Python Programming

Contents

- Python intro and history
- Python.org
- Python versions
- Python Standard Library
- Python flavours
- Cpython
- Python standard distribution
- Anaconda Distribution
- Python IDE's
- pyCharm
- Python data types
- Number
- String
- List
- List comprehension
- Tuple
- Range
- Set
- Set comprehension
- Dict
- Dict comprehension
- Bool
- Python control flow statements
- Modules and packages
- Dunder elements
- Functions
- The builtin functions map and filter.
- Exceptions
- Object Oriented Programming
- Classes

Contents

- Inheritance
- Some modules from the Python Standard Library
- File I/O
- Requests
- JSON
- Database access

Python intro

- Python is a language created by Guido van Rossum at the start of the 1990's.
- Python is
  - an interpreted language.
  - a dynamic language.
  - an object oriented language.
    - all data in Python are objects.

- The original and default installation uses the C language as the language "behind".
  - The interpreter in the standard installation (CPython) is written in C.

Python implementations

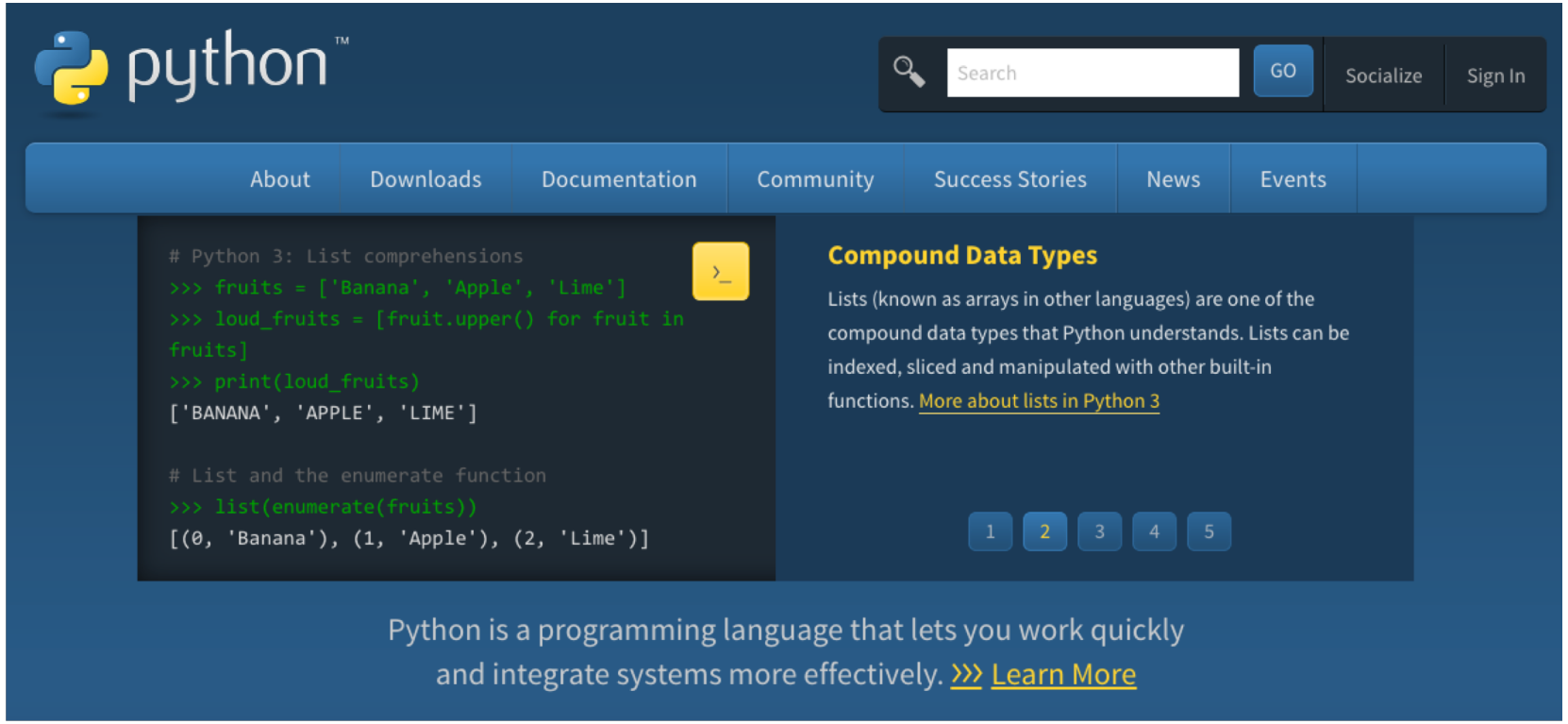
- Not only C is used as language "behind"
  - Jython
    - uses Java
  - IronPython
    - uses the .NET platform
  - Skulpt
    - JavaScript implementation
- Other Python flavours
  - PyPy
    - Just In Time compiled version of Python
    - Fast
  - MicroPython
    - Python implementation optimized to run on microcontrollers
  - IPython
    - Interactive Python

Python versions

- Not all language constructs in the original Python setup were correct and efficient.
- Changing them would mean incompatibility with earlier versions.
- While Python was in it's 2 series a decision was made to change a number of language constructs and put this changes in the 3 series.
- Part of the developer base still use version 2.x because existing tools and platforms are written in 2.x.
- Version 2.x is still supported but this will stop in 2020.
- In this Python course we are using 3.x.

Python community

- One of the strong points of Python is it's community of developers. The main website is Python.org



Documentation

- On Python.org extensive documentation about Python can be found.

Python standard library

- Python has a number of built-in types and functions but a large part of Pythons functionality is contained in modules which form a part of the standard implementations.
- They form the Python standard library.
- There are more than 200 modules in the standard library
- Modules contained in the standard library are i.e.

math
statistics
os
zlib
io
xml
datetime

urllib
json
shelve
pickle
and much more

See: https://docs.python.org/3/py-modindex.html

Python community

- A very important man in this community is Guido van Rossum himself.
- His title was BDFL (Benevolent Dictator For Life).
  - in the end he had to decide about changes and additions.
  - but in July 2018 he decided to retire from this title.
- Changes are proposed in the Python Enhancement Proposals (PEP).
- A steering council of 10 to 15 of Python developers take the decisions on the future of python.
- A very well known PEP is PEP8 "The style guide for Python code".
- Also PEP20, The Zen of Python, is famous.
  - use import this

PEP20

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
```

continues on the next sheet…

PEP20

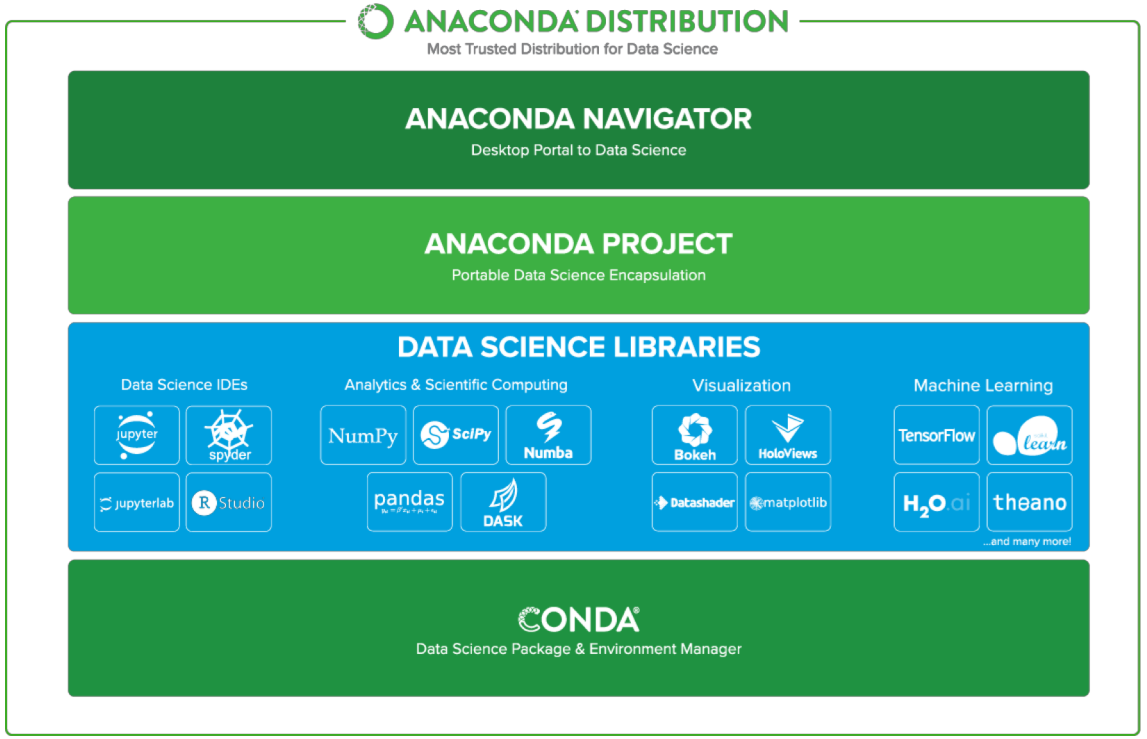
```
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Installing Python

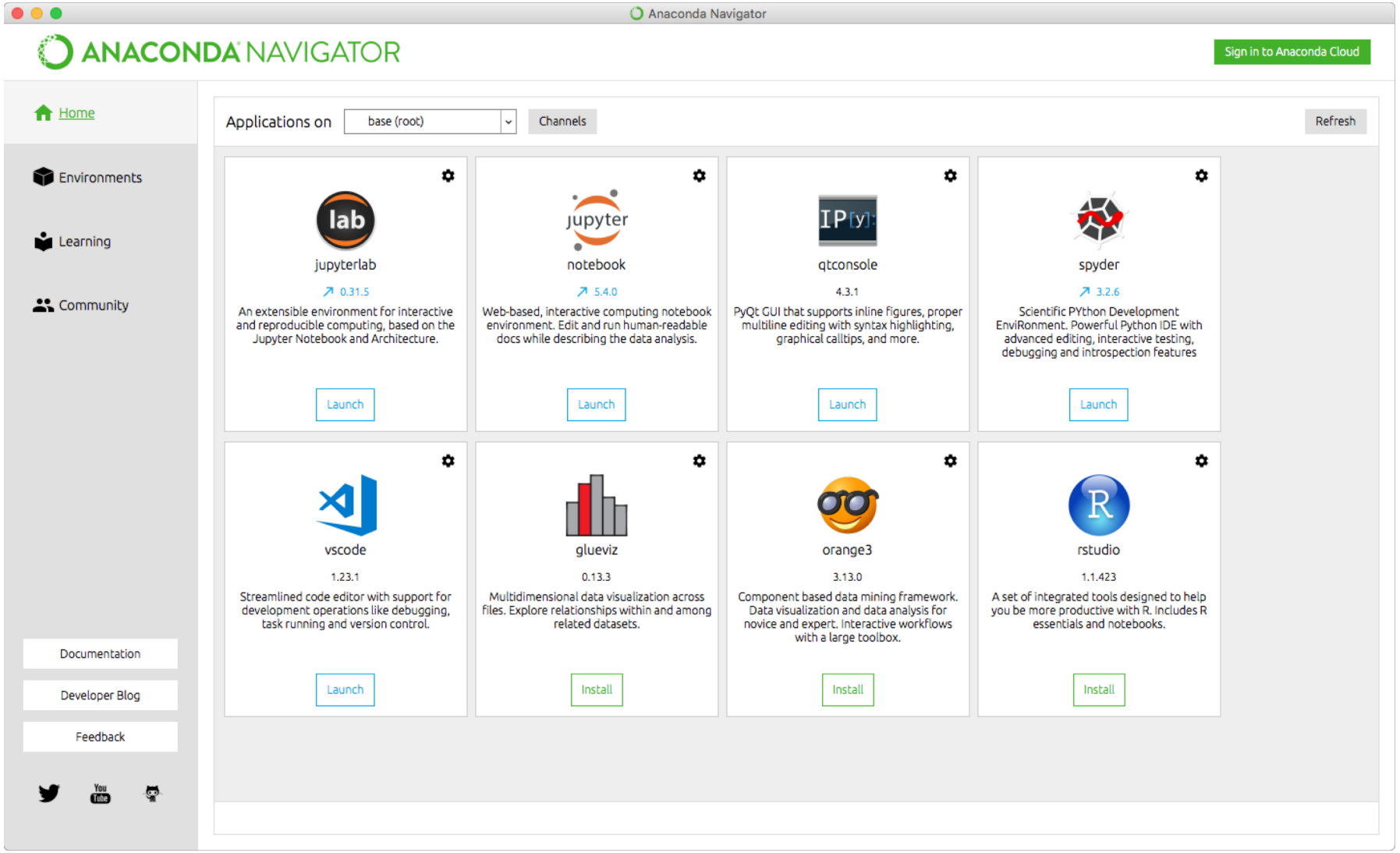
- A version from Python can be downloaded from the Python.org site.
- After downloading is completed Python can be installed by running the executable.
  - Make sure the option "Add Python to PATH" is checked.
- Typing the word Python in a command window will show the Python shell.
- With the installation of Python also IDLE, an Integrated Development and Learning Environment, becomes available.

Anaconda platform

- A Python distribution especially aimed at data science.
- Published by the Anaconda company.
- Includes
  - Python platform.
  - Anaconda Navigator.
  - Jupyter notebook.
  - Spyder
  - and more



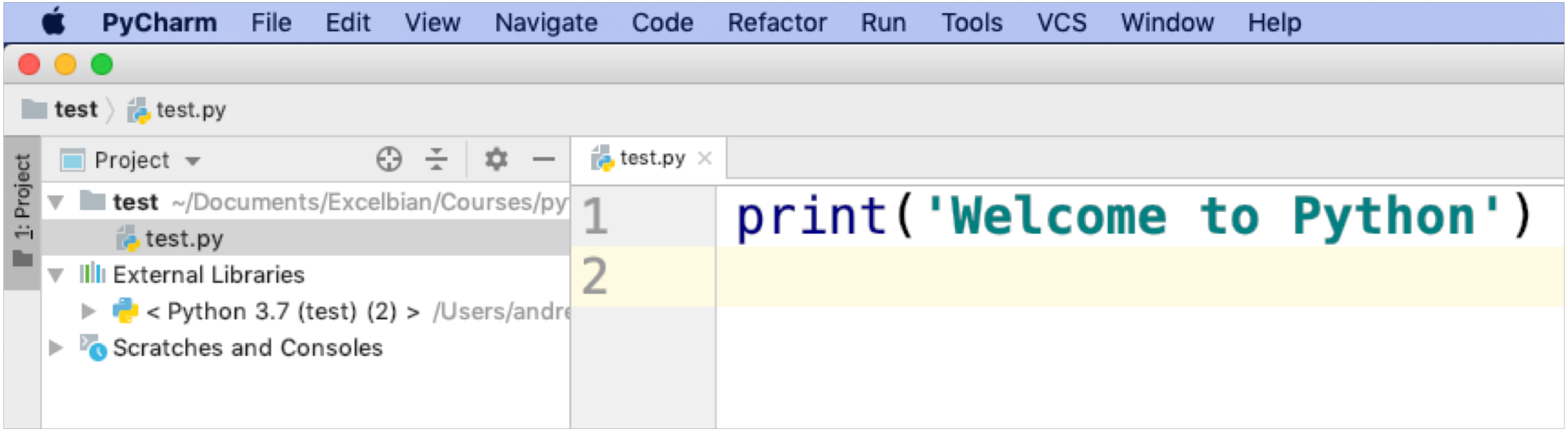
Anaconda Navigator



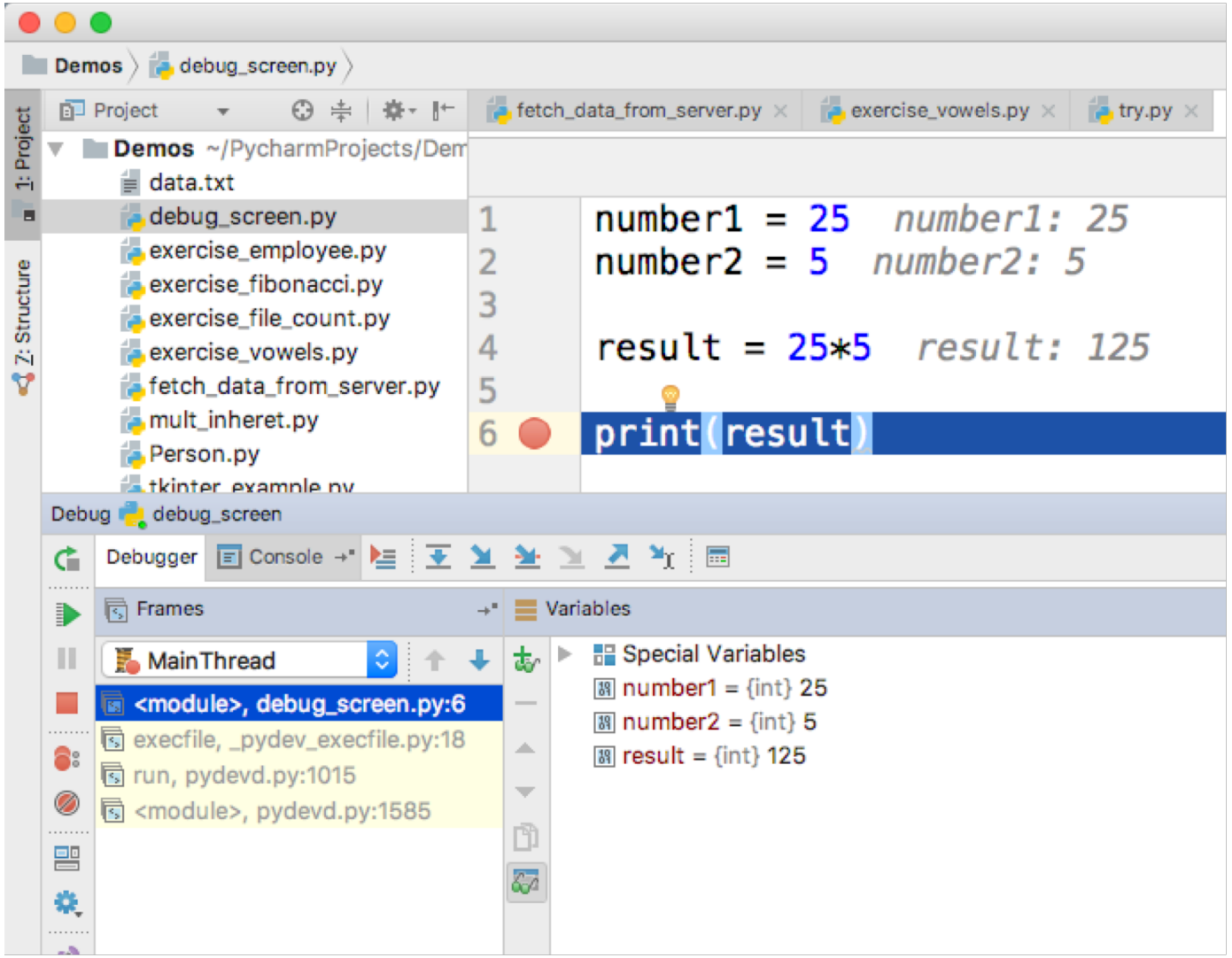
pyCharm Community edition

- pyCharm offers a complete professional development environment with
  - Debug facilities.
  - Code completion.
  - Code syntax checking.
  - Refactoring.
  - Multiple Python consoles.
  - Support for versioning tools.
  - Analysis tools.
  - And much more.

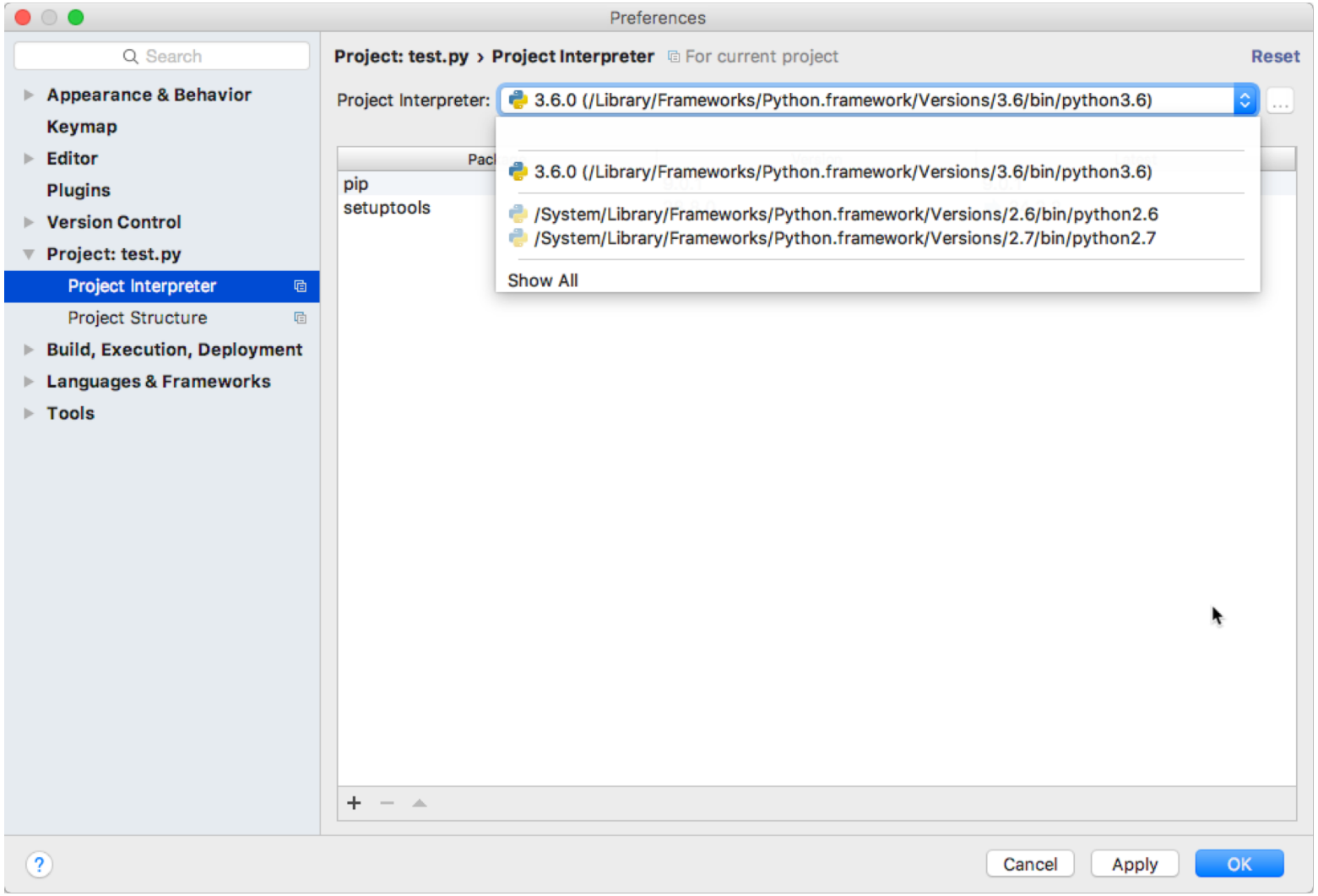
pyCharm editor perspective



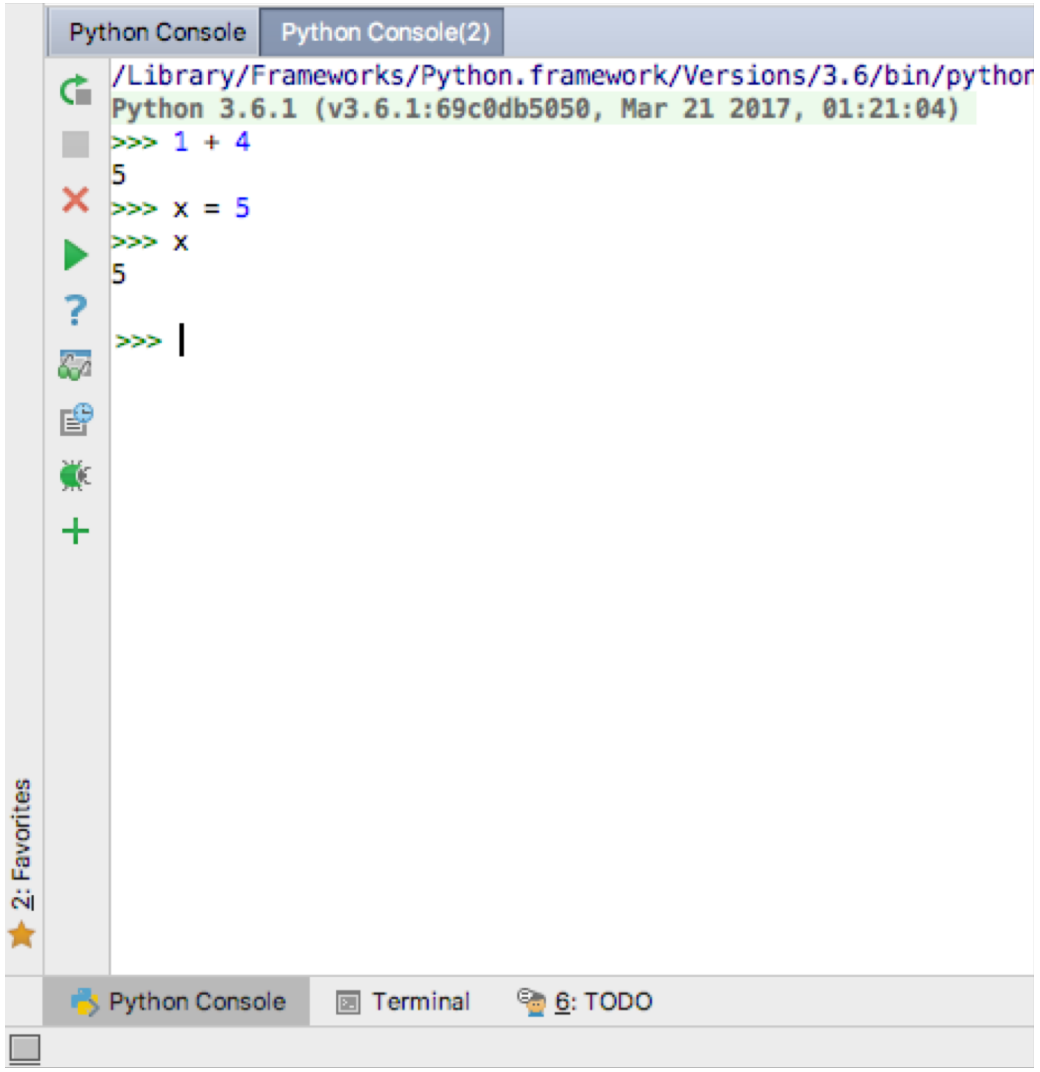
pyCharm debug perspective



pyCharm: setting the correct Python version



PyCharm: Phyton consoles



Exercise

- Create a project in PyCharm and and add a Python module/file.
- Create in the module a variable first with the value 12 and a a variable second with the value 40.
- Put the addition of both variables in the variable result
- Print the result.
- Do the same in the Python console in PyCharm.

Python syntax

- Statements are terminated by a line feed.
- Multiple lines can be spanned with the \ character.

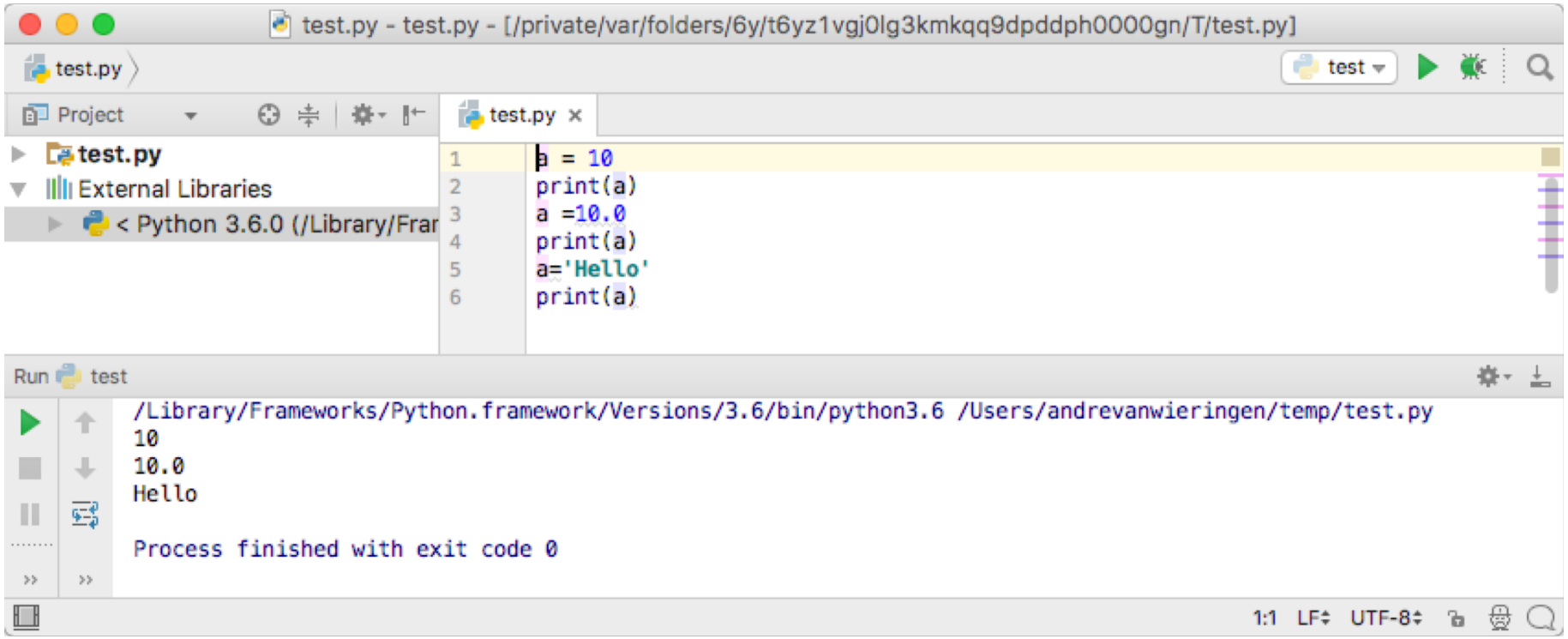
```
a = b + \
    c
```

- A colon : and indentation is used to denote different blocks of code.

```
if a:
    statement1
    statement2
else:
    statement3
```

Variables

- Python is a dynamically typed language where variable names are bound to different values, possibly of varying types, during program execution.



Identifiers and keywords

- An identifier is used to identify variables, functions, classes, modules etc.
- Identifiers can include letters, numbers and the underscore character _.
- Identifiers must start with a nonnumeric character.
- Identifiers are case sensitive.
- Identifiers starting or ending with underscores often have a special meaning.
- Identifiers with leading and trailing double underscores such as __init__ are reserved for special attributes and methods.
  - They are called dunder methods (double underscore methods).

Reserved words

- Python has the following reserved words

```
False       elif        lambda
None        else        nonlocal
True        except      not
and         finally     or
as          for         pass
assert      from        raise
break       global      return
class       if          try
continue    import      while
def         in          with
del         is          yield
```

Python concepts

- Programs are composed of modules.
- Modules contain statements.
- Statements contain expressions.
- Expressions create and process objects.
- All data in Python are objects.
- Objects have attributes (data and methods).
- Objects are constructed.
- Python contains built-in objects.

Importing modules

- Other files can be imported in Python with the statement
  - import module_name
- Import statements are put at the top of a file.

```
import math
import random

print('Pi is :',math.pi)
print('A random number :',random.random())
```

```
Pi is : 3.141592653589793
A random number : 0.57820153661132
```

Module example

my_module.py

```
my_variable = 100


def addition(plus):
    print('The result after addition is :', my_variable + plus)
```

another_module.py

```
import my_module

print('The variable is :',
my_module.my_variable)

my_module.addition(50)
```

```
The variable is : 100
The result after addition is : 150
```

Importing modules

- It is possible to change the name of the module internally in the importing file

```
import math as m

print('The sinus of 0 is :', m.sin(0))
```

```
The sinus of 0 is : 0.0
```

- To import only certain elements from a module use the statement
  - from module_name import name

```
from math import cos

print('The cosinus of 0 is :', cos(0))
```

```
The cosinus of 0 is : 1.0
```

Importing modules

- It is also possible to import all names from a module with a wildcard notation.

```
from my_module import *

print('The variable is :', my_variable)

addition(50)
```

Exercise

- Create a module with name my_module.py in PyCharm
- The contents of the module is
  - x = 30
  - y = 50
- import the module in a module name main.py
- In this module print the result when adding x and y.
- Now add the statement print(x+y) in my_module.py and run main.py again.

Python virtual environment

- Python application use modules and packages (sets of modules) which are not part of the standard library.
- If nothing is done these modules and packages are stored in the same directory as the standard packages.
- Using a virtual environment creates a separate directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages.
- Virtual environments can be created using the Anaconda GUI.
- They are also created automatically when creating a Pyton project in PyCharm.
- It is also possible to create and activate a virtual environment with Python or Anaconda on the command line

pip

- With pip (Pip Installs Packages) you can easily install packages which are not part of the standard library.
- The basic usage of pip is

```
python -m pip install some_package
```

- It’s also possible to specify an exact or minimum version directly on the command line.

```
python -m pip install some_package==1.0.4
python -m pip install "some_package>=1.0.4"
```

- Upgrading existing modules must be requested explicitly:

```
python -m pip install --upgrade some_package
```

conda

- The package manager for the Anaconda distribution is conda.
- Install a package or module with conda install ….
- Create a virtual environment either in the anaconda navigator or with

  conda create .. -n myenv
- Activate the virtual environment with conda activate myenv.
- Get a list of your virtual environments with conda env list

		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Important Python built-in types

- Numbers
  - int
  - float
  - complex
- Text sequence
  - str
- Other sequences
  - tuple
  - list
  - range
- Set
  - set
  - frozenset
- Mapping
  - dict
- Boolean
  - bool

Examples of built-in types

- Numbers
  - 143.2
- Strings
  - 'Hello'
- Tuples
  - (2,'h')
- Lists
  - [1,5,'b']
- Range
  - range(10)
- Dictionaries
  - {'first':5,'second':9}
- Sets
  - {5,45,9}
- Bool
  - True, False

Numbers

- The built-in numeric types in Python are
  - integers
  - floating-point numbers
  - complex numbers
    - Numbers with a real part and an imaginary part
- Numbers are immutable objects
- Performing an arithmetic operation on a number produces a new number.
- Some numeric literals
  - 200
  - 200.35
  - 200 + 30J
- Numerical literals can include single underscore characters to help in reading the literal
  - 1000_000_000

Integer numbers

- Integer literals can be decimal, binary, octal, or hexadecimal
  - decimal – sequence of digits, first digit nonzero.
  - binary – starts with 0b followed by a sequence of binary digits.
  - octal – starts with 0o followed by a sequence of octal digits.
  - hexadecimal – starts with 0x followed by hexadecimal digits.
- Examples
  - 2468
  - 0b01110
  - 0o45
  - 0x2AF

Floating-Point numbers.

- A floating-point number is a sequence of decimal digits which includes a decimal point.
- It can contain an exponent suffix, e or E.
- Examples
  - 2.33
  - 2.e3 (2000.0)
  - 2.E-3 (0.002)

Complex numbers

- A complex number consists of two floating-point values, for the real part and the imaginary part.
- The imaginary part is followed by the character j or J.
- j is the square root of -1.

```
z = 2 + 3j
```

- The real part of z can be obtained by z.real.
- The imaginary part by z.imag.
- Examples

```
z1 = 4+5j
z2 = 4-5j
print(z1*z2)                (41 + 0j)
```

Strings and string literals

- A sequence of characters
- Strings are immutable
  - An operation on a string always produces a new string.
- String literals specify a sequence of characters.
- They are define using single ', double " and triple ''' or """ quotes.
- Single and double quoted strings are restricted to one line.
- When the \ is used the string can be continued on the next line.
- A string literal delimited with triple quotes can can consist out of more than one line.
- The backslash \ character within a string makes it possible to use characters with a special meaning e.g. \n, the newline character

String literal examples

```
s1 = 'A string which continues \
    on the next line'
s2 = 'A string with a\n linefeed'
s3 = '''A string
      spanning
      more lines'''
s4 = 'A string with a back slash \\'
print(s1)
print(s2)
print(s3)
print(s4)
```

```
A string which continues     on the next line
A string with a
 linefeed
A string
      spanning
      more lines
A string with a back slash \
```

Raw strings

- When a string starts with r or R it is a raw string.
- In a raw string escape sequences are not interpreted but used as is.
- This is handy for e.g. path's with back slashes (Windows notation).

```
dir_name = r'c:\aDirectory'
```

String methods

- The class string has a number of string methods.
- Important methods are
- find()
  - finds a substring in a string. Returns the index where the substring starts or -1.
- isdigit()
  - returns true of the string is a digit.
- join()
  - returns a string which is the concatenation of the strings in the iterable. The string on which the join is called is the separator.

```
s = '-'.join(['Hans', 'Robert', 'Leo'])
print(s)
```

```
Hans-Robert-Leo
```

More string methods

- lower()
  - returns a string with all characters converted to lowercase.
- upper()
  - returns a string with all characters converted to uppercase.
- rstrip()
  - removes whitespace characters at the right side
  - also available lstrip,strip.
- split()
  - splits a string on a delimiter and returns a list.

```
l = 'H,e,l,l,o'.split(',')
print(l)                          ['H', 'e', 'l', 'l', 'o']
```

String methods examples

```
s = 'hello'
print(s.find('lo'))
print(s.find('kl'))
print('3'.isdigit())
print('    hello'.lstrip())
print(s.upper().find('lo'))
print(s)
print(s.upper().find('LO'))
s = 'Hello world!'
print(s.split(' '))
```

```
3
-1
True
hello
-1
hello
3
['Hello', 'world!']
```

Exercise

- Make lower case characters from the upper case characters and remove all white space characters from the string "There is something rotten in the state of Denmark"

- Change the string "python is a wonderful language" such that each word starts with a capital character.

String formatting

- String formatting can be done with
  - String formatting expression, using %
  - String formatting method, using {} and .format
  - String interpolation.

String formatting expression

```
print('The sun is %s' % 'shining')
print('Her name is %s and her age is %d' % ('Mary', 35))
print('%f %.2f' % (1 / 3, 1 / 3))
```

```
The sun is shining
Her name is Mary and her age is 35
0.333333 0.33
```

- Some string formatting codes

s= string, c= character, d=decimal, o=octal, x=hex, f = float

String formatting method

```
print('Her name is {0} and her age is {1}'.format('Mary', 35))
print('Her age is {1} and her name is {0}'.format('Mary', 35))
print('Her name is {} and her age is {}'.format('Mary', 35))

print('{0:f},{1:8.2f}'.format(1 / 3, 1 / 7))
```

```
Her name is Mary and her age is 35
Her age is 35 and her name is Mary
Her name is Mary and her age is 35
0.333333,    0.14
```

String interpolation

- In string interpolation the name of a variable can be used as a placeholder in the literal string.
- For this the literal has to be preceded with the character f (format).
- Operations on the variable can be done in place.

```
name = 'Fred'
age = 50
s1 = f'My name is {name}, my age next year is {age+1}'
print(s1)
```

```
My name is Fred, my age next year is 51
```

Lists

- A list is a mutable ordered sequence of items.
- The items are objects and they may be of different type.
- The notation is [] with the items separated by a comma.

```
s = ['Hello', 10, 3.3]
print(s)
s = [10]
print(s)
s = []
print(s)
s = list('Hello')
print(s)
s = list()
print(s)
```

```
['Hello', 10, 3.3]
[10]
[]
['H', 'e', 'l', 'l', 'o']
[]
```

Working with lists

```
print([1, 2] + [3, 4])
s = [3, 'Hello', 5.4]
print(s[0])
print(s[1])
s[2] = 15.9
print(s)
print([1, 2] * 3)
```

```
[1, 2, 3, 4]
3
Hello
[3, 'Hello', 15.9]
[1, 2, 1, 2, 1, 2]
```

List methods

- append()
  - add object at the end of the list
- pop()
  - pops an object from the list at a certain index
- insert()
  - inserts an object at a certain position

```
l = [1, 3, 5, 2, 7, 5, 5, 4, ]
l.append(20)
print(l)
print(l.pop(4))
l.insert(1, 12)
print(l)
```

```
[1, 3, 5, 2, 7, 5, 5, 4, 20]
7
[1, 12, 3, 5, 2, 5, 5, 4, 20]
```

More list methods

- remove()
  - remove an object at value
- extend()
  - add multiple items at the end of the list
- reverse()
  - reverses the list
- sort()
  - sorts the list in ascending order

```
l.remove(5)
print(l)
l.extend([10, 14])
print(l)
l.reverse()
print(l)
l.sort()
print(l)
```

```
[1, 12, 3, 2, 5, 5, 4, 20]
[1, 12, 3, 2, 5, 5, 4, 20, 10, 14]
[14, 10, 20, 4, 5, 5, 2, 3, 12, 1]
[1, 2, 3, 4, 5, 5, 10, 12, 14, 20]
```

Tuples

- A tuple is an immutable ordered sequence of items.
- A tuple is denoted with ().
- The items are objects and they may be of different type
- The items are separated by a comma.

```
a = (3, 'Hello', 5)
print(a)
b = 2, 4, 7
print(b)
c = 12,
print(c)
d = (12,)
print(d)
```

```
e = ()
print(e)
f = tuple()
print(f)
g = tuple('Hello')
print(g)
```

```
(3, 'Hello', 5)
(2, 4, 7)
(12,)
(12,)
()
()
('H', 'e', 'l', 'l', 'o')
```

Working with tuples

```
print((1, 2) + (3, 4))
print((10, 20) * 3)
t = (14, 5, 28, 9)
print(t[0])
print(t[3])
print(t[4])
```

```
(1, 2, 3, 4)
(10, 20, 10, 20, 10, 20)
14
9

print(t[4])

IndexError: tuple index out of range
```

Tuple methods

- A number of functions and methods are available for tuples

```
t = (5, 2, 4, 10, 4, 15, 4, 12)
print(len(t))
print(min(t))
print(max(t))
print(t.count(4))
print(t.index(4))
print(t.index(4, 3, 5))
```

```
8
2
15
3
2
4
```

- len, min and max are built-in functions.

Exercise

- Write statements to convert a string into a tuple.
- Write statements to convert the tuple back to a string.
- Create a tuple of 10 elements. Read the third element of the tuple and the third element from last.
- Write statement to change the first element in a tuple.

Expressions and Operators

- An expression is a combination of numbers (or other objects) and operators that computes a value when executed by Python.
- Python has the following arithmetic operators.
  - +, -, * have the meaning we expect
  - ** to the power of
  - / true division
  - // floor by division
  - % remainder by division

Operator examples

```
>>> 2 * 10
20
>>> 2 + 12
14
>>> -9 - 10
-19
>>> 20 / 6
3.3333333333333335
>>> 20 // 6
3
>>> 20 % 6
2
>>> 2 ** 4
16
```

Sequences

- Sequences
  - A sequence is an ordered container of items.
  - Sequences identify the individual elements by position. Position numbers start with zero.
- Sequence types are
  - string
  - tuple
  - list

Sequences and their operations

- Sequences are ordered containers with items that are accessible by indexing and slicing.
- For sequences a number of functions and methods are available.
  - e.g The len function to get the length of the sequence.
- Also indexing expressions are available to fetch elements.
- String, tuples and lists are sequences and thus all operations, functions and methods of sequences are also available to these three types.

Indexing

```
s = [1, 'a', 3.4, 12]
print(s[0])
print(s[-1])
print(s[-2])
print(s[4])
```

```
1
12
3.4
```

```
t = (3, 'hello', 50)
print(t[0])
print(t[1])
print(t[-1])
```

```
3
hello
50
```

```
s = 'World'
print(s[0])
print(s[-1])
print(s[len(s) - 1])
```

```
W
d
d
```

```
Traceback (most recent call last):
  File "/demos/indexing.py", line 5, in <module>
    print(s[4])
IndexError: list index out of range
```

Slicing a sequence

- Slicing is a way to extract an entire section in a single step.
- The general form of a slice is x[i:j].
- It returns the elements of x from i upto j.

```
s = 'Hello world!'
print(s[1:4])
print(s[4:])
print(s[-2:])
print(s[1:50])
```

```
ell
o world!
d!
ello world!
```

Sequence concatenation and repetition

- Sequences can be concatenated with the + sign.
- Two sequences are then joined into a new sequence.
- Also repetition is possible by using the * sign.
- A new sequence is created by repetition, the old sequence is not changed.

```
s = (1, 2, 3,)
t = (4, 5, 6,)
print(s + t)        (1, 2, 3, 4, 5, 6)
print(s * 4)        (1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Immutability

- Numbers, strings and tuples are immutable.
  - They can't be changed in place.
- Lists, and we will see later, dictionaries and sets are mutable.
  - They can be changed in place.

```
s = 'Hello world!'
s[0] = 'h'
```

```
l = list(s)
l[0] = 'h'
print(''.join(l))
```

```
hello world!
```

```
Traceback (most recent call last):
 File "/demos/immutability.py", line 2, in <module>
 s[0] = 'h'
TypeError: 'str' object does not support item assignment
```

List comprehensions

- With list comprehensions we can create lists from other sequences in a very natural way.

```
l = [1, 2, 3, 4, 5, 6, ]
m = [n ** 2 for n in l]
print(m)
k = [n for n in l if n % 2 == 0]
print(k)
```

```
[1, 4, 9, 16, 25, 36]
[2, 4, 6]
```

Unpacking a sequence

- A sequence can be unpacked into variables with an assignment operation.

```
t = (2, 3)
a, b = t
print('a is :', a)
print('b is :', b)

data = ['Hello', 3, 80.41, (1, 2, 3)]
word, index, amount, element = data
print(word, index, amount, element)
```

```
a is : 2
b is : 3
Hello 3 80.41 (1, 2, 3)
```

Discarding items while unpacking

- Sometimes not all elements of a sequence need to be unpacked.
- There is no special syntax for this but it is custom to use the throw away variable _.

```
data = ['Hello', 3, 80.41, (1, 2, 3)]
_, index, amount, _ = data
print(index, amount)
```

```
3 80.41
```

The star expression

- With the star expression more than one element can be unpacked in a variable. The variable becomes a list.

```
record = ('Hans', 'Drupsteen', '0101234567', '061234567')
firstName, lastName, *phoneNumbers = record
print(firstName, lastName, phoneNumbers)
```

```
Hans Drupsteen ['0101234567', '061234567']
```

The star expression

- With this technique it also possible to just unpack the first and last element disregarding the length of the original sequence

```
some_data = ['first_element', 1, 2, 34, 5, 11, 'last_element']
first, *_, last = some_data
print(first, last)
```

```
first_element last_element
```

The range type

- The range type represents an immutable sequence of numbers.
- A range can be created by the built-in function range
  - range(stop)
  - range(start, stop, [step])
- start, stop, step must be integers.
- If the step argument is omitted the step is 1

The range type

- For a positive step the contents of range r are determined with
  r[i] = start+ step * i, i >= 0 and r[i] < stop.

- For a negative step the contents are
  r[i] = start+ step * i, i >= 0 and r[i] > stop.

- A range can use indices
  - r[3]
  - r[-1] is the last element in the range, r[-2] the element before the last etc.

Range examples

```
r = range(10)
print(r)
print(r[0])
print(r[-1])
r = range(1, 10)
print(r)
print(r[0])
print(r[10])
```

```
range(0, 10)
0
9
range(1, 10)
1


File "/demos/range_examples.py", line 8, in <module>
print(r[10])
IndexError: range object index out of range
```

Range examples

```
r = range(1, 20, 2)
print(r)
print(list(r))
r = range(0, -20, -2)
print(r)
print(list(r))
r = range(0, 20, -2)
print(list(r))
```

```
range(1, 20, 2)
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
range(0, -20, -2)
[0, -2, -4, -6, -8, -10, -12, -14, -16, -18]
[]
```

Sets

- A set represents an unordered collection of unique elements.
- There are two built-in types
  - set – mutable.
  - frozenset – immutable.

Set examples

```
a = {42, 12.9, 'world'}
print(a)                                {'world', 42, 12.9}
a = {304}
print(a)                                {304}
a = set()
print(a)                                set()
a = set('Hello')
print(a)                                {'l', 'H', 'e', 'o'}
a.add('!')
print(a)                                {'H', '!', 'o', 'l', 'e'}
a = frozenset('Hello')
print(a)                                frozenset({'l', 'H', 'e', 'o'})
a.add('!')
```

```
File "/demos/set_examples.py", line 13, in <module>
a.add('!')
AttributeError: 'frozenset' object has no attribute 'add'
```

Some set operations

- For set and frozenset
  - issubset(t)
    - test whether every element in s is in t
  - s.issuperset(t)
    - test whether every element in t is in s
  - s.union(t)
    - new set with elements from both s and t
- For set
  - update(t)
    - return set s with elements added from t
  - discard(x)
    - removes x from set s if present

```
s = {1, 4, 6, 2}
print(s)
t = {5, 4}
print(s.issubset(t))
print(s.issuperset(t))
print(s.union(t))
s.discard(5)
print(s)
```

```
{1, 2, 4, 6}
False
False
{1, 2, 4, 5, 6}
{1, 2, 4, 6}
```

Special notation for set operations

```
s = {1, 4, 6, 2}
t = {5, 4}

print(s < t)  # is proper subset
print(s <= t)  # is subset
print(s | t)  # union
print(s & t)  # intersection
print(s - t)  # difference
print(s ^ t)  # symmetric difference
```

```
False
False
{1, 2, 4, 5, 6}
{4}
{1, 2, 6}
{1, 2, 5, 6}
```

Set comprehension

- As for lists it is possible to use comprehensions for sets

```
result = {x * x for x in range(-9, 10)}
print(result)
```

```
{64, 1, 0, 36, 4, 9, 16, 81, 49, 25}
```

- And with a condition

```
result = {x * x for x in range(-9, 10) if x % 2 == 0}
print(result)
```

```
{64, 0, 36, 4, 16}
```

Exercise

- Create a list of 20 numbers.
- Each number is a random integer between 0 and 9, 9 included.
- Create a new list with the unique numbers in the list.
- Create a new list with the unique numbers but keep the original ordering.

Dictionaries

- A dictionary is an arbitrary collection of objects indexed by arbitrary values called keys.
- Dictionaries are mutable.
- Before Python 3.6 dictionaries were not ordered.
- Since Python 3.6 they are ordered with the ordering they are created in.

Dictionary examples

```
d = {'e': 12, 'f': 4, 'x': 400}
print(d)
d = {6: 2, 9: 11}
print(d)
d = dict()
print(d)
d = dict(g=3, w=12.5, z='hello')
print(d)
d = dict([(2, 3), (8, 4)])
print(d)
d = dict(((3, 8), (9, 15)))
print(d)
```

```
{'e': 12, 'f': 4, 'x': 400}

{6: 2, 9: 11}

{}

{'g': 3, 'w': 12.5, 'z': 'hello'}

{2: 3, 8: 4}

{3: 8, 9: 15}
```

Indexing dictionaries

- The value in a dictionary can be fetched, changed and added using the index notation.

```
d = {'a': 1, 'b': 2, 'c': 3}
print(d['a'])
d['c'] = 20
print(d)
d['k'] = 100
print(d)
```

```
1
{'a': 1, 'b': 2, 'c': 20}
{'a': 1, 'b': 2, 'c': 20, 'k': 100}
```

Dictionary methods

- keys()
- values()
- items()
- pop(key)
  - remove the element with the key
- update(d)
  - merges two dictionaries
- get(key)
  - gets the value for the key

```
d = {'a': 4, 'g': 8}
print(d)                    {'a': 4, 'g': 8}
print(d.keys())             dict_keys(['a', 'g'])
print(d.values())           dict_values([4, 8])
print(d.items())            dict_items([('a', 4), ('g', 8)])
d.pop('a')
print(d)                    {'g': 8}
d2 = {3: 9, 5: 1, 'g': 500}
d.update(d2)
print(d)                    {'g': 500, 3: 9, 5: 1}
print(d.get('g'))           500
```

Dictionary comprehension

- Also for dictionaries comprehensions can be used

```
d = {x: x ** 2 for x in range(5)}
print(d)
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

- and with a condition

```
d = {x: x ** 2 for x in range(5) if x % 2 == 0}
print(d)
```

```
{0: 0, 2: 4, 4: 16}
```

Exercise

- Create three dictionaries and merge them into a new one.

None

- None denotes the null object.
- There is only one None object.
- None can be used as a placeholder to indicate that a variable is not referencing any object.

```
a = None
print(a)                    None
a = [None]
print(a)                    [None]
a = a * 4
print(a)                    [None, None, None, None]
```

Boolean values

- Any value in Python can be used as a truth value.
- Any none zero number or nonempty container is true.
- 0, None, an empty container is false.
- bool has two values True, False
  - with a string representation 'True', 'False' and numerical values 1 and 0.

```
x = 1
if x:
    y = 2
else:
    y = 3
print(y)
```

2

Compound statements

- Compound statements contain other statements
- They affect or control the execution of the other statements.
- Important compound statements are
  - if
  - while
  - for
- and for later discussion
  - try
  - with
  - function definition
  - class definition

The if statement

- General format

```
if test:
        statements
elif test1:
        statements1
else:
        statements2
```

```
grade = 'E'
if grade == 'A':
    print('Excellent')
elif grade == 'B':
    print('Good')
elif grade == 'C':
    print('Sufficient')
else:
    print('Not good')
```

Not good

The while statement

- General format

```
while test:
    statements
else:
   statements1
```

```
counter = 0
while counter < 10:
    print(counter, end=',')
    counter += 1
else:
    print('Counter too large')
```

0,1,2,3,4,5,6,7,8,9,Counter too large

The for statement

- General format

```
for element in list_of_elements:
    statements
else:
   statements1
```

```
for i in range(10):
    print(i, end=',')
else:
    print('Out of range')
```

0,1,2,3,4,5,6,7,8,9,Out of range

The break statement

- The break statement breaks out of the smallest enclosing 'for' or 'while' loop.
- The else statement if present for the statement will not be executed.

```
for n in range(2, 20):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n // x)
            break
    else:
        print(n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
17 is a prime number
18 equals 2 * 9
19 is a prime number
```

The continue statement

- The continue stops the execution of the current loop iteration but continuous with the next.

```
for num in range(2, 10):
    if num % 2 == 0:
        print('Found an even number', num)
        continue
    print('Found an odd number', num)
```

```
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

The pass statement

- The pass statement does nothing.
- It can be used when a statement is required syntactically but the program requires no action.

```
while True:
    pass
```

```
class MyClass:
    pass
```

- The while loop needs a statement
  - pass is a placeholder statement
- the class definition also needs a statement. pass is the placeholder
- pass can also be handy in debugging. Put a breakpoint on the pass statement.

Exercise

- Create a program which checks if a certain integer value is present in a list with integer values.
- Write a program which will find all numbers which are divisible by 9 but are not a multiple of 5, between 4000 and 6000 (both included). The numbers obtained should be printed in a comma-separated sequence on a single line.
- With a given integral number write a program that creates and prints a dictionary consisting of the key-value pairs (i, i*i). i running from zero to n, n included.

Python functions

- A function groups a set of statements so that they can be run more than once in a program.
- Functions are also created to encapsulate an algorithm.
- The definition of a function is

```
def name(arg1, arg2, …, argN):
    statements
```

- A function body can contain a return statement

```
def name(arg1, arg2, …, argN):
    …
    return value
```

Function examples

```
def print_something():
    print('Something is printed')


def get_squares(max_value):
    return [n ** 2 for n in range(max_value)]
```

- The definition of a function must be executed before it can be used.

```
print_something()
print(get_squares(10))
```

```
Something is printed
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Default parameters

- It is possible to give parameters in a function a default value.
- When the function is called but the parameter is not given a value the default value is used.

Default parameters examples

```
def get_squares(max_value=20):
    return [n ** 2 for n in range(max_value)]


def get_squares_between(min_value,
max_value=20):
    return [n ** 2 for n in range(min_value,
max_value)]
```

```
print(get_squares())
print(get_squares(15))
print(get_squares_between(10))
print(get_squares_between(15, 25))
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256,
289, 324, 361]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
[100, 121, 144, 169, 196, 225, 256, 289, 324, 361]
[225, 256, 289, 324, 361, 400, 441, 484, 529, 576]
```

Using parameter names

- It is also possible to call a function using the name of the parameters.
- This allows the parameters to be used out of order.
- It makes the source code more readable.
- It is possible to mix positional and keyword parameters.
- The positional parameters must come first.

```
def addition(first, second):
    print(first + second)

addition(second=40, first=30)
addition(30, second=40)
```

Variable length parameters

- It is possible to give a function parameters which accepts more than one object.
- There are two flavours.
  - variable length parameter without keywords.
  - variable length parameter with keywords.
- The parameter without keywords starts with a *
  - Internally in the function this becomes a tuple.
- The parameter with keywords starts with **
  - Internally in the function this becomes a dictionary.

Example

```
def var_par_function(first, *second, **third):
    print(first)
    print(second)
    print(third)


var_par_function(10)
var_par_function(10, 100, 200)
var_par_function(10, 100, 200, var1=20, var2=30)
```

```
10
()
{}

10
(100, 200)
{}

10
(100, 200)
{'var1': 20,
'var2': 30}
```

General function definition

- The most general function definition is

```
def myfunc(*args,**kwargs):
    # do something with args
    # do something with kwargs
```

- This function accepts as many parameters as the user wants to insert.
- In the body of the function we have the variable args with type tuple and kwargs with type dictionary.

Argument unpacking

- It is also possible to unpack a tuple, a list or a generator expression with a star prefix.

```
def print_elements(x, y, z):
    print(x, y, z)
```

```
print_elements(40, 50, 60)
```

```
40 50 60
```

```
t = (100, 200, 300)
```

```
print_elements(t)
```

```
File "/demos/argument_unpacking.py", line 9, in <module>
    print_elements(t)
TypeError: print_elements() missing 2 required positional arguments: 'y' and 'z'
```

```
print_elements(*t)
```

```
100 200 300
```

Argument unpacking

```
s = ['a', 'b', 'c']

print_elements(*s)                  a b c

e = (x ** 2 for x in range(3))      <generator object <genexpr> at 0x104577e08>

print(e)

print_elements(*e)                  0 1 4
```

Argument unpacking

- It is also possible to unpack a dictionary using a two star prefix.
- To use the dictionary in the body of the functions the names of the keys in the dictionary must be equal to the function parameter names.

```
d = {'x': 100, 'y': 200, 'z': 300}
print_elements(**d)
```

```
100 200 300
```

Global variables

- In Python it is possible to use global variables.
- Global variables are variables declared outside any function.
- A global variable can be used in a function with the keyword global.

```
def function_uses_global():
    global s
    print(s)


s = 'This is a global variable'
function_uses_global()
```

```
This is a global variable
```

Global scope and function scope

- Python does know global scope and function scope.
- But not block scope.

```
global_var = 'global var'

if True:
    block_var = 'block var'


def my_func():
    local_var = 'function var'


print(global_var)
print(block_var)
print(local_var)
```

```
print(local_var)
NameError: name 'local_var' is not defined
```

Exercise

- Create a function to calculate Fibonacci numbers.
- The parameter of the function is the n-th Fibonacci number.
- The Fibonacci numbers are 0,1,1,2,3,5,8,13,21,…
- We start numbering with the 0^{th} Fibonacci number.
- Can you calculate the Fibonacci number recursively?

Lambda expressions

- Small anonymous functions can be created with the lambda keyword.
- A lambda expression is a one-line expression.
- The syntax is

  lambda [parameter_list]:expression
- Lambda expression can replace a small function which is only called once.
  - Improves readability

Lambda expressions examples

```
import math

square_root = lambda x: math.sqrt(x)
print(square_root(16))

pairs = [(2, 'two'), (4, 'four'), (1, 'one'), (3,
'three')]
pairs.sort()
print(pairs)
pairs.sort(key=lambda pair: pair[1])
print(pairs)
```

```
4.0
[(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

Exercise

- Create a list of 100 random integers between 0 and 100.000. (Use the module random).
- Now create a list which only contains the even numbers from this list.

Python built-in functions

- The Python interpreter has a number of functions built into it that are always available.
- There is a table available in the Python documentation which shows them all.
- We will discuss some interesting built-in functions

- input
- all
- any
- map
- filter
- zip

The input function

- input([prompt])
- If the prompt argument is present, it is written to standard output without a trailing newline.
- The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that.
- When EOF is read an EOFError is raised.

```
name = input('Give me your name :')
age = int(input('Give me your age :'))
print(name)
print(age)
```

```
Give me your name :Leon
Give me your age :31
Leon
31
```

The all and any functions

- all(iterable)
- The all functions returns True if all elements of the iterable are true or if the iterable is empty.
- any(iterable)
- The any function returns True if any element of the iterable is true and the iterable is not empty.

```
l = [1, 4, 0, 6, 2]
print(all(l))                    False
print(any(l))                    True
t = (None, 'Hello', 1)           False
print(all(t))                    True
print(any(t))
```

The map function

- map(function,iterable)
- Applies a function to every item of an iterable and returns an iterable of the results.

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

def sqr(x):
    return x ** 2

print(list(map(sqr, items)))
```

[1, 4, 9, 16]

```
print(list(map(lambda x: x ** 2, items)))
```

[1, 4, 9, 16]

```
print([n ** 2 for n in items])
```

[1, 4, 9, 16]

The filter function

- filter(function,iterable)
- Constructs an iterable from those elements of the iterable for which the function returns true.

```
squares = map(lambda x: x ** 2, range(10))
special_squares = filter(lambda x: x > 5 and x < 50, squares)
print(list(special_squares))
```

```
[9, 16, 25, 36, 49]
```

The zip function

- zip([s1 [, s2 [,..]]])
- returns an iterator that produces a sequence of tuples where the nth tuple is (s1[n], s2[n], ...).
- The function stops when the shortest input iterable is exhausted.

```
alist = ['a1', 'a2', 'a3']
blist = ['b1', 'b2', 'b3']

for a, b in zip(alist, blist):
    print(a, b)
```

```
a1 b1
a2 b2
a3 b3
```

Creating a dictionary with the zip function

```
list1 = ['a', 'b', 'c', 'd', 'e']
list2 = [1, 2, 3, 4]

dict1 = dict(zip(list1, list2))
print(dict1)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

Exercise

- We have the following list of lists

[[34587, 'Learning Python', 'Mark Lutz', 4, 40.95],
[98762, 'Programming Python', 'Mark Lutz', 5, 56.80],
[77225, 'Head First Python', 'Paul Berry', 3, 24.99],
[88112, 'Introduction in Python3', 'Bernd Klein', 3, 32.96]]

- Write a program which returns a list of tuples, with two elements (pairs).
- Each tuple consists of the order number and the product of the price per item and the quantity.
- This product is increased by Euro 10 if the value of the order is less than 100 Euro.

Exceptions

- Exceptions are events that modify the normal flow of control.
- Exceptions occur during the execution of a program.
- They are triggered on errors.
- This is called the exception is raised.
- A raised exception must be excepted otherwise the program stops and shows a traceback.
- An exception is an object.
- An assertion is a special kind of exception. It is put in by the programmer to find out if the value of a variable is as expected.

Exception handling

- Place the source code that might raise an exception in a try block
- Add an except statement to catch the exception
- In the except statement try to solve the issue raised

```
def division(x, y):
    try:
        return x / y
    except ZeroDivisionError:
        print('You did divide by zero,stupid')
```

```
print(division(5, 2))
print(division(5, 0))
```

```
2.5
You did divide by zero, stupid
None
```

Using the exception object

- With the 'as var' modifier in the except statement it is possible to give the exception object a name and use the name to get more info.

```
def division(x, y):
    try:
        return x / y
    except ZeroDivisionError as e:
        print(e)


print(division(5, 2))
print(division(5, 0))
```

```
2.5
division by zero
None
```

Generic except statement

- We can use the generic except statement which catches all errors.
- But this is bad practice. Flagged by PEP8 as 'do not use bare except'

```
def open_file():
    try:
        f = open('aFile')
    except:
        print("Can't open file")
```

```
open_file()
```

```
Can't open file
```

Multiple exceptions handling block

- It is possible to use more then one except block in order to catch different exceptions.

```
def generate_exceptions(z):
    try:
        x = 10
        c = x / z  # Can generate a ZeroDivisionError

        d = (1, 2, 3)
        d[0] = 4  # Is a TypeError
    except ZeroDivisionError as e:
        print(e)
    except TypeError as e:
        print(e)
```

```
generate_exceptions(0)
generate_exceptions(10)
```

```
division by zero
'tuple' object does not support item assignment
```

Else clause

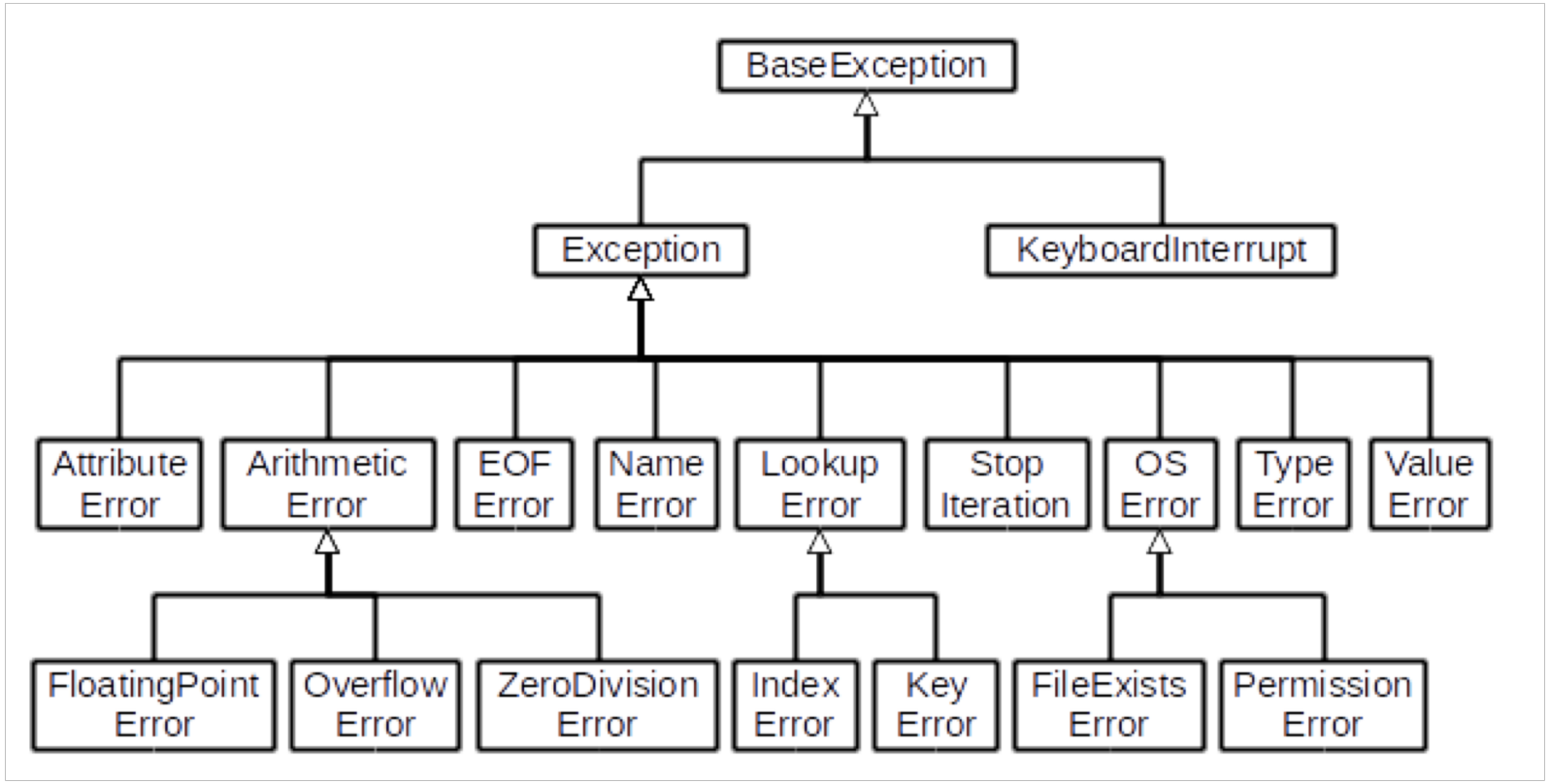
- It is possible to have an else clause
- This clause is executed when no exception is raised

```
def get_input():
    try:
        x = int(input('Enter a
number\n'))
    except ValueError:
        print('Not a number')
    else:
        print('Input was a number')


get_input()
```

```
Enter a number
10
Input was a number


Enter a number
a
Not a number
```

Python standard exceptions

- A number of standard exceptions are defined in Python.
- They are arranged in a hierarchy.



Examples of standard exceptions

- OverflowError
  - Raised when a calculation exceeds maximum limit for a numeric type.
- ZeroDivisonError
  - Raised when division or modulo by zero takes place for all numeric types.
- IOError
  - Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
- SyntaxError
  - Raised when there is an error in Python syntax.

Examples of standard exceptions

- ModuleNotFoundError
  - raised by import when a module could not be located
- TypeError
  - Raised when an operation or function is attempted that is invalid for the specified data type.
- ValueError
  - Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
- RuntimeError
  - Raised when a generated error does not fall into any category.

Try finally clause

- Also a finally clause can be added to a try block
- The finally block is always executed whether an exception is raised or not

```
try:
   do_some_stuff()
except:
   print('An exception')
finally:
   cleanup_stuff()
```

- Example open(file)
  - When a file is opened it also must be closed disregarding an exception is raised or not.

Raise an exception

- An exception can be raised.
- For instance when something in a function goes wrong.
- The function does not deal with this but raises an exception.
  - e.g. a connection with a server can not be made
- In the body of the function there is the statement

  raise SomeException
- The source code that calls the function uses the try .. except clauses to catch the error.

```
def raise_exception():
    raise Exception('My exception')


raise_exception()
```

Creating your own exception

- One way of creating new exceptions is by using the existing ones with a specific message
  - NetworkError('No network')
- A more thorough approach is sub classing an existing exception class

```
class MyException(Exception):

    def __init__(self, errno, message):
        self.args = (errno, message)
        self.errno = errno
        self.errmsg = message


raise MyException(1, 'Severe problem')
```

Exercise

- Write a function to ask a user for a maximum of three time to input a password of at least 8 characters of which one is a digit and one is a capital letter.
- After the three time give the user a message that login has failed
- Make a version of the function without exceptions
- Make a version which handles the exceptions in the function
- Make a version which raises an exception and handle the exception in the source code outside the function

Modules and Packages

- Python programs are organized in modules and packages.
- The Python standard library consists of a large number of modules.
- Any Python source file can be used as a module.
- A module is loaded in a file with the "import" statement as we have seen earlier.
- The import statement executes all of the statements in the loaded module.
- To access the functions, variables and classes of a module the name of the module must be used as a prefix.
- A module is only imported once.

Module search path

- When importing a module (or package) the paths to be searched for are defined in the sys module and can be inspected by the function sys.path.
- These paths are
  - the directory in which the file is run.
  - the path defined in the environment variable PYTHONPATH.
  - a number of subdirectories in the Python installation directory.

```
import sys

print(sys.path)
```

```
['demos', '/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6']
```

The __name__ property

- With the property __name__ the name of a module can be obtained.
- Using the __name__ property for the file which is run gives the name __main__
- This makes it possible to change the behaviour of a file when it is run or when it is imported.

```
my_variable = 10

if __name__ == '__main__':
    print('This is the main file')
    print(my_variable)
```

The dir function

- All the names defined in a module can be fetched with the dir function.

```
import math

print(dir(math))
```

```
['__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh',
'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp',
'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

Packages

- Packages make it possible to group modules under a common package name.
- A package is nothing else than a directory structure with modules.
- Before Python 3.3 it was mandatory to put an __init__.py module in each directory of the package.
- Since 3.3 this is optional but the __init__.py module is useful to initialize the use of a module because it is read by the run time at the moment the directory of the package is used in an import.

```
graphics/
    __init__.py
    primitive/
        __init__.py
        lines.py
        fill.py
        text.py
    formats/
        __init__.py
        gif.py
        png.py
```

Explicit import of modules in a package

- Modules can be imported explicitly.
  - import graphics.primitive.lines
- A function must then be accessed using the full name.
  - graphics.primitive.lines.lines_print()

lines.py

```
def lines_print():
    print('this is in lines_print')


print('This is the module lines.py')
```

```
import graphics.primitive.lines

graphics.primitive.lines.lines_print()
```

Explicit import of modules in a package

- Omitting the prefix can be done with the import
  - from Graphics.Primitive import lines

```
from graphics.primitive.lines import lines_print

lines_print()
```

- Also possible
  - from Graphics.Primitive.lines import linesPrint
  - or
  - from Graphics.Primitive.lines import *

```
from graphics.primitive.lines import *

lines_print()
```

__init__.py

- When a package or a subpackage is imported the files __init__.py are executed.
- This file may be empty but can also be used to restrict access.
- For example the file graphics/primitive/__init__.py can contain a variable __all__ which contains a list of all available modules when importing the package.

```
__all__ = ['lines', 'text', 'fill']
```

- Or to make accessing certain function easier by having import in the file.

```
from graphics.primitive.lines import lines_print
```

- lines_print is now available as if the function was defined in graphics.

Object Oriented Programming

- Large software systems are very complex.
- In general they are difficult to maintain and to extend.
- To make the development and maintenance easier software must be split up in smaller parts.
  - This can be done by using functions and structured data
  - Another approach is using classes and objects
- A class gives definitions for functions and data belonging to the class.
- Both functions and data are called attributes.
- For a function in the class also the name method is used.

Classes and objects

- From classes objects are made by initialization of the class.
- Classes make re-use possible by making more objects from the same class.
- It is also possible to create a sub-class of a class.
- A sub-class is more specialized as the super-class.
  - In general it has more data and more methods
- This is called inheritance and here also code is re-used.
- In general the data in a class is not direct accessible from the outside. Getter and setter methods are needed to get to the data.
- This is called encapsulation

Classes

- The general format of a class definition is as follows

```
class ClassName:
    statement-1
    …
    statement-n
```

- The definition of a class must be executed before it can be used.
- When a class definition is entered a new namespace is created.
  - variables and functions defined with the definition are bound to this namespace.
- When a class definition is left a class object is created.
  - This is a wrapper around the content of the namespace.

Classes

- A class is a user defined type.
- A class is instantiated to build an instance i.e. an object of the type.
- Instantiation is done by calling the class name.
- The __init__ function within the class definition initialises the created object
- The __init__ function has as first parameter the self parameter which references the just created object.
- If no __init__ function is present an object can still be created by calling the class but no initialisation takes place.

An example class

```
class Person:

    def __init__(self, firstname='', lastname='', age=-1):
        self.firstname = firstname
        self.lastname = lastname
        self.age = age
```

- self.firstname, self.lastname and self.age are instance variables belonging to an object created from this class.
- They are created the moment the __init__ method is executed.

Object creation

```
p1 = Person('John','Michels',34)

print('The first name of p1 is :'+p1.firstname +
      '\nthe last name is :' + p1.lastname +
      '\nthe age is :' + str(p1.age))
```

```
The first name of p1 is :John
the last name is :Michels
the age is :34
```

Object creation

```
p2 = Person()
print('The first name of p2 is :'+p2.firstname +
    '\nthe last name is :' + p2.lastname +
    '\nthe age is :' + str(p2.age))

p3 = Person()
print(p3)
```

```
The first name of p2 is :
the last name is :
the age is :-1

<__main__.Person object at 0x10469db38>
```

The docstring

- All modules, functions and classes should have a docstring that describes the module, class or function.
- A docstring is a triple-double-quoted string immediately at the beginning of a module, after the def statement or the class statement.
- This is not required by the language but it gives information about the function or class and can be read by people using the code.
- In a well documented program docstrings are quite large.
- For a function the dosctring describes the purpose of the function, and if needed, the parameters and what the function returns.
- PEP 257 describes docstring conventions.

docstring example

```
def add(x, y):
    """Returns the sum of x and y"""
    return x + y


class SavingsAccount:
    """The class SavingsAccount mirrors the
savings account
        of the bank.
        * there are two fields
            -balance
            -overdue_max
        * there are two methods
            extract_money
            add_money """
```

Showing the docstring

- the docstring info becomes visible when we call help on the function or class.
- It can also be made visible using the __doc__ attribute

```
help(add)
```

```
add(x, y)
    Returns the sum of x and y
```

```
print(add.__doc__)
```

```
Returns the sum of x and y
```

The __str__ method

- In general the information you want to show using print contains the same elements for all objects instantiated from the same class.
- To make this possible in a structured way the Python system method __str__ is available.
- When the print function is called on an object the Python runtime uses the __str__ method when available.

Example __str__

```
class Person:

    def __init__(self, firstname='', lastname='', age=-1):
        self.firstname = firstname
        self.lastname = lastname
        self.age = age

    def __str__(self):
        return self.firstname + ' ' + self.lastname + ' ' + str(self.age)


p1 = Person('John','Michels',34)
print(p1)
```

John Michels 34

Exercise

- Create a class Building with a name, city, street, number and the year it was build.
- Make objects of this class.
- Print some info about an object.
- Add the __str__ method to the class.
- Print some info again.

Instance objects

- Instance objects have attributes
- An attribute can be a
  - data attribute
  - method attribute
- Data attributes need not be declared.
  - When first assigned they exist.

```
class Person:

  def __init__(self, firstname='', lastname='', age=-1):
    self.firstname = firstname
    self.lastname = lastname
    self.age = age
```

methods

- A method is a function that belongs to an object.
- The first parameter of a method is self.
- This parameter refers to the object the method belongs to.

```
class Myclass:

    def myfunc(self):
        print('This is a method')


c = Myclass()
c.myfunc()
```

```
This is a method
```

Method example

```
class Person:
    def __init__(self, firstname='', lastname='', age=-1):
        self.firstname = firstname
        self.lastname = lastname
        self.age = age

    def __str__(self):
        return self.firstname + ' ' + self.lastname + ' ' +
str(self.age)

    def getfullname(self):
        return self.firstname + ' ' + self.lastname


p1 = Person('John', 'Michels', 34)

print(p1.getfullname())
```

John Michels

Private attributes

- The syntax of Python doesn't know the concept of private attributes.
- The convention is that prefixing an attribute with _ makes it private.
- When prefixing an attribute with __ (double underscore) the Python Interpreter will "mangle" the name.

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self._age = age


p = Person('Hans', 32)
print(dir(p))
```

```
['_Person__name', '__class__', '__delattr__', '__dict__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', '_age']
```

Getter and setters

- It is recommended to make instance variables private and used getters and setters to read/change the private variables.

Getters and setters

```
class Person:
    """
    This is a person
    """

    def __init__(self, firstname='Unknown', lastname='Unknown', age=-1):
        self.__firstname = firstname
        self.__lastname = lastname
        self.__age = age

    def setfirstname(self, firstname):
        self.__firstname = firstname

    def getfirstname(self):
        return self.__firstname

    def setlastname(self, lastname):
        self.__lastname = lastname

    def getlastname(self):
        return self.__lastname

    def setage(self, age):
        self.__age = age

    def getage(self):
        return self.__age
```

Exercise

- Change the building class such that all instance variables become private and create properties to access the data.

Inheritance

- With inheritance a new class is created that specializes or modifies the behaviour of an existing class.
- The original class is called base class or superclass.
- The new class is called derived class or subclass.
- Inheritance is specified by a comma separated list of class names in the definition of the derived class.
- The subclass inherits the attributes defined by its base classes.
- A derived class may redefine any of these attributes and add new attributes of its own.

Inheritance

- All classes inherit from object
- object is at the root of the Python classes.
- object has a default implementation for a number of methods like __str__

```
class Person:

    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name


class Employee(Person):

    def __init__(self, name, salary):
        super().__init__(name)
        self.salary = salary
```

The method super()

```
class Person:

    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name


class Employee(Person):

    def __init__(self, name, salary):
        super().__init__(name)
        self.salary = salary

    def __str__(self):
        return super().__str__() + ',' + str(self.salary)
```

```
e = Employee('Peter', 1000)
print(e)
```

```
Peter,1000
```

- With the method super() the subclass can get to it's direct superclass.

Exercise

- Create an Employee class as subclass of Person that has a private firstName, lastName, salary.
- Create an instance method payRaise with as parameter the percent pay raise. The pay raise is added to the current salary.
- Make sure an object of the class can be printed with relevant information.
- Make some employees and show some information

Exercise

- Create a Manager as a sub class of Employee with as an extra private variable a bonus.
- Create properties for bonus.
- Add a method payBonus which adds the bonus to the salary.
- Create Manager objects.
- Make sure you can show relevant information.

Class and instance variables

- The word instance variable is used for data unique to each instance.
- The word class variable is used for attributes shared by all instances of the class

```
class Employee:
   nr_of_employees = 0;  # class variable

   def __init__(self, name):
      self.name = name  # instance variable
      Employee.nr_of_employees += 1  # accessing a class variable


p = Employee('Andrew')

print(Employee.nr_of_employees)
print(p.nr_of_employees)
```

```
1
1
```

Class method

- Class methods are methods that operate on the class itself as an object.
- A class method must be prefixed with the @classmethod decorator.
- The first parameter is now the class object and by default it's name is cls.

A class method

```
class Employee:
    _max_salary = 5000;  # class variable

    def __init__(self, name):
        self.name = name  # instance variable

    @classmethod
    def get_max_salary(cls):
        return cls._max_salary


print(Employee.get_max_salary())
```

5000

The file type

- The built-in open function creates a Python file object, which serves as a link to the file.
- When opened data can be read or written to the file.
- Python differentiates between text files and data files.
  - Text file are unicode characters and consists of lines of text.
  - Data files consists of bytes of data and doesn't have lines.

Files

```
f = open('data.txt','w')
f.write('Hello')
f.write(' world\n')
f.close()
f = open('data.txt')
text = f.read()
print(text)
f.close()
```

```
Hello world
```

Writing lines

```
f = open('data.txt','w')
count = f.write('Hello\n')
print(count)
f.write('World\n')
f.write('Next line\n')
f.close()
```

```
6
```

Reading lines

```
f = open('data.txt')
for line in f:
    print(line)
```

```
Hello

World

Next line
```

The seek function

- The function seek can be used to change the position in the file.

```
f.seek(0)
for line in f:
    print(line,end='')
f.close()
```

```
Hello
World
Next line
```

File modes

- A file can be opened in different modes.
- r - opens a text file for reading only.
- rb - opens a binary file for reading.
- r+ - opens a text file for reading and writing.
- w - opens a text file for writing
- wb - opens a file for writing in binary format.
- a - opens a text file for appending

Exercise

- Create a program that reads a text file and produces a new text file with the same lines as in the original file but prefixed with a line number.

Reading a file binary

- Reading a file with in the binary mode will return byte values.
- The file can be read in chunks with read(N). N is the size of the buffer.
- Line feeds are not recognized as such.

The with statement

- In order to be sure that a file is closed when opening it.
- Also when an exception occurs
- Use can be made of the with statement.
- It is not necessary to close the file.
- This will be done by the Python runtime.

```
with open('output.txt', 'w') as f:
    f.write('Hi there!')
```

Exercise

- Create a text file, using Python, with the following two lines using the with statement.

  This is a text file.

  This file is read as a binary file.

- Read the file in binary mode and print the bytes received.
- Use the with statement.

Python network modules

- The standard library of Python has a number of network modules.
- For HTTP/HTTPS requests
  - urllib , urllib2, urllib3
  - requests.
- For TCP/UDP traffic
  - socket

The requests module

- The request module is seen as one of the best modules for HTTP/HTTPS traffic. Because of it's simple to use API and rich functionality.
  - Automatic parameter encoding.
  - Automatic content decoding.
  - Use of key/value cookies.
  - Unicode bodies.
  - Management of connection timeouts.
  - And more.

A simple Http request

- requests is not a module in the standard library.
- It must be installed with pip install requests
- Then the module requests must be imported.
- Make a get or post request.
- The object returned is a Response object.
- It has several methods to know the result of the request and to get at the data if present.

```
import requests

r =
requests.get('https://api.github.com/events')
if r.status_code == 200:
    print(r.text)
```

Http requests with parameters

- Http parameters are just a dictionary parameter in the request.

```
r = requests.get('https://api.github.com/user',
auth=('your_user_name', 'your_password'))
if r.status_code == 200:
    print(r.text)
else:
    print("request failed")

r = requests.post('http://httpbin.org/post', data={'key': 'value'})
```

Status codes

- Requests has a built-in status code lookup object for easy reference.

```
r = requests.get('https://api.github.com/events')
if r.status_code == requests.codes.ok:
    print(r.text)
```

- Some other values are

    requests.codes.bad_request

    requests.codes.forbidden

    requests.codes.internal_server_error

Response content

- The content-encoding which is a header in the http request and response is used to compress the media-type.
- When present, its value indicates which encodings were applied to the entity-body.
- It lets the client know how to decode.
- When using requests the content encoding can be read by

  r.enconding
- When the coding is json there is a built-in json decoder which raises an exception when the decoding fails.

  r.json()

Request timeout

- With a timeout a request will wait for a number of seconds on a reaction of the server.
- In production code a timeout is almost always a paremeter in the request.
- If not there is possibility of a hanging program.

```
try:
    requests.get('http://github.com', timeout=0.001)
except requests.exceptions.Timeout as e:
    print(e)
```

```
HTTPConnectionPool(host='github.com', port=80): Read timed out.
(read timeout=0.001)
```

json module

- Json to Python
- Data can be converted to from a Json string to Python objects (lists and dictionaries) using the function loads.
- The load function converts an Json file into a Python objects.
  - The file must be opened first.

```
import json

jsonData = '{"name": "Frank", "age": 39}'
jsonToPython = json.loads(jsonData)
print(jsonToPython)
```

```
{'name': 'Frank', 'age': 39}
```

json module

- Python to Json
- The dumps statement creates a Json data string from Python lists and dictionaries.
- The dump statement converts the objects into Json data in a file.
  - The file must be open for writing

```
import json

pythonDictionary = {'name': 'Bob', 'age': 44, 'isEmployed': True}
dictionaryToJson = json.dumps(pythonDictionary)
print(dictionaryToJson)
```

```
{"name": "Bob", "age": 44, "isEmployed": true}
```

Exercise

- Fetch the Json formatted file persons.txt from the server.
- Convert the file to Python objects.
- Create a list of Persons and print the list.

Python Database Access

- Python has a standard interface for database access, Python Database API Specification v2.0, described in PEP 249.
- Most Python database interfaces adhere to this standard.
- For each database a special module is used to implement the interface.
- Database which can be used are
  - MySQL
  - Oracle
  - Informix
  - Sybase
  - SQLite3

SQLite3 database

- We will use the SQLite3 module because this module is capable of creating and managing an sqlite3 database.
- Database is just a local file
  - Not a client server based solution as SQL
  - Not all types possible
  - More tables can be present in a database
  - Python has a SQLite module to deal with SQLite databases

Actions to create a SQLite table

- First the sqlite3 module must be imported.
- The database must be created/opened with the function.
  - conn = sqlite3.connect('database_name')
- Queries on the database use a cursor.
  - cursor = conn.cursor()
- A table must be created and filled with data.

Creating and filling a table

```
import sqlite3

conn = sqlite3.connect('employees.db')

with conn:
    cursor = conn.cursor()
    cursor.execute("Create TABLE Employee(Id INTEGER, FirstName TEXT,LastName TEXT, Salary INT)")
    cursor.execute("INSERT INTO Employee VALUES(1,'Hans','Zwart',2000)")
    cursor.execute("INSERT INTO Employee VALUES(2,'Mary','De Goede',2200)")
    cursor.execute("INSERT INTO Employee VALUES(3,'Joke','Bartels',2800)")
    cursor.execute("INSERT INTO Employee VALUES(4,'John','de Wit',3000)")
```

drop a table

```
import sqlite3

conn = sqlite3.connect('employees.db')

with conn:
    cursor = conn.cursor()
    cursor.execute("DROP TABLE Employee")
```

- Or

```
cursor.execute("DROP TABLE IF EXISTS Employee")
```

Creating a table using a primary key

```
import sqlite3

conn = sqlite3.connect('employees.db')

with conn:
    cursor = conn.cursor()
    cursor.execute("Create TABLE Employee(Id INTEGER PRIMARY KEY, FirstName TEXT,LastName TEXT, Salary INT)")
    cursor.execute("INSERT INTO Employee(FirstName,LastName,Salary) VALUES('Hans','Zwart',2000)")
    cursor.execute("INSERT INTO Employee(FirstName,LastName,Salary) VALUES('Mary','De Goede',2200)")
    cursor.execute("INSERT INTO Employee(FirstName,LastName,Salary) VALUES('Joke','Bartels',2800)")
    cursor.execute("INSERT INTO Employee(FirstName,LastName,Salary) VALUES('John','de Wit',3000)")
    cursor.execute("SELECT * FROM Employee")
    print(list(cursor))
```

- The primary key is auto incremented.

Prepared statements

- A query can first be prepared and then executed.
- This is called a prepared statement.
- Advantages of a prepared statements
  - Can be executed many times
  - Protects against SQL injection
- Uses question marks as placeholders.

```
import sqlite3

conn = sqlite3.connect('employees.db')

with conn:
    cursor = conn.cursor()
    cursor.execute("INSERT INTO Employee(FirstName,LastName,Salary) VALUES(?,?,?)", ('Anita','Bruin',4000))
```

Prepared statement executemany

- using the function executemany a list of employees can be inserted into the table.

```
import sqlite3

employees = [('Peter', 'Haantjes', 5000),
             ('Sanne', 'Jongejans', 6000),
             ('Ruud', 'van der mark', 2100)]

conn = sqlite3.connect('employees.db')

with conn:
    cursor = conn.cursor()
    cursor.executemany("INSERT INTO Employee(FirstName,LastName,Salary) VALUES(?,?,?)", employees)
```

Retrieving all data

```
import sqlite3

conn = sqlite3.connect('employees.db')

with conn:
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM Employee")
    print(list(cursor))
```

- selects all data from the table Employee.

Retrieving some data

```
import sqlite3

conn = sqlite3.connect('employees.db')

with conn:
    cursor = conn.cursor()
    cursor.execute("SELECT FirstName FROM Employee")
    print(list(cursor))
```

```
[('Hans',), ('Mary',), ('Joke',), ('John',), ('Anita',), ('Peter',), ('Sanne',), ('Ruud',)]
```

Retrieving data with a where condition

```
import sqlite3

conn = sqlite3.connect('employees.db')

with conn:
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM Employee WHERE Salary > 4000")
    while True:
        row = cursor.fetchone()
        if row is None:
            break
        print(row[1], row[2])
```

```
Peter Haantjes
Sanne Jongejans
Peter Haantjes
```

Fetch functions

- There are a number of fetch functions
- Fetches one row. The cursor is advanced one row.

```
cursor.fetchone()
```

- Fetches all results from the position of the cursor.

```
cursor.fetchall()
```

- In each case the cursor advances the number of rows fetched.

```
cursor.fetchmany(2)
```

Commit and rollback

- To write all changes definitely to the database call commit() on the connection.
- When changes are not committed they can be undone by calling rollback on the connection.

Rollback

```
import sqlite3

conn = sqlite3.connect('employees.db')

with conn:
    cursor = conn.cursor()
    cursor.execute("Create TABLE Employee(Id INTEGER PRIMARY KEY, FirstName TEXT,LastName TEXT, Salary INT)")
    cursor.execute("INSERT INTO Employee(FirstName,LastName,Salary) VALUES('Hans','Zwart',2000)")
    cursor.execute("INSERT INTO Employee(FirstName,LastName,Salary) VALUES('Mary','De Goede',2200)")
    cursor.execute("INSERT INTO Employee(FirstName,LastName,Salary) VALUES('Joke','Bartels',2800)")
    cursor.execute("INSERT INTO Employee(FirstName,LastName,Salary) VALUES('John','de Wit',3000)")
    conn.rollback()
    cursor.execute("SELECT * FROM Employee")
    print(list(cursor))
```

Commit

```
import sqlite3

conn = sqlite3.connect('employees.db')

with conn:
    cursor = conn.cursor()
    cursor.execute("Create TABLE Employee(Id INTEGER PRIMARY KEY, FirstName TEXT,LastName TEXT, Salary INT)")
    cursor.execute("INSERT INTO Employee(FirstName,LastName,Salary) VALUES('Hans','Zwart',2000)")
    cursor.execute("INSERT INTO Employee(FirstName,LastName,Salary) VALUES('Mary','De Goede',2200)")
    cursor.execute("INSERT INTO Employee(FirstName,LastName,Salary) VALUES('Joke','Bartels',2800)")
    cursor.execute("INSERT INTO Employee(FirstName,LastName,Salary) VALUES('John','de Wit',3000)")
    conn.commit()
    conn.rollback()
    cursor.execute("SELECT * FROM Employee")
    print(list(cursor))
```

Update

```
import sqlite3

conn = sqlite3.connect('employees.db')

with conn:
    cursor = conn.cursor()
    cursor.execute("UPDATE Employee SET Salary=2500 WHERE Id=1")
    print('Total number of rows updated :', conn.total_changes)
    result = cursor.execute("SELECT * FROM Employee")
    result = cursor.fetchone()
    print(result)
```

Delete

```
import sqlite3

conn = sqlite3.connect('employees.db')

with conn:
    cursor = conn.cursor()
    cursor.execute("DELETE FROM Employee WHERE Id=1")
    print('Total number of rows updated :', conn.total_changes)
    result = cursor.execute("SELECT * FROM Employee")
    result = cursor.fetchall()
    print(result)
```

Exercise

- Create the employees database with the Employee table.
- Fetch the data from the file employees.json from the server.
- Populate the table with the data.
- Show with that the data are in the table.