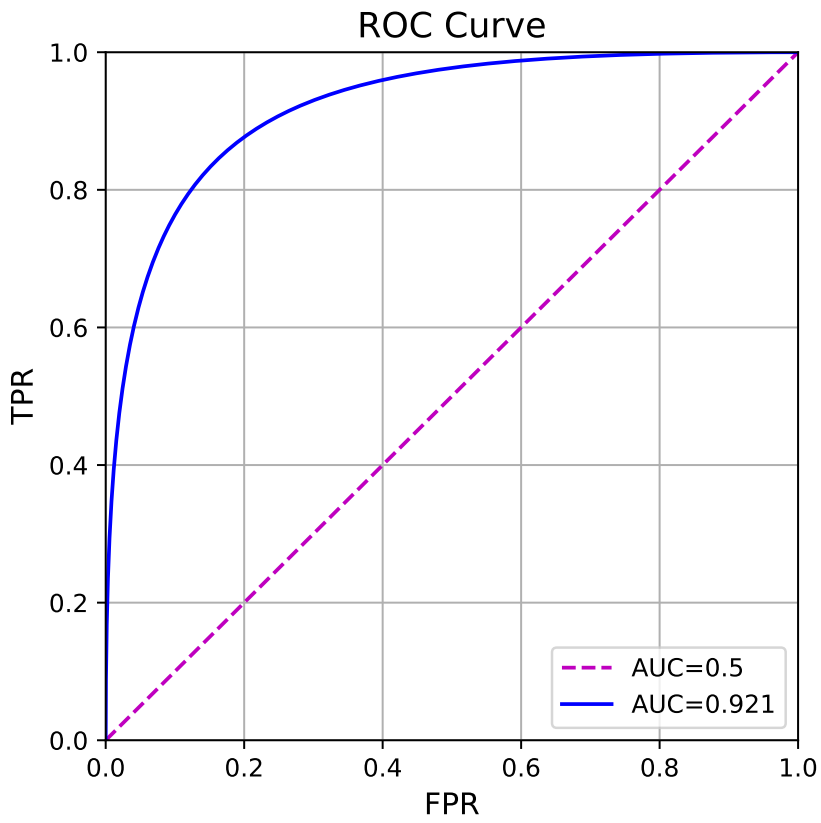
- the following function collects the data for a ROC curve for the given normal distributions on the given axes (result see next frame)

```
def roc_curve_collect(d_red, d_green):
  tot_d_red = d_red.sum()
  tot_d_green = d_green.sum()
  FPRs = [v/tot_d_red for v in d_red.tail()]
  TPRs = [v/tot_d_green for v in d_green.tail()]
  auc = integral_pointlist(FPRs,TPRs)
  return FPRs, TPRs, auc

def roc_curve_plot(d_red, d_green, ax):
  x = d_green.domain()
  FPRs, TPRs, auc = roc_collect(d_red, d_green)
  ax.plot(FPRs,TPRs,'b-')
  ax.plot(x,x, 'm--')
  ax.set_xlim([0,1])
  ax.set_ylim([0,1])
  ax.set_title('ROC Curve', fontsize=14)
  ax.set_ylabel('TPR', fontsize=12)
  ax.set_xlabel('FPR', fontsize=12)
  ax.grid()
  ax.legend(['AUC=0.5','AUC={:.3}'.format(auc)])
```

- the lists of false positive rates and true positive rates are computed by dividing the values of the tail-array by the corresponding total values of the distribution
- the AUC-value is computed by the func. `integral_pointlist`
- the two returned list are plotted where `FPRs` contains the x-values and `TPRs` the y-values
- all values are in the range $[0, 1]$ and so the corresponding limits are set
- the identity function $x \mapsto x$ (45-degree line) is plotted in magenta

Classification and ROC-curves (15/20)

- blue: ROC curve mapping $FPR(t)$ to $TPR(t)$
- magenta: worst possible ROC curve
  - occurs when distribution of scores for green and red objects is identical ($\Rightarrow$ no separation of classes possible)
- best possible ROC curve: constant function 1



- the classification performance is measured by the *area under the ROC curve* (AUC) which ranges from 0.5 (worst case) to 1 (best case)
- in our case the AUC is 0.921, a fairly good value

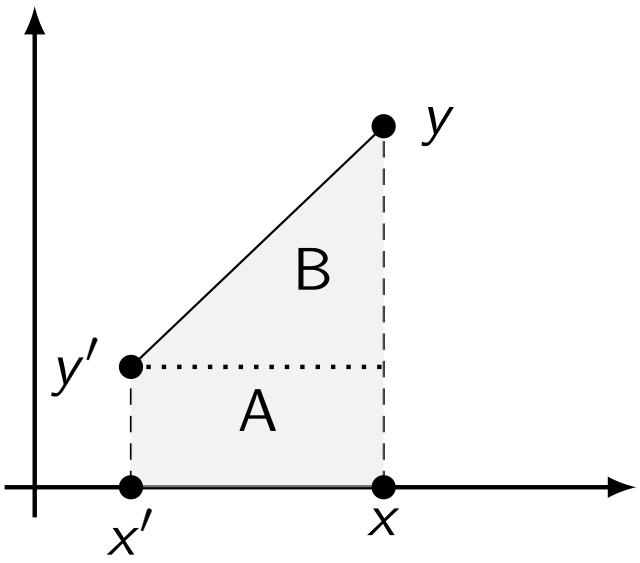
Classification and ROC-curves (16/20)

- it remains to show the implementation of the function `integral_pointlist`
- the idea is to create a pair for each of the corresponding FPR and TPR values and to sort these pair by the FPR value
- two consecutive pairs (x', y') and (x, y) then represent an interval of width $x - x'$ whose area is $\frac{1}{2}(x - x')(y + y')$ (see next frame)
- so one only needs to accumulate these areas for all consecutive pairs:

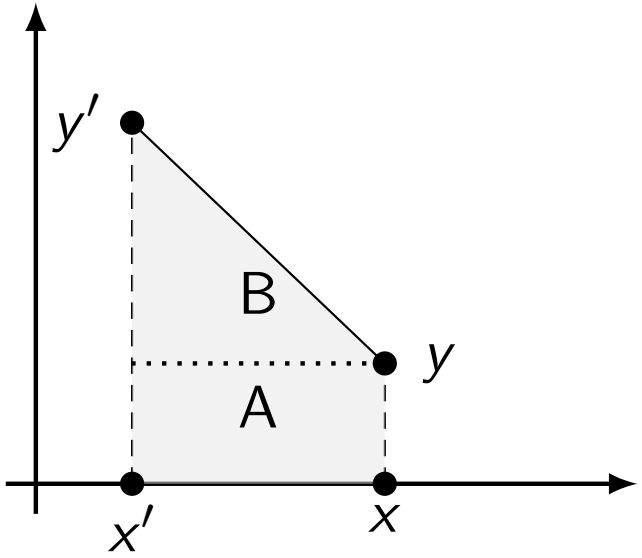
```
def integral_pointlist(fprs,tprs):
  assert len(fprs) == len(tprs)
  pairs = sorted(zip(fprs,tprs))
  area = 0
  prev_x, prev_y = pairs[0]
  for x, y in pairs[1:]:
    assert prev_x <= x
    area += 0.5 * (x - prev_x) * (prev_y + y)
    prev_x, prev_y = x, y
  return area
```

- `integral_pointlist` could be replaced by the library function `sklearn.metrics.auc`

Classification and ROC-curves (17/20)



$$
\begin{aligned}
A + B &= (x - x')y' + \frac{(x - x')(y - y')}{2} \\
&= \frac{2(x - x')y' + (x - x')y - (x - x')y'}{2} \\
&= \frac{(x - x')y' + (x - x')y}{2} \\
&= \frac{1}{2}(x - x')(y + y')
\end{aligned}
$$

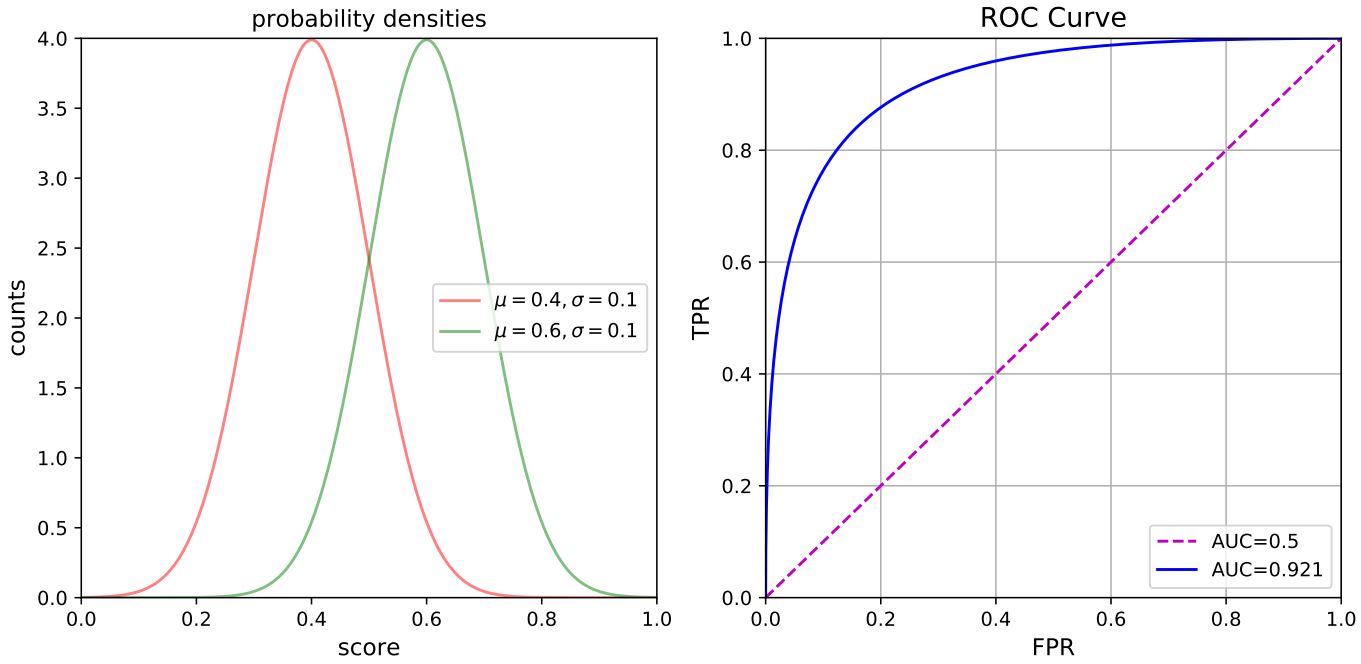


$$
\begin{aligned}
A + B &= (x - x')y + \frac{(x - x')(y' - y)}{2} \\
&= \frac{2(x - x')y + (x - x')y' - (x - x')y}{2} \\
&= \frac{(x - x')y + (x - x')y'}{2} \\
&= \frac{1}{2}(x - x')(y + y')
\end{aligned}
$$

Classification and ROC-curves (18/20)

– let's plot the PDF and the ROC curve next to each other for visual comparison:

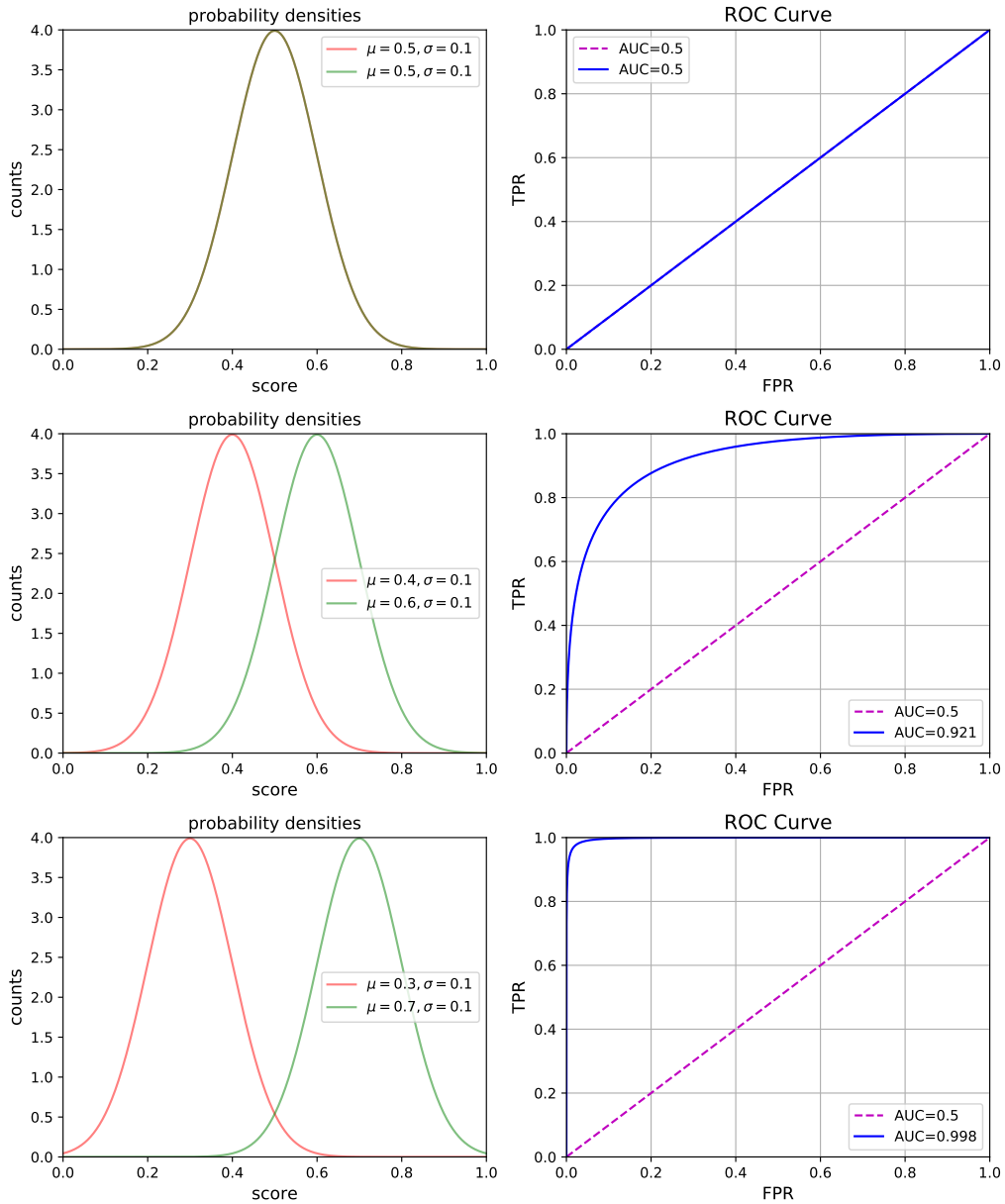
```
fig, ax = plt.subplots(1,2, figsize=(10,5))
plot_pdfs(d_red, d_green, ax[0])
plot_roc(d_red, d_green, ax[1])
fig.tight_layout()
fig.savefig('normal_plot_roc.pdf')
```



Classification and ROC-curves (19/20)

- next consider how the ROC curve (and the AUC) changes as the class separation (i.e. the model performance) improves
- we force the change by altering the mean value of the normal distributions

μ		AUC
0.5	0.5	0.500
0.4	0.6	0.921
0.3	0.7	0.998



Classification and ROC-curves (20/20)

- these plots were created by the following commands
- here we exploit that the `subplot`-method of `matplotlib` allows to specify the number of rows (i.e. 3) and columns (i.e. 2) we want to fill with the subplots

```
fig, ax = plt.subplots(3,2, figsize=(10,12))
means_tuples = [(0.5,0.5),(0.4,0.6),(0.3,0.7)]
for row, (d_red_mean, d_green_mean) in enumerate(means_tuples):
    d_red = NormalDist(num_points,d_red_mean,stddev,'r')
    d_green = NormalDist(num_points,d_green_mean,stddev,'g')
    plot_pdfs(d_red, d_green, ax[row,0])
    plot_roc(d_red, d_green, ax[row,1])
fig.tight_layout()
```

final remark

- while our examples of ROC curves were derived from normal distributions, the concept of operator receiver characteristics is not restricted to this kind of distributions

Numpy and Polynomials (1/2)

- numpy provides the method `poly1d` to specify the coefficients of a polynomial, and thus the polynomials itself
- for example, the polynomial $3x^2 + 2x - 1$ is specified as follows:

```
polynomial = np.poly1d([3, 2, -1])
```

- one can determine the order of a polynomial (using the method `order`) or even solve the polynomial for 0 (using the method `roots`)

```
print('polynomial order: {}'.format(polynomial.order))
print('solution of polynomial = 0: {}'.format(polynomial.roots))
```

```
polynomial order: 2
solution of polynomial = 0: [-1.          0.33333333]
```

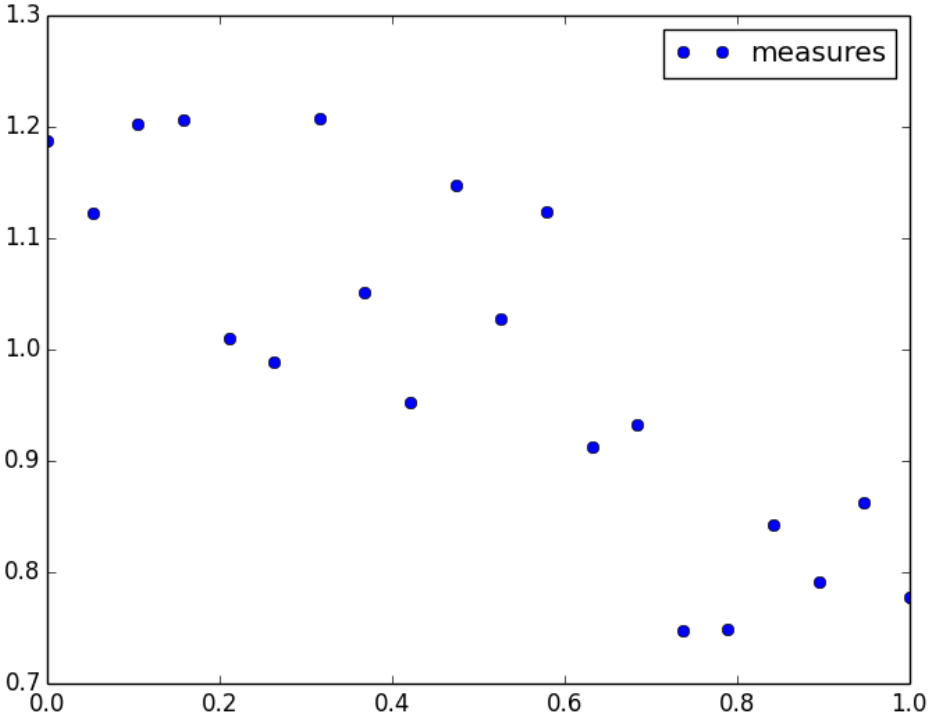
- indeed the two solutions are correct as

Numpy and Polynomials (2/2)

$$
\begin{aligned}
x = -1 \Rightarrow 3x^2 + 2x - 1 &= 3(-1)^2 + 2(-1) - 1 \\
&= 3 - 2 - 1 \\
&= 0 \\
x = \frac{1}{3} \Rightarrow 3x^2 + 2x - 1 &= 3\left(\frac{1}{3}\right)^2 + 2\frac{1}{3} - 1 \\
&= 3\frac{1}{9} + \frac{2}{3} - 1 \\
&= \frac{1}{3} + \frac{2}{3} - 1 \\
&= 0
\end{aligned}
$$

Numpy and Curve fitting (1/3)

- as the last example of applying numpy we consider the problem of fitting a polynomial (of known order) to a set of points in space, called measures
- here is a plot of the measures



Numpy and Curve fitting (2/3)

- we want to find a polynomial of order 3 which fits these measures well
- this is the fitting problem
- suppose we have the X- and corresponding Y-values describing the measures in two `numpy`-arrays `x` and `y` of the same length
- then we can simply use the method `np.polyfit` as follows:

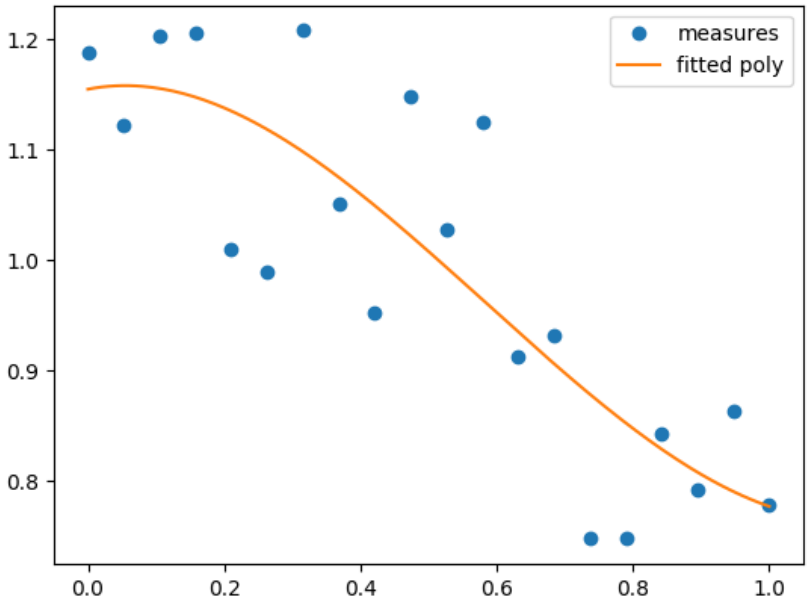
```
pdegree = 3
termlist = list()
fitted = np.polyfit(x, y, pdegree)
for idx, f in enumerate(fitted):
  termlist.append('{:.2f} x^{}'.format(f,len(fitted)-idx-1))
print('fitted coefficients: {}'.format(fitted))
print('fitted polynom:        {}'.format(' + '.join(termlist)))
```

- `np.polyfit` employs the least-square method for determining the coefficients of the polynomial
- the previous code prints the corresponding polynomial as follows:

Numpy and Curve fitting (3/3)

```
fitted coefficients: [ 0.66274499 -1.16202901  0.12140909  1.15461701]
fitted polynom:        0.66 x^3 + -1.16 x^2 + 0.12 x^1 + 1.15 x^0
```

- a plot of the measures and the polynomial $0.66x^3 + -1.16x^2 + 0.12x^1 + 1.15x^0$ shows that the latter fits the former well



synopsis: methods using numpy arrays

`m.dtype`	obtain base type
`m.ndim`	obtain number of dimensions
`m.shape`	obtain tuple of array dimensions
`len(m)`	obtain size of first dimension of array
`m.resize((r,c))`	resize array `m` to hold `r` rows and `c` columns; previous values are maintained; additional elements are initialized to 0; if `m` is a 1-dim array, then omit `c`
`a + 1`	add 1 to each entry of array `a`
`2 ** a`	use the values from array `a` as exponents of 2 and create corresponding list which has same length as `a`
`mat1 + mat2`	add corresponding elements from `mat1` and `mat2`, provided both matrices have the same number of rows and columns; return result of addition; works for `-` and `*` in analogous way
`mat1.dot(mat2)`	return product of matrix `mat1` and matrix `mat2`
`mat.T`	return transposed version of matrix `mat`
`sum(mat,axis=i)`	return sum of matrix-values on axis `i`; `i=1` for rows; `i=0` for columns
`max(arr)`	return maximum value of array `arr`
`mean(arr)`	return mean value of elements in array `arr`
`median(arr)`	return median value of elements in array `arr`
`std(arr)`	return standard deviation of elements in array `arr`

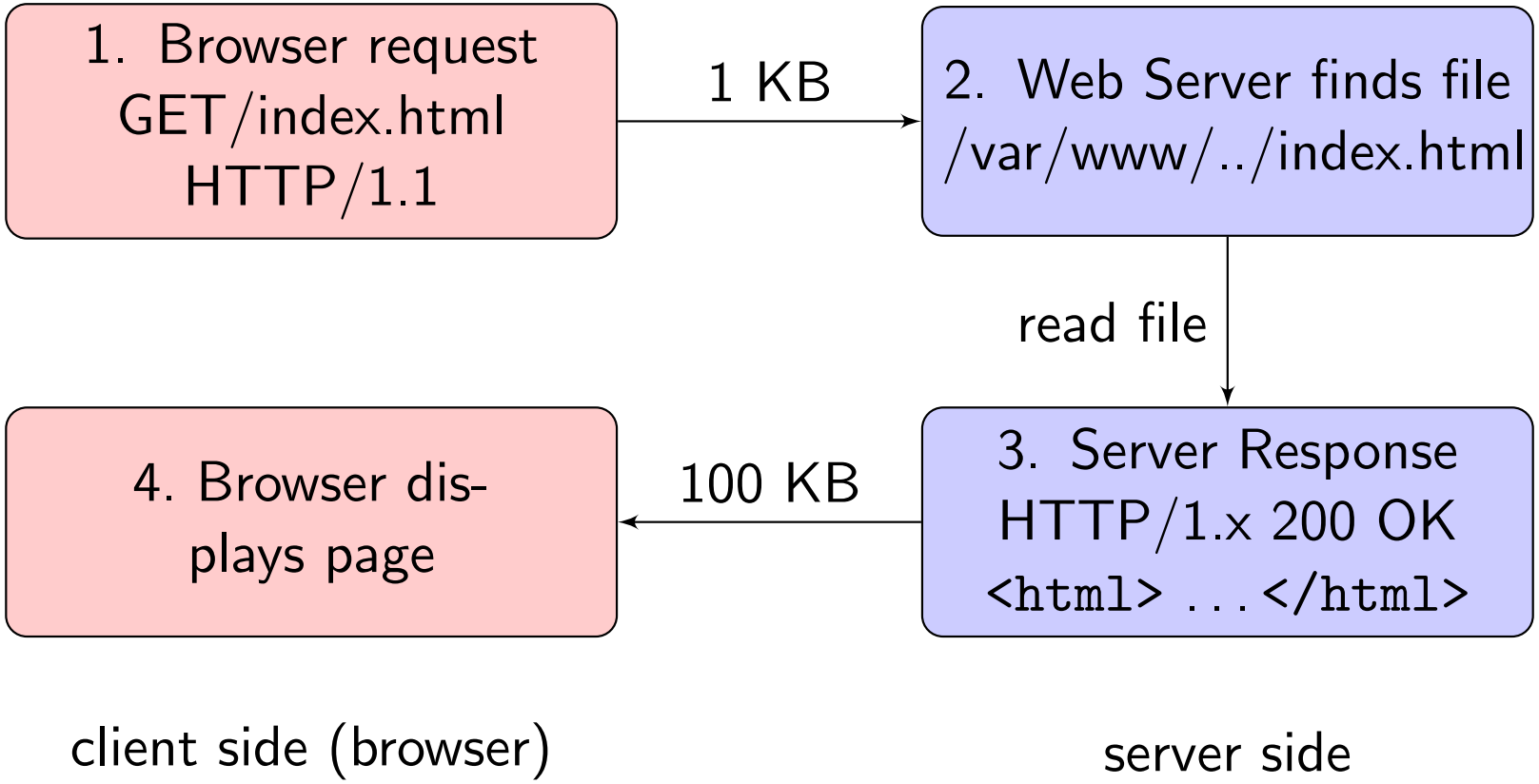
Web programming with python and flask (1/4)

- Python and other modern scripting languages are well equipped with techniques relevant for solving tasks related to the WWW
- these techniques allow building websites, web services, and web applications
- main task is to handle HTTP-requests generated by the browser and communicate these to a web server which delivers the results to be displayed
- HTTP = Hypertext Transfer Protocol
- here is the typical data flow in a HTTP request and response

this chapter is based on code and frame content prepared by Fabian Hausmann

Web programming with python and flask (2/4)

A HTTP request and response



Web programming with python and flask (3/4)

- Python has all techniques directly build in to handle HTTP requests and responses
- they are not so easy to use

⇒ but there are several frameworks that have been developed on top of Python to simplify these tasks

- we will use Flask, a web framework written for and in python
  - minimalistic approach to handle HTTP requests and deliver responses to clients (usually the browser)
- Flask includes a micro webserver, so no extra installations necessary
- more information on `http://flask.pocoo.org/`

Web programming with python and flask (4/4)

- installation (with administrator-permission):

```
pip3 install flask
```

- run an application[10]

```
FLASK_APP=</path/to/your/application.py> flask run
```

  and open `http://127.0.0.1:5000/` in your web browser

- path following `FLASK_APP=` can be relative or absolute

[10] At the ZBH flask is installed under: /usr/local/zbhtools/anaconda/bin/flask

Flask and Hello World! (1/2)

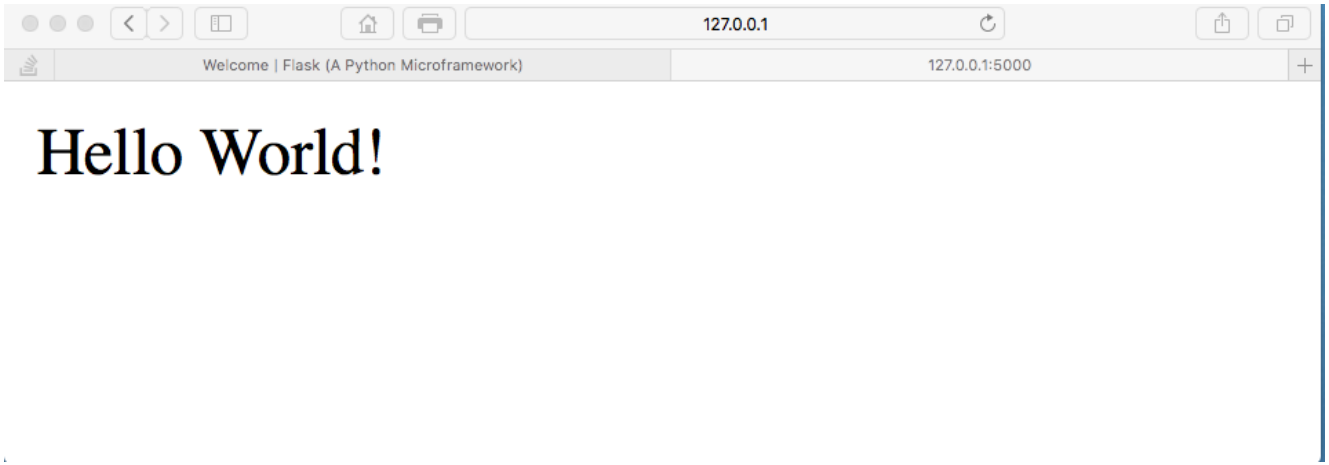
- here is a minimal python script `hello.py` running an app in flask:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
  return 'Hello World!'
```

- start with the appropriate import statement
- set up a web application with the current name
- specify a route, i.e. an execution point of the function defined next
- in our case we use the route /
- this specifies that the function following the route is executed once `http://127.0.0.1:5000/` is loaded in a web browser

Flask and Hello World! (2/2)

```
$ FLASK_APP=./hello.py flask run
 * Serving Flask app "./hello.py"
 * Environment: production
   WARNING: Do not use development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

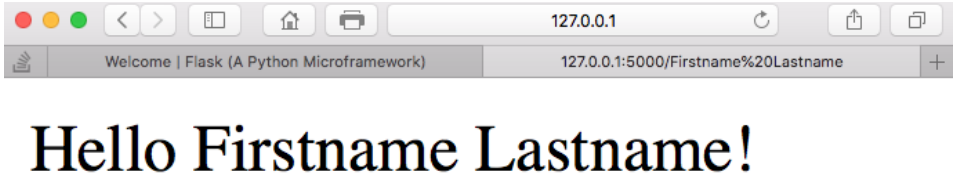


Hello World! Advanced (1/2)

- one can specify a more general route including a variable, e.g. `name`
- the variable is instantiated by the suffix of the URL after the last `/`
- that is, the URL `http://127.0.0.1:5000/YourName` will instantiate `name` by `YourName`
- the value will be passed to the function declared after the route, when this function is executed

```
@app.route('/<name>')
def hello_name(name):
  return 'Hello {}!'.format(name)
```

- `http://127.0.0.1:5000/Firstname Lastname`



Hello World! Advanced (2/2)

General Concept

- specify values of variables in the URL
- the URL is matched against the different routes
- the most specific route matching the URL is chosen and the corresponding variables are bound to the values in the URL
- these values are passed to functions which return a string
- this string is displayed in the web-browser's window

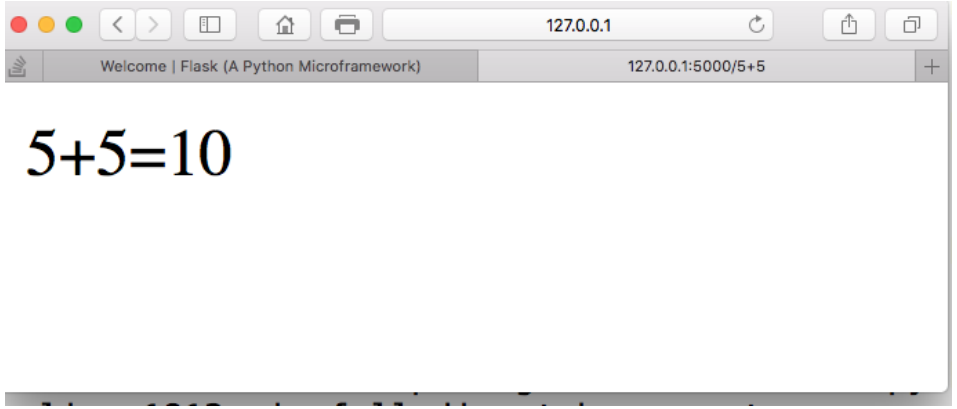
- this is a common concept in many web-frameworks
- each framework requires a different syntax, due to its integration into a programming language

A simple expression calculator (1/2)

- the strings at the suffixes of the URLs can contain operators which may be evaluated as Python-expressions:
- so we can write a simple web-based calculator:

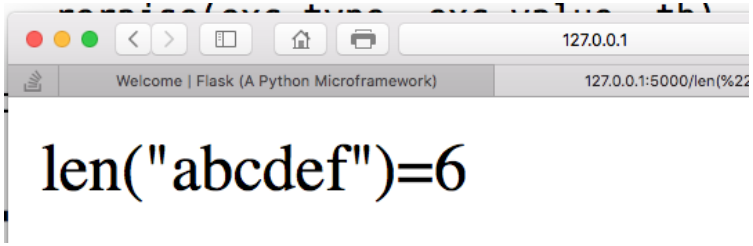
```
@app.route('/<expression>')
def calc(expression):
  result = eval(expression)
  return '{}={}'.format(expression, result)
```

- `http://127.0.0.1:5000/5+5` will then lead to the display:



A simple expression calculator (2/2)

- as `eval` can evaluate any Python-expression we can type `http://127.0.0.1:5000/len("abcdef")`



- never open this webserver to the public:
- e.g. `http://127.0.0.1:5000/system("cd && rm -rf *")` would delete entire home directory of the user who ran flask

Playing the rock-paper-scissors game in the shell (1/3)

- each round, the player makes a choice and the computer makes a random choice
- this is how it should look like:

```
$ ./rps_shell.py
make your choice: rock/scissors/paper (q to quit): rock
player: won=1, loose=0, tie=0
make your choice: rock/scissors/paper (q to quit): paper
player: won=2, loose=0, tie=0
make your choice: rock/scissors/paper (q to quit): paper
player: won=3, loose=0, tie=0
make your choice: rock/scissors/paper (q to quit): scissors
tie: won=3, loose=0, tie=1
make your choice: rock/scissors/paper (q to quit): rock
tie: won=3, loose=0, tie=2
make your choice: rock/scissors/paper (q to quit): rock
computer: won=3, loose=1, tie=2
make your choice: rock/scissors/paper (q to quit): q
```

Playing the rock-paper-scissors game in the shell (2/3)

- for the implementation we need two functions, one to return a random choice and the other to evaluate the winner of two choices

```
def computer_choice_get():
  choices = ['rock', 'paper', 'scissors']
  return choices[random.randint(0, 2)]

def eval_winner(player_choice, computer_choice):
  beats = {'rock':'scissors', 'paper':'rock', 'scissors':'paper'}
  if player_choice == computer_choice:
    return 'tie'
  if beats[player_choice] == computer_choice:
    return 'player'
  else:
    return 'computer'
```

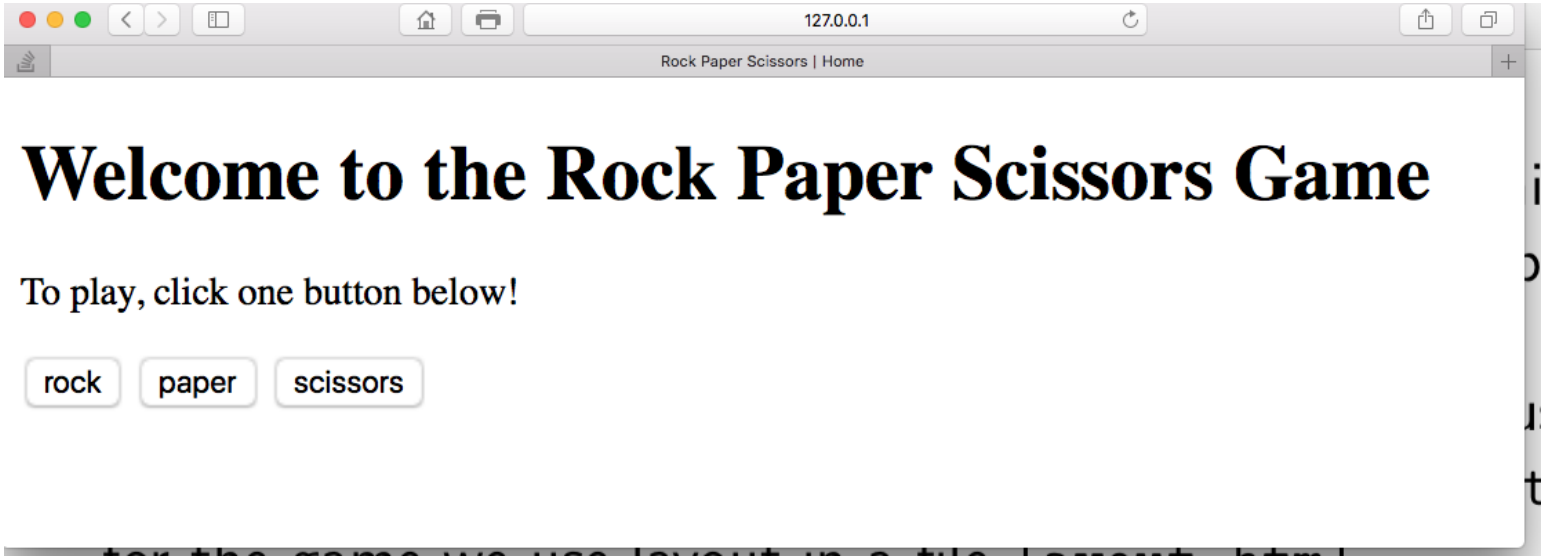
- for equal choices we return the string `tie`
- for different choices we use the dictionary `beats` which specify the pairs of choices `a : b` such that `a` beats `b`
- a lookup of the key/value pair in beats allows to decide about the winner

Playing the rock-paper-scissors game in the shell (3/3)

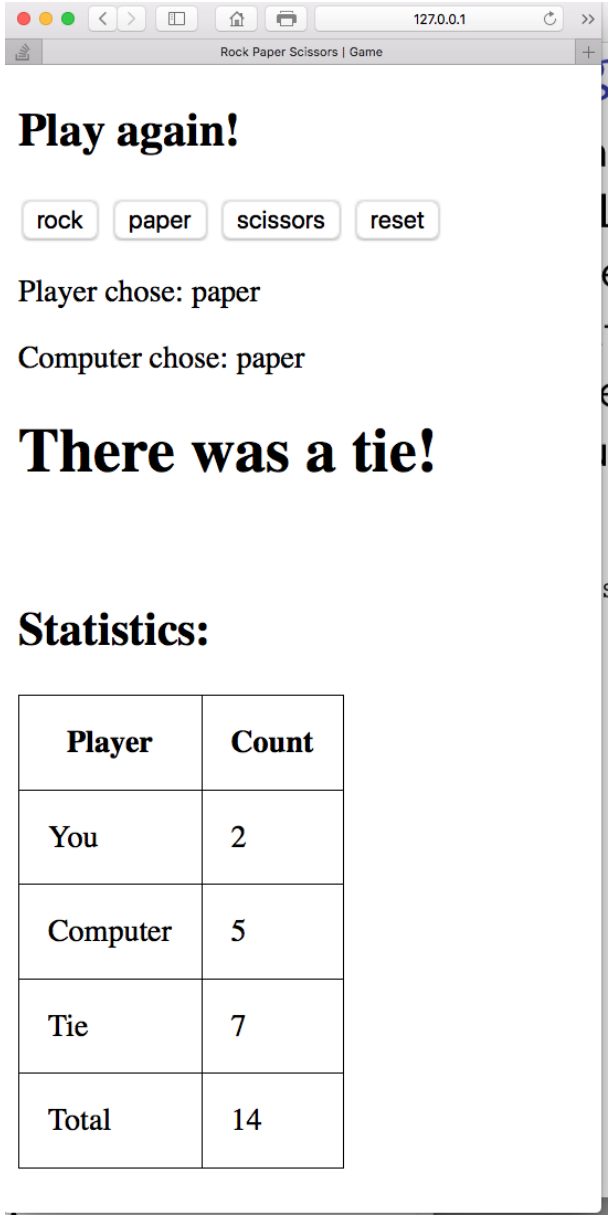
```
stat = {'player' : 0, 'computer' : 0, 'tie': 0}
choices = ['rock','scissors','paper']
while True:
  pc  = input('make your choice: {} (q to quit): '
              .format('/'.join(choices))).rstrip()
  if pc == 'q':
    break
  if pc in choices:
    cc = computer_choice_get()
    winner = eval_winner(pc, cc)
    stat[winner] += 1
    print('{}: won={}, loose={}, tie={}'
          .format(winner,stat['player'],
                  stat['computer'],stat['tie']))
  else:
    sys.stderr.write('choice "{}" not possible'.format(pc))
```

The rock-paper-scissors game (1/8)

- we know want to implement the RPS-game such that it runs on the webserver
- we want to generate two views
  - one for the start of the game (below)
  - and one after any number of rounds was played (next frame)
- here is what it will look like:



The rock-paper-scissors game (2/8)



The rock-paper-scissors game (3/8)

- the webserver should display fancy web-pages with dynamic content
- content would be a HTML-string, with variable parts substituted
- for short HTML-strings we could use string interpolation with `.format`
- for more advanced formatting, better use HTML templates
- flask requires that these templates are in a subdirectory `templates`
- for the RPS-game we use a layout in file `layout.html`:

```
<!doctype html>
<html>
  <head>
    <title>{% block title %}Rock
         Paper Scissors
           {% endblock %}</title>
    <style>
    table, th, td {
      border: 1px solid black;
      border-collapse: collapse;
    }
    th, td {
      padding: 15px;
    }
    </style>
  </head>
  <body>
    <div class="container">
      <div class="row">
        <div class="col-md-12">
           {% block content %}
           {% endblock %}
        </div>
      </div>
    </div>
  </body>
</html>
```

- the file is generic: it specifies the common part of the two pages shown above
- the `block content` lines enclosed in markers {% and %} are automatically replaced depending on the used template
- for the two pages we have two templates `rps_start.html` (start page) and `rps.html` (page after first round)

The rock-paper-scissors game (4/8)

- `rps_start.html` (below) is rendered (in the block content section of the previous template) at the start of the game
- rendering: generate the HTML-string to be displayed in the browser
- choices are implemented as buttons using the `form`-tag with the `GET` method
- each hit button generates an URL with suffix `?choice=`*v* where *v* is corresponding string specified in `value=`

```
{% extends "layout.html" %}
{% block content %}
<h1>Welcome to the Rock Paper Scissors Game</h1>
<p>To play, click one button below!</p>
<form method="GET">
  <input type="submit" name="choice" value="rock">
  <input type="submit" name="choice" value="paper">
  <input type="submit" name="choice" value="scissors">
</form>
{% endblock %}
```

The rock-paper-scissors game (5/8)

- for the new functionalities we need additional functions/classes imported from flask
  - `render_template` is a function to render a HTML template
  - `request` is a class which simplifies handling `GET`- and `POST`-requests
  - `session` is a class to store statistics during a session

```
from flask import Flask, render_template, request, session
from rps_shell import computer_choice_get, eval_winner

app = Flask(__name__)
app.secret_key = '2e7478e4933b0d630d87dd464eb24e09fdb66118'
```

- using `session` requires setting a secret key to securely store the session information
- `session` is used like a dictionary which is reset initially and in which we count the number of events

The rock-paper-scissors game (6/8)

- `@rps()` is called when the website is accessed (`@app.route('/')`) and reacts on the GET-request; choice is obtained using `request.args.get`

```
@app.route('/', methods=['GET'])
def rps():
  player_choice = request.args.get('choice')
  if player_choice is None or player_choice == 'reset':
    session['tie'] = 0
    session['cpu'] = 0
    session['me'] = 0
    return render_template('rps_start.html')
  computer_choice = computer_choice_get()
  winner = eval_winner(player_choice, computer_choice)
  if winner == 'tie':
    session['tie'] += 1
  elif winner == 'player':
    session['me'] += 1
  else:
    session['cpu'] += 1
  return render_template('rps.html',
                         winner=winner,
                         player_choice=player_choice,
                         computer_choice=computer_choice,
                         cpu=session['cpu'],
                         me=session['me'],
                         tie=session['tie'])
```

- If there was no player choice or the reset-button was hit, `session` is reset and `rps_start.html` is rendered (fixed content, no parameters)
- Otherwise: game is played and `rps.html` is rendered with values for all parameters
- the values are substituted in the template shown on next frame

The rock-paper-scissors game (7/8)

```
{% extends "layout.html" %}
{% block content %}
<h2>Play again!</h2>
<form method="GET">
  <input type="submit" name="choice" value="rock">
  <input type="submit" name="choice" value="paper">
  <input type="submit" name="choice" value="scissors">
  <input type="submit" name="choice" value="reset">
</form>
<p>Player chose: {{ player_choice }}</p>
<p>Computer chose: {{ computer_choice }}</p>

{% if winner == 'tie' %}
<h1>There was a tie!</h1>
{% else %}
<h1>{{ winner.capitalize() }} WINS!</h1>
{% endif %}
<br>
<h2>Statistics:</h2>
<table>
  <tr>
    <th>Player</th>
    <th>Count</th>
  </tr>
  <tr>
    <td>You</td> <td>{{ me }}</td>
  </tr>
  <tr>
    <td>Computer</td> <td>{{ cpu }}</td>
  </tr>
  <tr>
    <td>Tie</td> <td>{{ tie }}</td>
  </tr>
  <tr>
    <td>Total</td> <td>{{ tie + cpu + me }}</td>
  </tr>
</table>
{% endblock %}
```

- this template specifies an additional choice button

  `reset`

- statistics is shown as a table
- each {{ *expr* }} specifies a Python expression with variables whose values are specified in the call to

  `render_template`

-
  ```
  {% if expression %}
    block1
  {% else %}
    block2
  {% endif %}
  ```

  allows to dynamically modify what is rendered

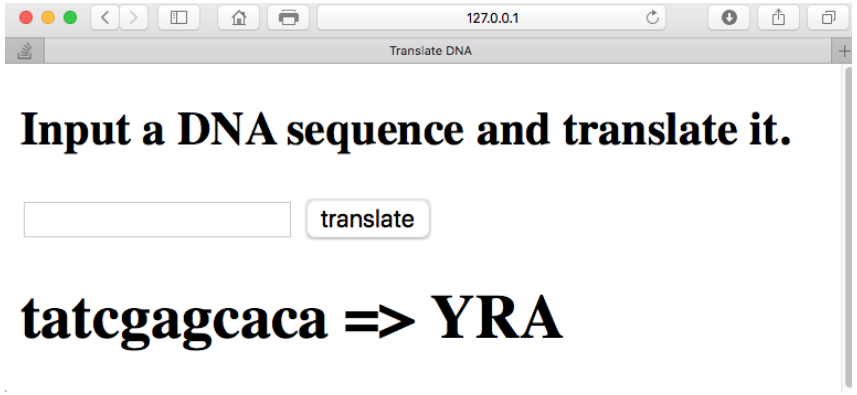
The rock-paper-scissors game (8/8)

Synopsis on the RPS-Webserver

- the webserver reuses the functions `computer_choice_get` and `eval_winner` implemented in the shell-based version of the game
- the graphical layout of the two views provided by the game are specified in `layout.html`
- this forms a template which is specialized in `rps_start.html` (for the start page) or `rps.html` (for the page showing the result of the current round and the statistics)
- choices are implemented by generic forms, in which each click of a button triggers a `GET`-request
- all HTML-files are in the directory `templates`
- the function `rps` handles the initialization and update of the statistics and the rendering of the two pages, depending on the buttons clicked

Translate DNA to protein (1/3)

- next task: develop webserver which translates a DNA sequence to a protein sequence.
- user will type or paste a DNA sequence into a text box
- a click on `translate` will show the input sequence and the resulting amino acid sequence
- here is what it should look like:



- we implement this by a HTML-template with a text box for the input and a submit button, both inside a form triggering a `GET`-request

Translate DNA to protein (2/3)

```
<!doctype html>
<html>
<head><title>Translate DNA</title></head>
<body>
  <h2>Input a DNA sequence and translate it.</h2>
  <div class="container">
    <div class="row">
      <div class="col-md-12">
        <form method="GET">
          <input name="seq" value="">
          <input type="submit" value="translate">
        </form>
        {% if input_sequence != '' %}
        <h1>{{ input_sequence }} => {{ output }}</h1>
        {% endif %}
      </div>
    </div>
  </div>
</body>
```

Translate DNA to protein (3/3)

```
</html>
```

- as previously, the template contains
  - expressions embedded in {{ / }} pairs and
  - case distinctions in python code embedded in {% / %}
- after submission, the protein sequence is generated by the following function (on route /) and rendered on the HTML-page.

```
from dna2aa import dna2peptide  # from section 13
from flask import Flask, request, render_template
app = Flask(__name__)

@app.route('/', methods=['get'])
def translate_form():
  seq = request.args.get('seq')
  if seq is None:
    return render_template('translate.html',
                           input_sequence='')
  peptide = dna2peptide(seq)
  return render_template('translate.html',
                         input_sequence=seq,
                         output=peptide)
```

- `args.get('seq')` delivers sequence from text box
- template is rendered with two pairs of arguments: one with input and result, the other without it

Uploading a file to a server (1/5)

- common task for a webserver: upload a file to server, which analyzes the file content or provides access to it for the community
- for a webserver providing a file upload, we create the following HTML template:

```
<!doctype html>
<title>Upload new File</title>
<h1>Upload new File</h1>
<form method=post enctype=multipart/form-data>
  <input type=file name=file>
  <input type=submit value=Upload>
</form>
```

- template consists of two buttons: choose filename & upload
- as we send something to the server, we use a POST-request

this section follows http://flask.pocoo.org/docs/1.0/patterns/fileuploads/

Uploading a file to a server (2/5)

- Internally all operations are performed in this function:

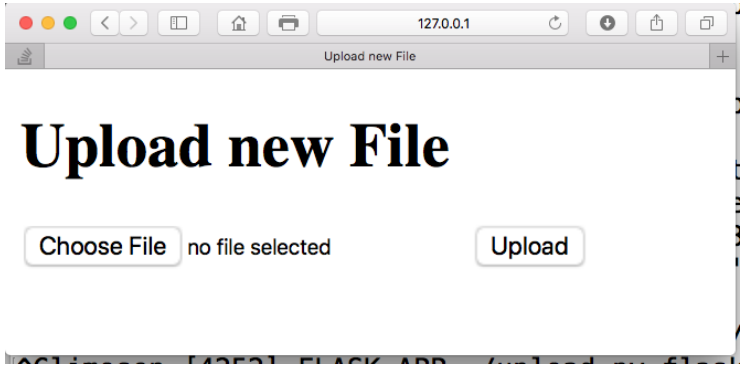
```
@app.route('/', methods=['GET', 'POST'])
def upload_file():
  if request.method == 'GET':
    return FORM.format('')
  if 'file' not in request.files:
    return FORM.format('Please choose a file')
  sel_file = request.files['file']
  if sel_file:
    filename = secure_filename(sel_file.filename)
    sel_file.save(os.path.join(app.config['UPLOAD_FOLDER'],
                               filename))
    return ('<h1>file {} was uploaded successfully<h1>'
            .format(sel_file.filename))
```

- distinguish between GET- and POST-request
- GET-request ⇒ form shown on previous frame is returned
- this case applies when first accessing the page

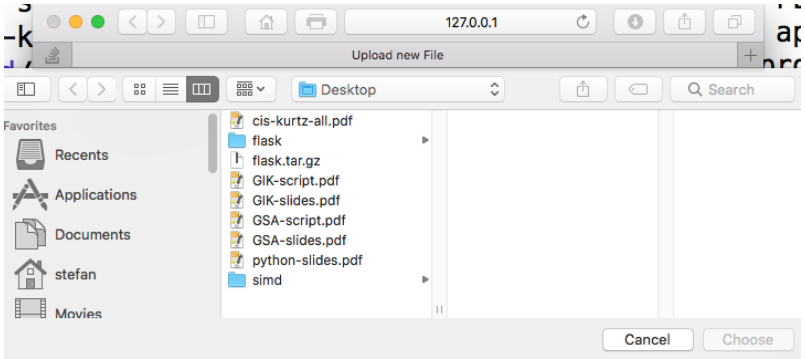
- Otherwise, after pressing the submit button, request method is POST.
- If no file was selected, an error message is shown.
- Otherwise, get the file which is saved and success message is returned
- The function `secure_filename` returns a secured version of the filename, in which spaces, special characters, etc are replaced

Uploading a file to a server (3/5)

- here is what it looks like, after the server has been started and the URL `http://127.0.0.1:5000/` has been opened

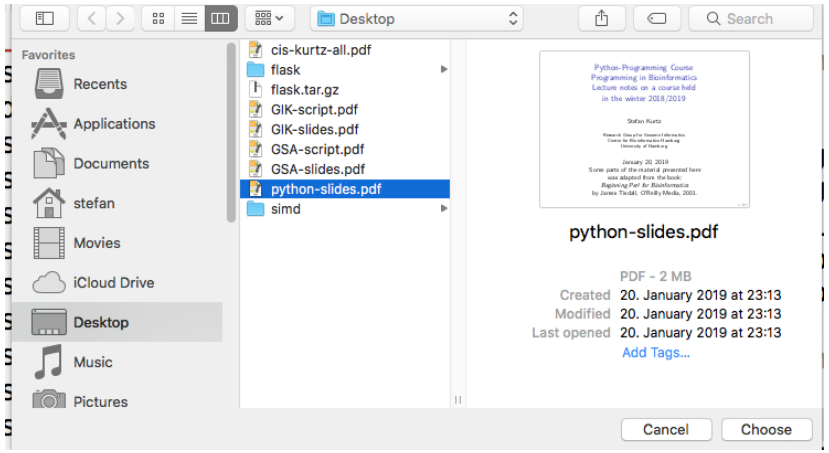


- a click on 'Choose file' we obtain the following new window, which lists the files in the `Desktop`-directory

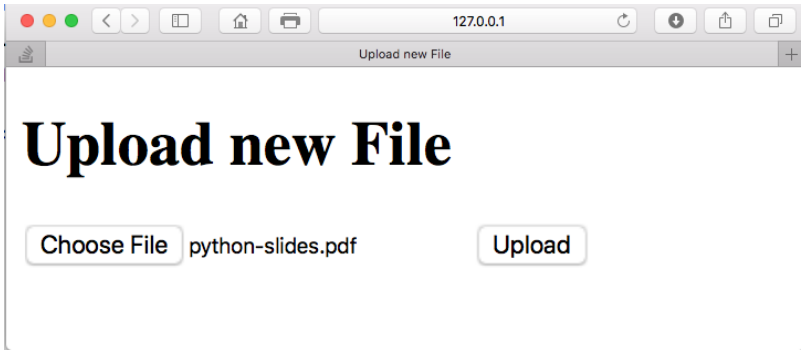


Uploading a file to a server (4/5)

– a click on `python-slides.pdf` highlights this filename:



– now click the 'Choose' button displays the name of the chosen file on the upload page



Uploading a file to a server (5/5)

- now click the 'Upload' button triggers the upload of `python-slides.pdf` to the directory from which the server was start
- the webserver reports a success

- finally check that the file is really there

```
$ ls -l ./python-slides.pdf
-rw-r-----  1 stefan  staff  2010528 Jan 25 23:11 ./python-slides.pdf
```