https://siscourses.ethz.ch/python_one_day (https://siscourses.ethz.ch/python_one_day)

About me

- my name is Uwe Schmitt
- I work for Scientific IT Services of ETH
- I started to use Python in 2003
- Python is still my favorite all purpose language

About Scientific IT Services (SIS)

- group of connected experts from different backgrounds
- assist researchers in IT related issues

For example:

- we run Euler computing cluster
- we developed and maintain openBIS and openBIS-ELN
- we offer consulting related to data analysis and programming
- we develop software on demand
- we maintain software
- we offer courses
- code and data clinics

About this course

- introduction to basic topics and concepts
- we can not learn all Python features
- exercise section for every topic
- your programming speed will vary, so: some of you will not manage to solve all exercises in the given time, this is why we have some optional exercises.
- lunch break one hour
- feel free to take a break during the exercise sections

In this course we use:

- Python 3
- PyCharm IDE installed on your computers (community edition of PyCharm is for free)

About Python 2 vs Python 3

Python community developed Python 3 to clean up the language, but this transition from Python 2 to Python 3 broke backwards compatibility:

- Python 3.Y programs will run with Python 3.X interpreter if Y <= X
- Same for Python 2
- But 2.X programs will not run if your intepreter is 3.Y

Which version should I choose ?

- Python 3 is definitely the better language !
- All new features and improvements happen in Python 3
- Use `Python 3.4` if it should run on other platforms
- Use `Python 3.6` if you can install Python as you like

When to use Python 2?

- existing code base in Python 2
- if you depend on libraries not ported to Python 3
- which is rarely the case

How to use the code from this course with Python 2.7 ?

Just start your script with the following lines, this should make most of the code from today runnable with Python 2.7:

```
from __future__ import print_function, division
try:
    input = raw_input
    range = xrange
except NameError:
    pass
```

Usefull links:

- get Python: http://python.org/downloads (http://python.org/downloads)
- official documentation: https://docs.python.org (https://docs.python.org)
- PyCharm (IDE): https://jetbrains.com/pycharm (https://jetbrains.com/pycharm)
- Jupyter Notebook: https://jupyter.org/ (https://jupyter.org/)
- Hitchhiker's Guide to Python: https://docs.python-guide.org (https://docs.python-guide.org)
- Python Module Of The Week (PyMOTW): https://pymotw.com (https://pymotw.com)

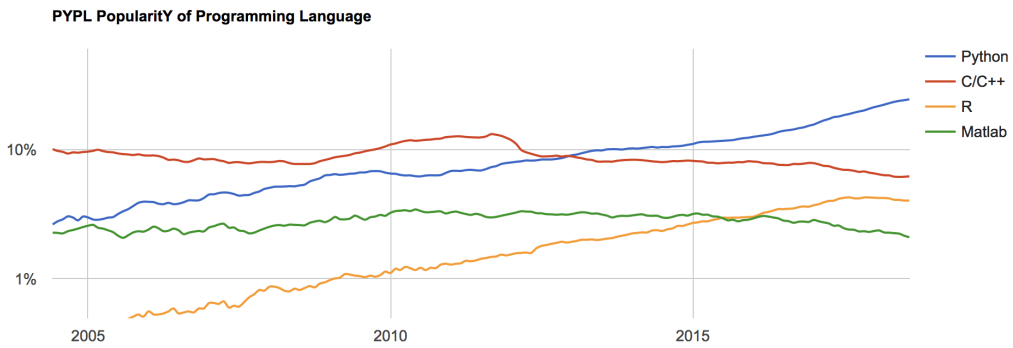
I want to analyze data with Python

See http://www.scipy-lectures.org/intro/ (http://www.scipy-lectures.org/intro/)

About Python

- easy to read syntax and easy to learn
- interpreted language, no compilation step needed
- supports efficient programming
- multi praradigm: object oriented, procedureal and functional concepts
- huge eco system of external libaries (Python packages, "batteries included")
- multi purpose: scientific applications, web frameworks, game programming, ...
- open source
- platform independent (almost)

Increasing popolarity according to google searchs for tutorials



0. Setting up PyCharm

The following instructions may raise warning from the fire wall which you can ignore. It also might ask you some questions related to the styling of PyCharm, the offered default settings are fine.

- Open `PyCharm` on your computer (on Windows under `Jet Brains` in the menu)
- Create a new Project
- Create a new Python script (**new Python File** and NOT **new File**)
- Enter `print(42)` in the script editor
- Execute the program (see `Run` menu)

How to use this script

- the square boxes contain code
- below such a box you see the output
- try to match output to code

- **Dont copy-paste !**

1. Basics: variables, a bit of math and console input/output

- `print` writes strings and values
- a plain `print()` prints a line break aka `"\n"`
- separate arguments with `,`

```
print("values are", 1, 2, 3)
print()
print("done")
```

```
values are 1 2 3

done
```

Dynamic type system

No declaration of variable types, just assign values. Type of variable is determined from value on the right side of `=`:

```
a = 1.23
print(a * a)
```

```
1.5129
```

Check type of a variable with built in `type` function:

```
print(type(a))
```

```
<class 'float'>
```

```
b = 4711 * 42
print(type(b))
```

```
<class 'int'>
```

```
c = "I heart Python"
print(type(c))
```

```
<class 'str'>
```

Rebinding

you may reuse variable names in a script, the type might change:

```
a = 6
print(a, type(a))
a = 3.14
print(a, type(a))
```

```
6 <class 'int'>
3.14 <class 'float'>
```

Comments in Python

```
# this is a single line comment
print(3)   # and a comment at the end of the line

"""
this is a
multiline comment
"""
```

```
3
```

```
'\nthis is a \nmultiline comment\n'
```

Valid variable names:

- start with lower or upper case letter or `_`
- followed by lower or upper case letters or `_` or digits
- names of builtin **functions** allowed (but not recommended)
- names of **Python statements** not allowed.

```
# those are fine:
a_b_c = 1
a123 = 2
_aXzA = 3

# not so good, you see the different color ?
type = 4

# bang !
for = 3
```

```
  File "<ipython-input-9-e0de88e4150d>", line 10
    for = 3
        ^
SyntaxError: invalid syntax
```

PEP 8 (https://www.python.org/dev/peps/pep-0008/ (https://www.python.org/dev/peps/pep-0008/)) recommends:

- use lower case letters and "_" unless you name classes:

  prefer `this_is_a_long_name` over `thisIsALongName`.

- use "CamelStyle" only for class names
- PyCharm IDE indicates PEP 8 violations
- PyCharm "reformat code" functionality helps
- further: use only lower case letters for file names

Naming conflicts:

- built in functions as `type` and statements as `for` may conflict with your preferred variable name
- use `type_` and `for_` instead as names.

Basic math

`+`, `*`, `-`, `/` and parenthesis as usual, `**` for exponentiation:

```
print(2 * (3 + 4) - 7)
```

```
7
```

```
print(2 ** 10)
```

```
1024
```

The carret operator `^` also exists in Python (https://stackoverflow.com/questions/2451386/what-does-the-caret-operator-in-python-do (https://stackoverflow.com/questions/2451386/what-does-the-caret-operator-in-python-do)) but **does not compute exponentation** as in some other programming languages:

```
print(2 ^ 10)
```

```
8
```

Python 3 (Python 2 was different) always does floating point division:

```
print(-7 / 2)
```

```
-3.5
```

Integer divison (round down to the next integer) is `//`:

```
print(-7 // 2)
```

```
-4
```

`%` for modulo (aka division reminder) computation:

```
print(13 % 4)
```

```
1
```

We use `%` in a few examples in this script to check if a given number is a multiple of another number. For example `12` is a multiple of `4` because `12 % 4` is zero.

Integers do not overflow in Python:

```
x = 2 ** 62  # this can be represented by a 64 bit integer.
y = 2 ** 63  # this overflows in 64 bit integers
print(x, y)
```

```
4611686018427387904 9223372036854775808
```

There is no distinction between single or double precision floats, the Python `float` type is always with double precision, but may overflow:

```
print(2.0 ** 1000)
print(2.0 ** 1500)
```

```
1.0715086071862673e+301
```

```
---------------------------------------------------------------------
--------
OverflowError                             Traceback (most recent ca
ll last)
<ipython-input-17-4dd8e105c5a9> in <module>()
      1 print(2.0 ** 1000)
----> 2 print(2.0 ** 1500)

OverflowError: (34, 'Result too large')
```

Algebraic updates:

```
x = 3
x += 3  # same as x = x + 3
x *= 2  # same as x = x * 2
x /= 4  # same as x = x / 4
print(x)
```

```
3.0
```

Math with the `math` module

- Python ships with extra modules collected in the so called "Python standard library".
- The documentation on http://python.org (http://python.org) is complete but https://pymotw.com/3/ (https://pymotw.com/3/) is more detailed and shows more examples.
- Use `import` at the beginning of your script to use such modules.

```
import math
```

```
print(math)
```

```
<module 'math' from '/usr/local/Cellar/python35/3.5.2/Frameworks/Python.framework/Versions/3.5/lib/python3.5/lib-dynload/math.cpython-35m-darwin.so'>
```

Now functions and constanstants are "attached" to `math` (Python speak: "attriibutes" of "math"):

```
print(math.pi)
```

```
3.141592653589793
```

```
print(math.sin(1.0))
```

```
0.8414709848078965
```

Python help system:

```
print(help(math))   # lots of output
print(help(sum))
```

Alternative ways to import functions, values, etc:

```
from math import pi, e, sin, log
```

```
print(log(e))
```

```
1.0
```

`from math import *` works, it imports everything (which might be a lot), but is **dangerous**, for example this overwrites a variable `e`:

```
e = 123
from math import *
print(e)
```

```
2.718281828459045
```

Simple input output `input`

```
name = input("what is your name ? ")
print("hi", name, "how do you do ?")
print(type(name))
```

```
what is your name ? bob
hi bob how do you do ?
<class 'str'>
```

Type conversions:

```
print(float("1.23"))
```

```
1.23
```

```
print(int("42"))
```

```
42
```

```
print(int("1.2"))
```

```
---------------------------------------------------------------------
--------
ValueError                                Traceback (most recent ca
ll last)
<ipython-input-30-3063a460e01f> in <module>()
----> 1 print(int("1.2"))

ValueError: invalid literal for int() with base 10: '1.2'
```

```
x = float(input("give me a number: "))
print(x, "squared is", x * x)
```

```
give me a number: 1.23
1.23 squared is 1.5129
```

Exercise session 1

1. What is the reminder of 2^{63} divided by 13 ?
2. Which number is larger: π^e or e^π ?
3. Start a "Python Console" in PyCharm (you find this in "Tools" menu). The console allows you to enter Python commands for direct execution.
4. Type in the console: `import math`, then `help(math.hypot)` and read the output.
5. Write a script which asks for the diameter of a circle and computes its area and circumference.
6. Run `import this`.
7. Run `import antigravity`.

Optional exercises:

1. Write a program which asks for the coefficients p and q in the equation $x^2 + px + q = 0$ and prints the solution(s):
$$x = -\frac{p}{2} \pm \sqrt{\left(\frac{p}{2}\right)^2 - q}.$$
What x do you get for $p = 2$ and $q = 1$? What happens if you use $p = 1$ and $q = 1$?
2. Use the module `cmath` instead of `math` for the preceeding exercise and test again with $p = 1$ and $q = 1$.
3. Compute `math.hypot(1e300, 1e300)`. Now implement the same computation using the pythagorean theorem. and do the same computation again, what do you observe ? Can you explain your implementation and rewrite the formula to match the result of `math.hypot(1e300, 1e300)`?

2. String basics

Strings are defined using delimiters `"` or `'` or `"""` or `'''`:

If you choose `"` as delimiter you may use `'` in the string and the other way round.

```
print("hi, it's time to go")
```

```
hi, it's time to go
```

```
print('this is "a quote"')
```

```
this is "a quote"
```

```
long = """multi line string ...
it works"""

print(long)
```

```
multi line string ...
it works
```

The `repr` function gives us more detailed information (usefull when debugging)

```
print(repr(long))
```

```
'multi line string ...\nit works'
```

Multi line comments in Python

```
# this is a single line comment

print(3)

"""
this is a multi line comment
the comment ends here
"""

print(4)
```

```
3
4
```

String "algebra":

```
print("3.1" + '41')
```

```
3.141
```

```
print(3 * "\o/ ")
```

```
\o/ \o/ \o/
```

```
print(len("12345"))
```

```
5
```

Creating strings using string interpolation (old fashioned)

String interpolation relaces placeholders eg `%s` by given values. The expression

```
template % args
```

creates a new string by replacing the placeholders in `template` provided by the value(s) in `args`:

```
name = "uwe"
greeting = "hi '%s' how do you do" % name
print(greeting)
```

```
hi 'uwe' how do you do
```

You can have multiple placeholders and arguments, but the number of placeholders and the number of arguments must be the same. For multiple arguments you have to use paranthesis as shown below:

```
a = 1
b = 2
output = '%s plus %s is %s' % (a, b, a + b)
print(output)
```

```
1 plus 2 is 3
```

Other placehoders as `%s` exist, eg for formatting floats with given precision:

```
import math
print("pi up to 3 digits is %.3f" % math.pi)
```

```
pi up to 3 digits is 3.142
```

More details at https://docs.python.org/3/library/stdtypes.html#string-formatting-operations (https://docs.python.org/3/library/stdtypes.html#string-formatting-operations)

String formatting (modern!)

```
print("{} = {} + {}".format(a + b, a, b))
print("{2} = {0} + {1}".format(a, b, a+b))
print("{c} = {a} + {b}".format(a=a, b=b, c=a+b))
```

```
3 = 1 + 2
3 = 1 + 2
3 = 1 + 2
```

```
print("{pi:.3f}".format(pi=math.pi))
```

```
3.142
```

Many more options, see https://www.digitalocean.com/community/tutorials/how-to-use-string-formatters-in-python-3 (https://www.digitalocean.com/community/tutorials/how-to-use-string-formatters-in-python-3) and the cheat sheet at https://pyformat.info/ (https://pyformat.info/)

String methods

Many string operations are "attached" to string object.

Python strings are immutable ("const"). So string methods never change the string oject in place. So for example the following `upper` method creates and returns a new string:

```
# transforms string "hello" to a new string:
greeting = "hello"
print(greeting.upper())
print(greeting)        # unchanged !
```

```
HELLO
hello
```

Method calls can be chained. For example this `startswith` method ...

```
print("hi you".startswith("hi"))
```

```
True
```

... can be called on the result of `upper()`:

```
print("hi you".upper().startswith("HI"))
```

```
True
```

Overview of available string methods

You can use

```
print(help(str))
```

to list all available string methods

Some useful string methods:

- `count(substring)` counts non overlapping occurrences of substring
- `replace(a_string, b_string)` replaces all occurrences of a_string by b_string
- `lower()` and `.upper()` convert characters to upper resp. lower case
- `strip()` removes all white-spaces (space, tab and new line characters) from both ends of the string
- `strip(characters)` removes all single characters occurring in characters from both ends of the string.
- `lstrip() as .strip()` but only from the beginning of the string
- `rstrip() as .strip()` but only from the end of the string
- `startswith(txt)` checks if the given strings starts with txt
- `endswith(txt)` checks if the given string ends with txt.

You find a more complete list at https://www.shortcutfoo.com/app/dojos/python-strings/cheatsheet (https://www.shortcutfoo.com/app/dojos/python-strings/cheatsheet)

String "slicing"

Use `[..]` for accessing parts of a string, counting start with `0`.

```
print("Python"[1])
```

```
y
```

Negative indices start at the end, `-1` is the last character, `-2` the character before the last character and so on:

```
print("Python"[-2])
```

```
o
```

To access substrings we use the so calles *slicing* notation `[m:n]`, the first value is the starting index, the secon one the end index, the end index is **exclusive**:

```
print("Python"[2:4])
```

```
th
```

Why exlusive right limits ?

The following relations hold for slicing:

1. `len(a[n:m]) == m - n`
2. `a[i:j] + a[j:k] == a[i:k]`.

Some other examples for slicing:

```
print("Python"[1:-1])
```

```
ytho
```

short forms:

```
print("Python"[:2])
```

```
Py
```

```
print("Python"[2:])
```

```
thon
```

Limits can be exeeded:

```
"abc"[1:5]
```

```
'bc'
```

```
"abc"[5:7]
```

```
''
```

Strings are immutable

You can not modify a string in place, instead you have to create a new one !

Exercise session 2

1. Reproduce the examples starting at "Creating strings using string interpolation".
2. Try to forecast the values of the variables in the following snippet using pen and paper, use `help(str.rstrip)` or the internet for looking up the used methods. Then use Python to validate your results.

```
values = "012" * 3 + """'a'bc"""
a = values[:2] + values[0] + values[2:3]
b = a + values[len(values) - 2].upper()
c = a.strip('0')
d = a.find("A")
e = "{2} / {1} / {0}".format(a[:3], a[3:5], a[:5])
# print(values, a, b, c, d, e)
```

3. Python functions

The general syntax to define a function in Python is:

```
def <name>(<arg_0>,...):
    <body>
```

For example:

```
def print_squared(x):
    print(x, "squared is", x * x)
```

```
print_squared(42)
```

```
42 squared is 1764
```

A Function without arguments, **all lines with same indentation** form the body of the function:

```
def say_hello():
    print("hello")
    print("how do you do !")
    print("nice weather, eh ?")
print("hi")
```

```
hi
```

- as soon as indentation decreases the body of the function is completed.
- no curly braces
- no "end" or similar statements
- this applies also for branches and loops

```
say_hello()
```

```
hello
how do you do !
nice weather, eh ?
```

Good style:

1. use four spaces (or multiples) for indentation
2. no TABs (PyCharm automatatically replaces TAB by spaces !)

Return statement

```
def times_3(x):
    return 3 * x
```

```
print(times_3(7))
```

```
21
```

"Duck Typing"

No type declaration for the arguments. During execution of the function Python determines if operations on arguments fit:

```
print(times_3("ab"))
```

```
ababab
```

A missing or plain `return` statement returns `None`:

```
def do_nothing(x):
    return

def print_x(x):
    print("x is", x)

print(do_nothing(0))
```

```
None
```

```
print(print_x(1))
```

```
x is 1
None
```

Python supports multiple return values:

```
def sum_and_diff(x, y):
    sum_ = x + y
    diff = x - y
    return sum_, diff

a, b = sum_and_diff(7, 3)
print("sum is", a)
print("diff is", b)
```

```
sum is 10
diff is 4
```

Python doc strings

```
def average_3(a, b, c):
    """this function computes the average of
    three given numbers
    """
    return (a + b + c) / 3.0

help(average_3)
```

```
Help on function average_3 in module __main__:

average_3(a, b, c)
    this function computes the average of
    three given numbers
```

A few words about execution order

```
def fun():
    print("i am fun")
    gun()

def gun():
    print("i am gun")

fun()
```

```
i am fun
i am gun
```

Calling `gun` within `fun` works, to understand this we follow the order of execution:

1. Python interpreter starts to execute script at first line
2. then Python interpreter sees declaration of `fun` and stores function name and start of end of function body.
3. then Python interpreter sees declaration of `gun` and stores function name and start of end of function body.
4. then Python interpreter sees call of `fun`, because of step 1 the interpreter knows where the body of `fun` starts and thus starts to execute the corresponding code lines.
5. After the `print("i am fun")` Python interpreter sees call of `gun` and again: because of step 2 the interpreter knows where the body of `gun` starts and thus starts to execute the corresponding code lines.

Default arguments (when defining a function)

You can declare functions with different number of arguments:

```
def greet(name, formula="hi", ending="?"):
    print(formula, name, "how do you do", ending)

# this uses the default value "hi" for the second argument:
greet("uwe")

# and this overruns the first default argument:
greet("urs", "gruezi")

# and this one overungs all:
greet("buddy", "yo", "\o/")
```

```
hi uwe how do you do ?
gruezi urs how do you do ?
yo buddy how do you do \o/
```

Naming arguments (when calling a function)

You can name parameters with `=` when calling a function, so you do not have to remember the order and your code is better readable as you need not to look up the meaning of parameters:

```
greet("urs", ending="???", formula="gruezi wohl")
```

```
gruezi wohl urs how do you do ???
```

Another example:

```
# declare function with default arguments as seen above:
def exchange_chf_to_eur(money, bank_discount=0.02, rate=1.2):
    return money * rate * (1.0 - bank_discount)

print(exchange_chf_to_eur(100.0, rate=1.05))
```

```
102.89999999999999
```

```
print(exchange_chf_to_eur(rate=1.05, money=200.0, bank_discount=0.01))
```

```
207.9
```

Exercise Session 3

1. Repeat the examples above, starting at "Multiple return values"
2. Write a function which takes the diameter of a circle and returns its area and circumference
3. Write a function which takes 1 up to 3 values and computes their product:

```
product(2) returns 2
product(2, 3) returns 6
product(2, 3, 4) returns 24
```

Optional exercises (higher order functions):

We can pass functions as function arguments too, so try to understand what the following functions do:

```
def avg_at_1_2(f):
    return (f(1) + f(2)) / 2.0

print(avg_at_1_2(math.sqrt))
```

```
1.2071067811865475
```

```
def compose(f, g):
    def f_after_g(x):
        return f(g(x))
    return f_after_g

f = compose(math.exp, math.log)
print(f(3))

f = compose(math.sqrt, math.asin)
print(f(1.0))
```

```
3.0000000000000004
1.2533141373155001
```

4. If / elif / else

Logical values

Python has a type `bool` which can take two values `True` and `False`:

```
ok = True
print(ok, type(ok))
```

```
True <class 'bool'>
```

Comparisons:

Logical values result from comparing numbers:

notation	meaning
`a < b`	a is less than b
`a > b`	a is greater than b
`a <= b`	a is less than or equal to b
`a >= b`	a is greater than or equal to b
`a == b`	a is is equal to b
`a != b`	a is not equal to b

Comment:

- `=` aka variable assignment is a **statement** ("it does something")
- `==` aka test for equality is an **expression** (it can be evaluated to compute a value)

Logical computations:

Logical values can be combined

notation	meaning
`a and b`	`True` if `a` and `b` are `True`
`a or b`	`True` if `a` or `b` are `True`
`not a`	`True` if `a` is `False` else `False`

```
print(3 > 4 or 4 > 3)
```

```
True
```

```
print(3 < 7 and 7 < 12)
```

```
True
```

if / elif /else

- Python uses `if`, `elif` and `else` keywords for branching code execution.
- No `else if`!
- The level of indentation defines the blocks, no "end" statement or braces !
- after `if` follow zero to `n` `elif`s and then zero ore one `else`.

```
def test_if_even(x):
    if x % 2 == 0:
        print(x, "is even")
    else:
        print(x, "is odd")

test_if_even(12)
```

```
12 is even
```

indentations can be nested:

```
def some_tests(x):
    if x > 0:
        if x % 2 == 0:
            print(x, "is positive and even")
        else:
            print(x, "is positive and odd")
    elif x == 0:
        print (x, "is zero")
    else:
        print (x, "is negative")
```

Code blocks:

Rule:

```
A code block ends if the level of indentation becomes less than the indent
ation of the first line of the block. Or if the program ends.
```

Recommended:

```
Use multiples of 4 spaces for indentation
```

```
some_tests(4)
```

```
4 is positive and even
```

```
some_tests(-1)
```

```
-1 is negative
```

Exercise block 4

1. Repeat the examples above
2. Write a function which takes one value and doubles this if the value is even, else return the value unchanged. So the function returns `4` for input `2` and `3` for input `3`.
3. Write a function which takes a value and tests if it is a multiple of three and if it is a multiple of four. The function prints an appropriate message for all four cases.
4. In the early days of `Python` some users disliked indentation and requested to implement curly brace in Python to mark code block (like in `C`ish languages and `R`). Run `from __future__ import braces` and you can see the answer of the Python core developers.

Optional exercises

1. Try to forecast and understand the output of the following snippet:

```
def return_true():
    print("this is return_true function")
    return True

def return_false():
    print("this is return_false function")
    return False

# this is "short circuit evaluation", see https://en.wikipedia.org/wiki/Short-ci
rcuit_evaluation
print(return_true() or return_false())
print()
print(return_false() or return_true())
print()
print(return_true() and return_false())
print()
print(return_false() and return_true())
```

```
this is return_true function
True

this is return_false function
this is return_true function
True

this is return_true function
this is return_false function
False

this is return_false function
False
```

See https://en.wikipedia.org/wiki/Short-circuit_evaluation (https://en.wikipedia.org/wiki/Short-circuit_evaluation)

1. "Recursion" aka "a function calls itself": Try to understand what the following function computes:

```
def compute(n):
    if n <= 1:
        return 1
    return n * compute(n - 1)
```

Can you implement a function which computes the sum of the first `n` numbers using recursion ?

5. Python loops

Python has `while`:

- if the condition after `while` is `False` from the beginning, skip the associated code block
- else execute the code block and "jump back" to `while`.
- check again
- etc

```
i = 11
while i % 5 != 0:
    print(i)
    i = 2 * i +1
```

```
11
23
47
```

and `continue` and `break`:

- `continue` skips the end of the `while` code block and "jumps back" to `while` immediately
- `break` stops looping and program execution contiues after the `while` block

```
x = 9
while x > 0:
    x = x - 1
    if x % 2 == 0:
        continue   # skips rest of body of while
    print(x)
    if x % 3 == 0:
        break      # quit body of while

print("done")
```

```
7
5
3
done
```

No `do until` (or similar). Use `while True:` + `break` instead.

```
while True:
    symbol = input("give me one letter: ")
    if len(symbol) == 1:
        break
    print("this was not one symbol, try again !")
```

```
give me one letter: 123
this was not one symbol, try again !
give me one letter:
this was not one symbol, try again !
give me one letter: 3
```

Python lists

Python has some container types for collecting values. `list` is one such a type:

```
li = [1, 2, 4, 8]
print(li)
```

```
[1, 2, 4, 8]
```

length of a list:

```
print(len(li))
```

```
4
```

```
print(type(li))
```

```
<class 'list'>
```

The empty list is `[]`:

```
print(type([]))
print(len([]))
```

```
<class 'list'>
0
```

List of strings:

```
li = ["hi", "ho"]
```

Mixed types:

```
li = [1, 2.0, True, "hi"]
print(li)
```

```
[1, 2.0, True, 'hi']
```

More about lists below.

for loops

`for` has the general form

```
for <variable> in <iterable>:
    <codeblock>
```

An example for such an **iterable** are lists:

```
for name in ["urs", "uwe", "guido"]:
    print("I say hi to", name)

print("I also say hi to everybody I forgot")
```

```
I say hi to urs
I say hi to uwe
I say hi to guido
I also say hi to everybody I forgot
```

Here `name` is a variable which you can name as you like.

In the first iteration `name` is `urs`, in the second iteration `name` is `uwe`, in the third iteration `name` is `guido`. Then the list is exhausted and iteration stops.

Here we iterate over a list of numbers to sum them up:

```
def sumup(numbers):
    sum_ = 0.0
    for number in numbers:
        sum_ = sum_ + number
    return sum_

print(sumup([1, 2, 3]))
```

```
6.0
```

For counting loops use the `range` function which returns an iterable:

```
for i in range(4):
    print(i, "squared is", i * i)
```

```
0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
```

Now with different starting value:

```
for i in range(1, 4):
    print(i, "squared is", i * i)
```

```
1 squared is 1
2 squared is 4
3 squared is 9
```

And with a step size:

```
for i in range(1, 4, 2):
    print(i, "squared is", i * i)
```

```
1 squared is 1
3 squared is 9
```

Antipattern (C style)

This is correct code, and correlates to the way you iterate in languages as `C`:

```
numbers = [1, 3, 5]
for i in range(len(numbers)):
    print(numbers[i])
```

```
1
3
5
```

The more "python" and thus more readable version is:

```
for number in numbers:
    print(number)
```

```
1
3
5
```

The `range` function returns an iterable:

```
print(range(1, 4))
```

```
range(1, 4)
```

To see what an iterable produces (and thus how it behaves when used in a `for` loop), you can pass it to the `list` funtion which converts an iterable to the list of the values the iterable produces:

```
print(list(range(1, 4)))
```

```
[1, 2, 3]
```

Comment: Python also has some **INFINITE iterators**, so be prepared !

Exercise block 5

1. Repeat examples above
2. Write a function which takes a list of numbers and computes their average (extend the example for summing up numbers)
3. Modify this function so that it computes the average of all even numbers in the list. You also have to count the even numbers for this.
4. What happes if you loop with `for` over a string ? What is the result if you pass a string to `list()`.
5. Can you write shorter expressions for `range(0, 3)` and `range(1, 4, 1)` ?

1. `import random`, lookup function `random.randint`.
2. Programm "number guessing game":
   - Computer generates random secret number in range 0 .. 100
   - User guesses number until secret number is found
   - After every guess the computer tells if guess is too low or too high. Hint: create a random number, then use a `while True` loop and `break` if the users guess is correct.
3. Write a function which tests if a given number is prime.

6. More about Python container types

Lists

Python lists collect data, types may be mixed and the list may be as long as your computers memory allows.

Some usefull, but not all list methods:

```
numbers = [1, 2, 3]
numbers.append(0)

print(numbers)
```

```
[1, 2, 3, 0]
```

Comment: string methods do not change the corresponding string inplace but return a result. `append` returns `None` but changes the list in place:

```
numbers = [1, 2, 3]
numbers_new = numbers.append(0)

print("numbers_new is", numbers_new)
print("numbers is", numbers)
```

```
numbers_new is None
numbers is [1, 2, 3, 0]
```

```
print(len([1, 2, 4]))
```

```
3
```

```
print(numbers)
numbers.sort()
print(numbers)
```

```
[1, 2, 3, 0]
[0, 1, 2, 3]
```

To find the position of an element:

```
print(numbers.index(2))
```

```
2
```

```
print(numbers.index(4))
```

```
---------------------------------------------------------------------
--------
ValueError                                Traceback (most recent ca
ll last)
<ipython-input-109-d2c585bc2403> in <module>()
----> 1 print(numbers.index(4))

ValueError: 4 is not in list
```

Element access, similar to strings:

```
print(li)
print(li[0], li[-1])
```

```
[1, 2.0, True, 'hi']
1 hi
```

List slicing, similar to strings:

```
print(li[1:-1])
```

```
[2.0, True]
```

In contrast to Python strings, which are immutable ("const"), you can use index access as well as slicing for manipulation of a list

```
li[1] = 37
print(li)
```

```
[1, 37, True, 'hi']
```

Deletion of parts of a list works like this:

```
del li[1:3]
print(li)
```

```
[1, 'hi']
```

Tuples

"immutable" lists. Use round instead of square brackets:

- mixed types allowed too
- slicing works

```
a = (1, 3, 5)
print(a)
print(type(a))
```

```
(1, 3, 5)
<class 'tuple'>
```

Rules of thumb:

- Use lists for collecting data of same type
- Use tuples for grouping data of different type

For example like this:

```
person_1 = ("jesus", 2015)   # name + age
person_2 = ("watson", 87)

persons = [person_1, person_2]
```

Slicing / index access again:

```
tp = (1, 2, (1, 2), "")
print(tp[1:-1])
```

```
(2, (1, 2))
```

Empty tuple is `()`, for one element tuples use `(x,)` notation:

```
print((1,))  # prints tuple with one element
print((1))   # prints integer number 1
```

```
(1,)
1
```

Member ship test

The keyword `in` tests for membership:

```
print(2 in [0, 1, 2, 3])
```

```
True
```

```
print(2 in (0, 1, 2, 3))
```

```
True
```

For strings `in` tests for substrings:

```
print("ab" in "asdfabc")
```

```
True
```

Negation

```
print(not (2 in (1, 2, 3)))
```

```
False
```

simpler:

```
print(2 not in (1, 2, 3))
```

```
False
```

Exercises block 6

- repeat the examples from above
- write a function which takes a list of numbers and returns a new list which contains the squares of the odd numbers from the input list.

Optional exercise

- Look up the definition of the fibionacci number sequence
- write a function which takes an integer number n and computes a list of the first n fibionacci numbers (hint: start with `[1, 1]` and extend the list).
- What is the output of the following program:

```
def f(x, a=[]):
    a.append(x)
    return len(a)

f(0)
f(1)
```

You might use additional `print` statements to inspect what's going on here.

7 Dictionaries

Dictionaries, aka "hash tables" or "look up tables" allow presentation of two column tables. The map a `key` to a `value`.

For example:

first name (key)	family name (value)
monty	python
curt	cobain

```
first_to_family_name = { 'monty': 'python',
                         'curt' : 'cobain',
                       }
```

```
print(first_to_family_name)
```

```
{'monty': 'python', 'curt': 'cobain'}
```

You see above that printing the dictionary has a different order than in its definition. Dictionaries are only for representing a mapping from values in the left column of the table to their counterpart in the right column. Ordering is not respected.

To lookup up a value use brackets:

```
print(first_to_family_name["monty"])
```

```
python
```

You can insert new values or overwrite existing values like this:

```
first_to_family_name["uwe"] = "schmitt"
print(first_to_family_name)
```

```
{'monty': 'python', 'curt': 'cobain', 'uwe': 'schmitt'}
```

Size of a dictionary:

```
print(len(first_to_family_name))
```

```
3
```

Left column of table are "keys":

```
print(first_to_family_name.keys())
```

```
dict_keys(['monty', 'curt', 'uwe'])
```

Right column are "values":

```
print(first_to_family_name.values())
```

```
dict_values(['python', 'cobain', 'schmitt'])
```

Comment: `.keys()` and `.values()` return iterators which look like a list and partially behave like a list. So you can use `for` to iterate over keys and values, but you can not append to them.

The empty dictionary is `{}`:

```
d = {}
print(d)
print(len(d))
```

```
{}
0
```

Restrictions

- values of dictionaries may have arbitrary type
- keys must be immutable (so: on lists, but tuples, int, float, str, bool, ...)

Lookup of non existing keys:

```
print(first_to_family_name["jesus"])
```

```
---------------------------------------------------------------------
--------
KeyError                                  Traceback (most recent ca
ll last)
<ipython-input-131-1f6c4e093fdb> in <module>()
----> 1 print(first_to_family_name["jesus"])

KeyError: 'jesus'
```

in order to test if a value appears as an key of a dictionary use `in`: **only checks keys, not values !**

```
print("jesus" in first_to_family_name)
```

```
False
```

```
print(first_to_family_name.get("monty"))
print(first_to_family_name.get("jesus"))
```

```
python
None
```

Dictionaries may have different types for keys and values:

```
what_a_mess = { 3      : 9,
                5      : {25: 125},
                1.2    : (1, 2),
                "four": [4]
              }
print(what_a_mess)
```

```
{1.2: (1, 2), 3: 9, 5: {25: 125}, 'four': [4]}
```

```
print(what_a_mess[5])
print(what_a_mess[5][25])
```

```
{25: 125}
125
```

A few comments on dictionaries:

- writing and reading from a dictionary is extremly fast, also for huge dictionaries.
- dictionaries are very helpful in many places
- the Python interpreter internally is built around dictionaries

This is an example usage how to use a dictionary to create a histogram of numbers when the actual numbers are not known from the begining:

```
def histogram(elements):
    histogram = {}
    for element in elements:
        if element not in histogram.keys():
            histogram[element] = 1
        else:
            histogram[element] += 1
    return histogram

print(histogram([1, 2, 1,  3]))
```

```
{1: 2, 2: 1, 3: 1}
```

Exercises block 7

- repeat the examples above
- create a dictionary which maps numbers 1 to 100 to their squares

Optional exercises

- Use the debugger of PyCharm, maybe introduce temporary variables, Pythons help system to understand the following functions:

```
def do_something(z):
    cc = {}
    for x in z.split():
        cc[x] = cc.get(x, 0) + 1
    return cc

print(do_something("love love me doo"))
```

```
{'love': 2, 'doo': 1, 'me': 1}
```

```
def something_other(z):
    c = []
    for w in z.split(","):
        if len(w) % 2 == 0:
            c.append(w)
    c.sort()
    return "-".join(c)

print(something_other("hi,what is,this,good,for"))
```

```
good-hi-this
```

- Write a function which inverts a dictionary. So it maps values to keys of a given dictionary. This works only if the values are unique.
- Now assume the values are not unique and write a function which creates a dictionary mapping values to a list of keys from the given dictionary. Eg for input `{1: 2, 2: 2, 3: 4}` the function returns `{2: [1, 2], 4: [3]}`.

Not covered here: sets

The basic function of a set is to answer "is a given value contained in a given set ?".

This allows answering questions like

- what is the intersection of two sets ?
- what is the union of two sets ?
- what is the set difference between two sets ?
- etc

Properties:

- Elements in a set have no order
- one element can only occur once
- membership lookup is very fast

sets: http://www.learnpython.org/en/Sets (http://www.learnpython.org/en/Sets)

8. File I/O

Open a text file (the traditional way) for writing

The following example creates a text file holding two lines `hi\n` and `ho\n`:

```
fh = open("say_hi.txt", "w")
print("hi", file=fh)
print("ho", file=fh)

# always close a file because data may be in buffer:
fh.close()
```

- `open(path, mode)` returns an file object (file handle) which we use for manipulating the given file.
- `mode` maybe `"r"`, `"w"`, `"a"` (or some other types...) for "reading", "writing" and "appending".
- the `file=fp` named parameter when calling `print` redirects the output to the file.
- `fp.close` closes the file.

```
#show output of say_hi.txt
```

```
hi
ho
```

This "traditional way" is dangerous if you forget to close the file or if an error resumes programm execution before the `close` method is called !

Background: most OS do not write immediatly to disk if you call `write` but collect data until an internal memory region (buffer) is filled. So you never now exactly what is still in the buffer and what is on disk. Only after closing or calling `fp.flush()` you can be sure that your data is on disk.

Open a file (modern way) using `with`:

Since Python 2.5 the `with` statement is supported. This statement executes the following body in a secure way, so that the file is always closed, even in case of an error inside the body.

```
with open("say_hi.txt", "w") as fh:
    print("hi", file=fh)
    print("ho", file=fh)
```

```
#show output of say_hi.txt
```

```
hi
ho
```

Reading from a text file

For text files there are two ways to read: `readlines` returns the file line by line in a list of strings:

```
with open("say_hi.txt", "r") as fh:
    print(fh.readlines())
```

```
['hi\n', 'ho\n']
```

Comment: there is also a method called `readline` (no `s` at the end !) which only reads one line. So take care the use the right method name.

What is nice in Python is that you can loop over the lines in a file using `for`:

```
with open("say_hi.txt", "r") as fh:
    for line in fh:
        print(line)
```

```
hi

ho

```

Why those empty lines ? We still can access the latest value of `line`:

```
print(repr(line))
```

```
'ho\n'
```

So `line` also contains the line break `\n` from the file. And as `print` automatically starts a new line when done, we get the empty extra lines.

To get rid of the `\n`:

```
with open("say_hi.txt", "r") as fh:
    for line in fh:
        line = line.rstrip()
        print(line)
```

```
hi
ho
```

Performance tip: use the `for` loop for iterating over a file. For huge files this only reads as much bytes as needed in every iteration and thus works for files which are larger than your computers memory !

Again: if you are not sure what an iterator produces you may use `list` (unless the iterator is infinite):

```
with open("say_hi.txt", "r") as fh:
    print(list(fh))
```

```
['hi\n', 'ho\n']
```

Working with multiple files at the same time:

```
with open("say_hi.txt", "r") as fh_in:
    with open("say_hi_upper.txt", "w") as fh_out:
        for line in fh_in:
            print(line.rstrip().upper(), file=fh_out)
```

```
with open("say_hi.txt", "r") as fh_in, open("say_hi_upper.txt", "w") as fh_out:
        for line in fh_in:
             print(line.rstrip().upper(), file=fh_out)
```

To break a long code line we can use `\` at the end of a line:

```
with open("say_hi.txt", "r") as fh_in,\
     open("say_hi_upper.txt", "w") as fh_out:
        for line in fh_in:
             print(line.rstrip().upper(), file=fh_out)
```

```
#show output of say_hi_upper.txt
```

```
HI
HO
```

Reading and writing csv files

This is not covered in this course, but look at https://pymotw.com/3/csv/index.html (https://pymotw.com/3/csv/index.html) ! If you work with csv files, do not implement your own reader and writer, use the `csv` module, there are some corner cases which are tricky (for example you have a cell which contains a "," or linebreak "\n"), and there are some variations (dialects).

Another option is to install `pandas`, a library for handling so called "data frames". `pandas` can read and write from / to multiple sources, like `csv` and `xlsx` files, but also tables from relational databases.

Reading and writing binary files

For binary files the modes for opening are `"rb"`, `"wb"` and `"ab"` (reading, writing and appending).

In addition to the methods we introduced above the file handle has methods `read` and `write` for interaction. These are mostly used for binary files and not for text files.

Serialisation

Serialisation writes (even nested) data structures in a binary format to a disk. You can recover this data later on easily. In Python serialisation is called "pickling" like conserving vegatables in a jar.

```
# this is a complex data structure:
data = (1, { 1: 2, 3: [1, 2, 3], "s": (1, 2)})
print(data)
```

```
(1, {1: 2, 3: [1, 2, 3], 's': (1, 2)})
```

```
import pickle
with open("data.bin", "wb") as fp:  # open in "write binary" mode
    pickle.dump(data, fp)
```

```
# this is how it looks like on disk:
#show output of data.bin
```

```
K}q(KK]q(KKKeXsqKK qu q.
```

```
# now recover your complex data structure:
with open("data.bin", "rb") as fp:
    recovered = pickle.load(fp)
print(recovered)
print(recovered == data)
```

```
(1, {1: 2, 3: [1, 2, 3], 's': (1, 2)})
True
```

Exercise block 8

1. Repeat the examples above
2. Write a script which writes square numbers `1, 4, 9, ..., 100` line by line to a text file, check the content with your file system explorer then write some code to read the numbers again and compute their product.

Optional exercises

1. Lookup how to use the `csv` module and use it to write a 10 x 10 multiplication table to a csv file
2. Use the same module to read the data from the file again and compute the sum of all entries.

9. Classes

This section assumes that you already had an introduction to object oriented programming in some programming language and will show how classes work in Python.

A class in Python is declared with the `class` statement, methods are defined with `def` like functions.

Remember: a `class` is a "recipe" or "template" for the creation of objects. Or the other way round: a object is an instance of a `class`.

```
import math

class Vector2D:

    def __init__(self, x0, y0):
        self.x = x0     # set attribute x
        self.y = y0     # set attribute y

    def length(self):
        """method which computes length of Vector2D"""
        return math.hypot(self.x, self.y)

# create an instance, this creates the object and
# calls __init__ with args 1 and -1:
p = Vector2D(1, -1)

# now we can access attributes
print(p.x, p.y)

# and call method
print(p.length())
```

```
1 -1
1.4142135623730951
```

- `class Vector2D` provides the name of your class.
- `__init__` it the initializer method which is called if you instantiate `Vector2D(1, -1)`. It sets the attributes `x` and `y`.
- There is no type declaration for attributes, you just set them inside `__init__`.
- There are no rules for "private" or "protected" attributes and methods. A user of a class can access every attribute and method.
- But: it is common practice to use names starting with a **single** `_` for private attributes and methods.
- `self` is the current instance of the object. (In C++ / Java we have `this` instead).
- In Python you must use `self` as the first parameter when you **declare** methods, but you do not provide it if you **call** the method.

Some internals

Methods are attached to the class, so for examle you can access

```
print(Vector2D.__init__)
print(Vector2D.length)
```

```
<function Vector2D.__init__ at 0x10f0d8e18>
<function Vector2D.length at 0x10f0d8bf8>
```

If you call `p.length()` Python looks up the class of `p` which is `Vector2D` and translates this to `Vector2D.length(p)`, so `self` is set to `p` if you call `Vector2D.length` and so the computation works on attributes of `p`.

Special methods

There are many other special methods having names starting and ending with `__` like `__init__`. For example `__str__` is called whenever you convert a object to a string representation. This is the case when you print the object:

```
import math

class Point2D:

    def __init__(self, x0, y0):
        self.x = x0
        self.y = y0

    def length(self):
        return math.hypot(self.x, self.y)

    def __str__(self):
        return "Point2D(x=%s, y=%s, lenght=%s)" % (self.x, self.y, self.length
()) # attribute and method access here !

p = Point2D(1, -1)

# this calls the __str__ method:
print(p)
```

```
Point2D(x=1, y=-1, lenght=1.4142135623730951)
```

There are many other special functions for customizing your objects, see https://docs.python.org/2/reference/datamodel.html#special-method-names (https://docs.python.org/2/reference/datamodel.html#special-method-names)

Inheritance

We want to create a class `Vector2D` which reuses and overwrites methods from `Point2D`, but also adds addition for vectors by implementing the special method `__add__`:

```
class Vector2D(Point2D):

    def __str__(self):
        """ overrides __str__ from Vector2D """
        return "Vector2D(x=%s, y=%s)" % (self.x, self.y)

    def __add__(self, other):
        """ is called when you execute self + other """
        return Vector2D(self.x + other.x, self.y + other.y)

# create instance
v1 = Vector2D(1.0, 2.0)

# access attributes
print(v1.x, v1.y)

# calls __str__:
print(v1)

# calls method in base class !
print(v1.length())

# this is in base class too, see result:
print(v1 + v1)
```

```
1.0 2.0
Vector2D(x=1.0, y=2.0)
2.23606797749979
Vector2D(x=2.0, y=4.0)
```

Exercise block 9

1. Repeat the examples above.
2. Extend `Vector2D` to implement a method `scale` which takes a float `x` and scales the attributes `x` and `y` internally.
3. Implement a method `__mul__` which takes another vector and returns the dot product (scalar product) of both. `__mul__` is called if you use `v1 * v2`.
4. Create a class `ComplexNumber` which inherits `Vector2D` and reimplements `__mul__` for complex arithmethic. Reimplement `__str__` to achieve output in the style of `1 + 2i`.

10. tuple unpacking, enumerate, list comprehensions

Tuple unpacking allows taking values from a tuple without index access, so you need less code and it is often more readable. Instead of writing

```
tp = (1, 2, 3)
a = tp[0]
b = tp[1]
c = tp[2]
print(a + b + c)
```

```
6
```

... you can write:

```
a, b, c = tp
print(a + b + c)
```

```
6
```

You can ommit parantheses for declaring a tuple:

```
a, b, c = 1, 2, 3
print(a + b + c)
```

```
6
```

So: multiple return values of function is nothing else than returning tuples followed by tuple unpacking.

Good style: use `__` for not needed values if you do tuple unpacking. So if you only want to unpack `a` use the following style to avoid declaration of variables you will not use later on:

```
tp = (1, 2, 3)
a, __, __ = tp
```

Tuple unpacking is handy for exchanging values, you need no temporary variables:

```
print(a, b)
a, b = b, a
print(a, b)
```

```
1 2
2 1
```

If this was to fast, here is a more detailed implementation:

```
print(a, b)
tp = (b, a)  # creates tuple
a, b = tp    # unpacks the values and overwrites variables "a" and "b" with new
 values !
print(a, b)
```

```
2 1
1 2
```

Using `zip`

To iterate over two (or more) lists at the same time use `zip` + tuple unpacking after `for`:

```
numbers = [1, 2, 4, 8]
words = ["one", "two", "four", "eight"]

for number, word in zip(numbers, words):
    print(number, word)
```

```
1 one
2 two
4 four
8 eight
```

`zip` also returns an iterator which gives you a tuple in every iteration. Tuple unpacking with `for` now does tuple unpacking for evey tuple created by the `zip` iterator.

```
print(list(zip(numbers, words)))
```

```
[(1, 'one'), (2, 'two'), (4, 'four'), (8, 'eight')]
```

If the iterables passed to `zip` have different lengths, the shortest one determines the result:

```
print(list(zip("abcde", "012", "XY")))
```

```
[('a', '0', 'X'), ('b', '1', 'Y')]
```

Iterating using `enumerate`

If you want to iterate over a list and you want to count at the same time `enumerate` is handy:

```
for i, word in enumerate(words):
    print(i, word)
```

```
0 one
1 two
2 four
3 eight
```

`zip` and `enumerate` work on other iterables (a file handles) as well:

```
with open("say_hi.txt", "r") as fh:
    for i, line in enumerate(fh):
        print("line", i, "is: ", line.rstrip())
```

```
line 0 is:  hi
line 1 is:  ho
```

List comprehensions

List comprehensions allow creation and transformation of lists in a comprehensive and readable way. For example the following two lines ...

```
squares = [i * i for i in range(6)]
print(squares)
```

```
[0, 1, 4, 9, 16, 25]
```

... are equivalent to

```
squares = []
for i in range(6):
    squares.append(i * i)
print(squares)
```

```
[0, 1, 4, 9, 16, 25]
```

But you can filter too:

```
squares_of_odds = [i * i for i in range(6) if i % 2 == 1]
print(squares_of_odds)
```

```
[1, 9, 25]
```

```
squares_of_odds = []
for i in range(6):
    if i % 2 == 1:
        squares_of_odds.append(i * i)
print(squares_of_odds)
```

```
[1, 9, 25]
```

We used `range` only for demonstration, you take any other iterable instead:

```
words = ["hi", "this", "is", "list", "comprehension"]
print([w.upper() for w in words if len(w) % 2 == 0])
```

```
['HI', 'THIS', 'IS', 'LIST']
```

sorting: decorate - sort - undecorate pattern

If you want to sort a list of strings not by their alphabetic order but by their length you can provide a `key` parameter which is a function which indictes the ordering:

```
names = ["python", "programming", "I", "like"]
print(sorted(names))
```

```
['I', 'like', 'programming', 'python']
```

```
print(sorted(names, key=len))
```

```
['I', 'like', 'python', 'programming']
```

Before introducing this `key` parameter in Python 2.4 people used the following strategy which I demonstrate for educational purposes:

```
# decorate:
decorated_names = [(len(s), s) for s in names]
print("decorated names: ", decorated_names)

# tuples are sorted lexicographically:
# the first element defines the ordering

decorated_names.sort()
print("sorted decorated names:", decorated_names)

# undecorate
names = [name for (__, name) in decorated_names]
print("names sorted by length:", names)
```

```
decorated names:  [(6, 'python'), (11, 'programming'), (1, 'I'),
(4, 'like')]
sorted decorated names: [(1, 'I'), (4, 'like'), (6, 'python'), (11,
'programming')]
names sorted by length: ['I', 'like', 'python', 'programming']
```

Exercises

- Transform the list `[2, 3, 5, 7, 11]` to a new list such that the result contains the doubled value of elements from list being smaller than 7.
- Can you use the decorate-sort-undecorate pattern to sort a list of strings without considering their case. ?

11. try / except / finally

```
x = 1 / 0
```

```
---------------------------------------------------------------------
--------
ZeroDivisionError                         Traceback (most recent ca
ll last)
<ipython-input-179-f9f847a0a080> in <module>()
----> 1 x = 1 / 0

ZeroDivisionError: division by zero
```

Catch exceptions:

```
def divide(a, b):
    return a / b

try:
    x = divide(3, 2)
    y = divide(7, 0)
except ZeroDivisionError:
    print("oops")
```

```
oops
```

You can raise your own exceptions:

```
def fun(number):
    if number < 0.0:
        raise Exception("{} is negative! ".format(number))
    return number
```

```
print(fun(1.0))
```

```
1.0
```

```
# read the output below line by line !!!
print(fun(-1.0))
```

```
---------------------------------------------------------------------
--------
Exception                                 Traceback (most recent ca
ll last)
<ipython-input-183-c513a8d426e1> in <module>()
      1 # read the output below line by line !!!
----> 2 print(fun(-1.0))

<ipython-input-181-98fe1eee559e> in fun(number)
      1 def fun(number):
      2     if number < 0.0:
----> 3         raise Exception("{} is negative! ".format(number))
      4     return number

Exception: -1.0 is negative!
```

```
def is_float(string):
    try:
        float(string)
        return True
    except ValueError:
        return False

print(is_float("1.2"))
print(is_float("1.ab"))
```

```
True
False
```

see https://docs.python.org/3/tutorial/errors.html (https://docs.python.org/3/tutorial/errors.html)

Some topics not covered in the course

https://www.digitalocean.com/community/tutorials/how-to-use-args-and-kwargs-in-python-3 (https://www.digitalocean.com/community/tutorials/how-to-use-args-and-kwargs-in-python-3)

http://www.python-course.eu/python3_lambda.php (http://www.python-course.eu/python3_lambda.php)

slicing with stepsize: http://www.pythoncentral.io/how-to-slice-listsarrays-and-tuples-in-python/ (http://www.pythoncentral.io/how-to-slice-listsarrays-and-tuples-in-python/)

generators and `yield` statement: https://realpython.com/blog/python/introduction-to-python-generators/ (https://realpython.com/blog/python/introduction-to-python-generators/)

Other resources ¶

https://siscourses.ethz.ch/python_one_day/reference.html (https://siscourses.ethz.ch/python_one_day/reference.html)

Within ETH network (or via VPN) you can use http://proquest.safaribooksonline.com/ (http://proquest.safaribooksonline.com/) to access book like:

- http://shop.oreilly.com/product/0636920028338.do (http://shop.oreilly.com/product/0636920028338.do)
- http://shop.oreilly.com/product/0636920028154.do (http://shop.oreilly.com/product/0636920028154.do)
- ADVANCED: http://shop.oreilly.com/product/0636920032519.do (http://shop.oreilly.com/product/0636920032519.do)

```
/Users/uweschmitt/Projects/python-course-one-day/lib/python3.5/site
-packages/ipykernel_launcher.py:9: UserWarning: get_ipython_dir has
moved to the IPython.paths module since IPython 4.0.
  if __name__ == '__main__':
```