Python Advanced

Warning!

- **For Python Advanced, it is mandatory to have followed the course Python Fundamentals.**

- **Python Fundamentals is an entry requirement for Python Advanced.**

Program

Day 1

File handling
Exception handling
Object oriented programming
Classes, methodes and attributes
Magic methodes
Inheritance

Day 2

Python Standard Library
Python Package Index

Python for Data Science beginnings

Recap

- Python prompt
- Executing a Python module
- Variables
- Operators
- Built-in functions
- Strings
- Conditional statements
- Loop statements
- Datastructures – Lists, Sets, Dicts
- Comprehension
- Functions – definition, arguments, return
- Generators

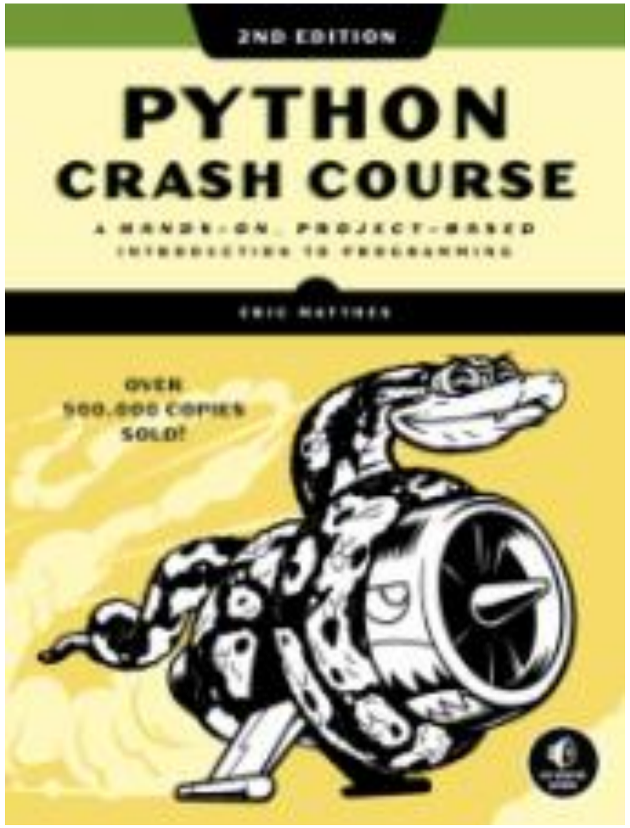
Book

Part I: Basics

1. Getting started
2. Variables and Simple Data Types
3. Introducing Lists
4. Working with Lists
5. If Statements
6. Dictionaries
7. User input and While Loops
8. Functions
9. Classes
10. Files and Exceptions
11. Testing Your Code

Part II: Projects

12. Project 1: Alien Invasion
13. Project 2: Data Visualization
14. Project 3: Web Applications


2ND EDITION
PYTHON CRASH COURSE
A HANDS-ON, PROJECT-BASED INTRODUCTION TO PROGRAMMING
ERIC MATTHES
OVER 500,000 COPIES SOLD!

Resources:
https://ehmatthes.github.io/pcc_2e/regular_index/

Read from a file

- The **open()** built-in function is used to access files
- A file can be opened in different modes: read, write or append
- The keyword **with** specifies a context manager

read	readlines	seek
readline	write	close

```
keyword = 'xxx'
filename = 'data.txt'

with open(filename) as f:
    for line in f:
        line = line.strip()
        if keyword in line:
            print(line)
```

Write to a file

- Modes: r, w, a, b

```
filename = 'data.txt'

with open(filename, 'w') as f:
   f.write('ID, A, B\n’)
   f.write('line 1, 2.0, 10.0\n’)
   f.write('line 2, 2.1, 10.0\n’)
   f.write('line 3, 2.1, 10.0\n')
```

Exercise 1.1

Writing to and reading from a file

- First create a file with open and write mode
- Write a header line to the file. E.g. 'ID,var1,var2,var3'
- Write a couple of lines to the file. E.g. '1001, 5, 1.23, "Y"'

- Then create a new program
- Open the file in read mode
- Read the header and split into a list of headers
- For each line split the line into values
- Create a dictionary with the header and the values using zip()

- Filter on one off the fields and print the lines

Exercise 1.2

Read a CSV file

Filter lines from a CSV file

Tips:

- Get the ca-500.csv file
- Open the file within a context manager with the keyword **with**
- Read the first line. The header.
- In a for loop through all lines.
- For each line strip the newline character from the end with **strip**
- Split the line into a list of values with **split**
- Only select lines with city 'Montreal'
- Print firstname, lastname, city and email

Solution Read a CSV file

```
filename = "ca-500.csv"

with open(filename) as f:

    headers = f.readline().rstrip("\n").split(';')

    for line in f:
        columns = line.rstrip("\n").split(';')

        d = dict(zip(headers, columns))

        if d['city'] in ('Montreal', 'Vancouver'):

            print("{:10} {:15} {:20} {:30}".format(
                  d['first_name'],
                  d['last_name'],
                  d['city'],
                  d['email']))
```

Reading a CSV file with csv

- **csv** module from Python Standard Library

```
import csv

filename = "ca-500.csv"
colnames = ('first_name', 'last_name', 'city', 'email')

with open(filename) as f:
    reader = csv.DictReader(f, delimiter=';', quotechar='"')

    for d in reader:
        if d['city'] in 'Montreal’:

            print('{:10}{:20}{:30}{:30}'.format(
                *[d[fieldname] for fieldname in colnames]))
```

Exceptions

- Run-time errors cause the execution of the code to stop.
- Run-time errors are called **Exceptions**

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

- Where are many different types of Exceptions:

StopIteration
SystemExit
StandardError
ArithmeticError
OverflowError
FloatingPointError
ZeroDivisionError
AssertionError
AttributeError
EOFError
ImportError
KeyboardInterrupt
LookupError
IndexError
KeyError
NameError
UnboundLocalError
EnvironmentError
IOError
OSError
SyntaxError
IndentationError
SystemError
SystemExit
TypeError
ValueError
RuntimeError
NotImplementedError

Catching Exceptions

- Exceptions can be caught by using the **try** and **except** keywords.
- Different exceptions can be caught by multiple except blocks
- A **finally** and an **else** block can optionally also be added.

```
try:
   d = 1/0
   d = int("one")

except ZeroDivisionError:
   print("Cannot divide by zero")

except ValueError:
   print("Wrong type")

except:
   print("Another type of error occured")
```

EAFP versus LBYL

EAFP

Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many try and except statements. The technique contrasts with the LBYL style common to many other languages such as C.

LBYL

Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the EAFP approach and is characterized by the presence of many if statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between "the looking" and "the leaping". For example, the code, if key in mapping: return mapping[key] can fail if another thread removes *key* from *mapping* after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

Throwing an Exception

- An exception can also be thrown from the code with the **raise** keyword.
- An built-in Exception can be thrown or a custom user-defined Exception

```
def calculate_area(width, height):
    if width < 0 or height < 0:
        raise Exception('Invalid argument')
    return width * height

try:
    area = calculate_area(10, -2)

except Exception as err:
    print(err)
```

Exercise 1.3

Custom error message

Try to open a (non-existing) file for reading and return a custom error message if the file does not exist.

Tips:

- Assign a **filename** to a variable. E.g. filename = 'a_file_that_does_not_exist.txt'
- Add a **try** statement
- Open the file within the try block using the context manager **with**
- If succesfull **read** the complete file and print the contents.
- Add an **except** statement for an IOError exception
- Within the except block **print** a custom error message "Cannot open the file"

Exercise 1.4

Foolproof numeric input

Create a function that asks to enter a number between two bounds given as arguments. The function should gracefully handle numbers outside of the bounds and also wrong types of input.

Tips:

- Define a function numeric_input with argument lower and upper
- In a while loop use input() to get a response from the user
- Turn the input into a number with int()
- Catch the error if the input cannot be converted to a number and give a message.
- Check if the number is between the given bounds. If not give a message.
- Break out of the loop if a correct number was entered.
- Return the number
- Test the function

History of way of programming

- Before 1990, every programmer was programming sequentially.
- Later, programmers began programming with functions and procedures.
- In the 1990s, a programming standard came, called:

  Object Oriendted Programming
- Nowadays, this has become the standard in all programming languages. Also in Python!

Classes vs Object

- Question to you:

- What is a class?

- What is an object?

Classes vs Objects

- Entities from the real world are supposed to be made in software.
- Goal: Make domain classes for the Python application
- With the help of a domain class, you can make domain objects.

Classes vs Objects

- A class can be compared to a plaster.
- An object be compared to a cast.

- Casts can be made with plasters.

- Example:
  A cast of a Hand is made out of
  a plaster of the Hand.
- Cast of a Hand = Object
- Plaster of the Hand = Class

- Each object is different with its own characteristics (attributes) and behaviour (mehods).

Source picture: https://i.pinimg.com/originals/be/30/c9/be30c9788460cad517028306f4259cb4.jpg

Object Oriented Programming

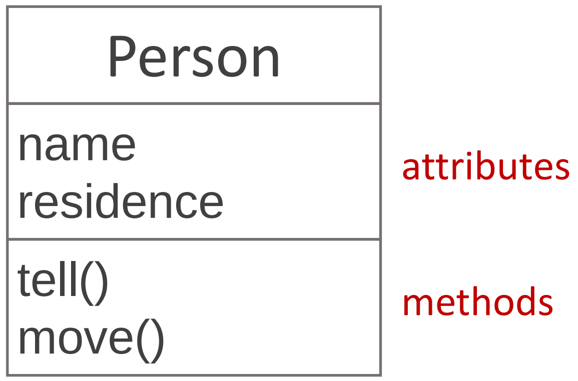
Class

Objects

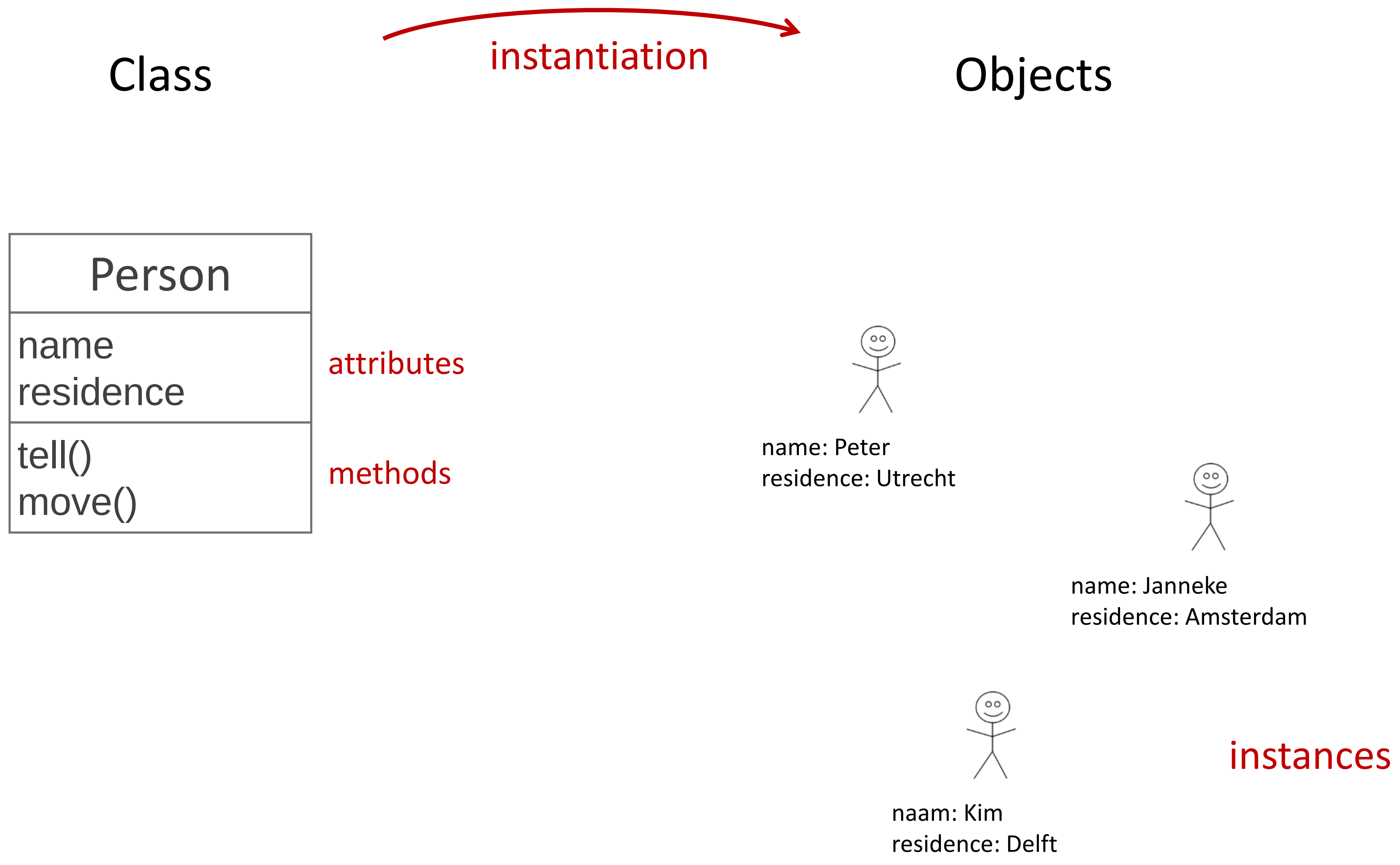
Object Oriented Programming

Class

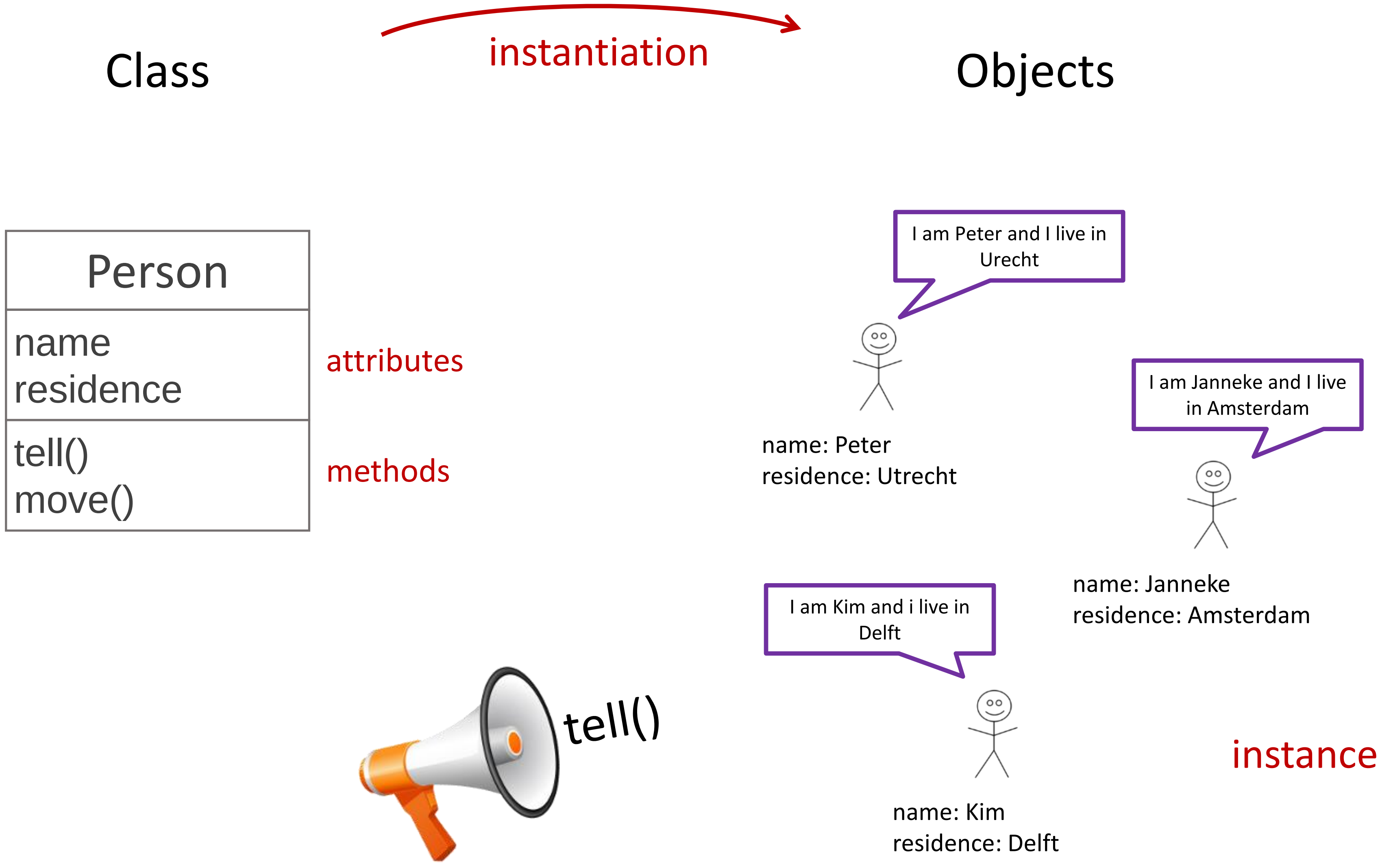
Objects



Object Oriented Programming



Object Oriented Programming



Object Oriented Programming

Class $\xrightarrow{\text{instantiation}}$ Objects

str

upper()
lower()
split()
strip()
join()
title()
capitalize()
replace()

attributes

methods

'something'

'something else'

'hello world'

instances

Object Oriented Programming

Class $\xrightarrow{\text{instantiation}}$ Objects

list

append()
insert()
extend()
pop()
sort()

attributes

methods

[1, 2, 3]

['something', 'something else']

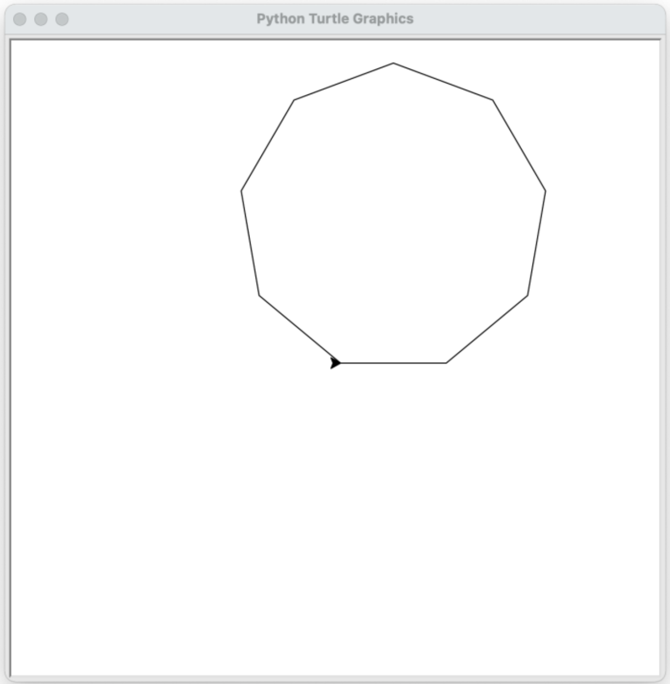
instances

Exercise 1.5 or demonstration by the trainer

Turtle

Draw a polygon with turtle.

- Check out the different methods that you can use with turtle
  - in particular: forward and left
- Import turtle
- End the program with turtle.done()
- Draw a square
- Draw a polygon
  - Calculate the angle of each corner https://nl.wikipedia.org/wiki/Regelmatige_veelhoek



Classes

- First define a class with the keyword **class**
- Instantiate an object with the class
- Set the state of the object by assigning values to the attributes
- Call the methods of the object
- Use the object operator **.** (a dot) to access attributes and methods

```
class Person:
    pass

# --------------------------------

p = Person()
p.name = 'Albert'
p.residence = 'Amsterdam'
```

Methods

- Methods are just like functions with in a class. Methodes can have arguments and a return statement just like normal functions.
- The first argument is automatically set to the 'target' object. This is typically called **self**.
- Also access methods met the object operator .

```
class Person:
    def tell(self):
        return f'I am f{self.name}'

# -----------------------------------------------------------

p = Person()
p.name = 'Albert'
print( p.tell() )
```

Object initialisation

- When an object in created (instantiated) from a class de **__init__** method is automatically called
- __init__ is called a **magic method**. Where are many magic methods.

```
class Person:
   def __init__(self, name):
      self.name = name
   def tell(self):
      return(f'I am {self.name}')

# -----------------------------------------------------------


p = Person('Albert')
print( p.tell() )
```

Public or not

- An attribute can be indicated as **non-public** by adding _ as a prefix to the name of the attribute.
- This is more of a a guideline "We are all adults and know what we're doing."
- You can add a double underscore __ to obstify de name of the attribute outside of the class. This is to prevent naming collissons when inheriting classes.

```
class Person:
   def __init__(self, name):
      self._name = name
   def tell(self):
      return(f'I am {self._name}')

# ------------------------------------------------------------

p = Person('Albert')
print( p.tell() )
```

Attributes

- Attributes are typically initialized in the __init__ method.
- Attributes are dynamic and can be assigned a value anywhere.

```
class Person:
    def __init__(self, name, residence = 'unknown'):
        self.__name = name
        self.__residence = residence
    def tell(self):
        return(f'I am {self.__name} from {self.__residence}')

# ------------------------------------------------------------

p = Person('Albert', 'Amsterdam')
print( p.tell() )
```

Object attributes vs Class-wide attributes

- Most attributes are attributes related to objects.
- This means that an object has to be made first, before you can use or access that attribute.
- It can only work for that specific object.
- But sometimes there are attributes that you want to use from the class itself.
- These are called: Class-wide attributes

Object methods vs Class-wide methods

- Most methods are methods related to objects.
- This means that an object has to be made first, before you can use that method.

- It can only work for that specific object.

- But sometimes there are methods that you want to use from the class itself.

- These are called: Class-wide methods

Class-wide attributes

- Class wide attributes are attributes that are related to the class instead of to an object of that class.
- Class wide attributes can be accessed by all objects of the class

```
class Mathematics:
   pi = 3.14159
   e = 2.71828

print( Mathematics.pi )
print( Mathematics.e )
```

Class-wide methods

- Most methods are methods related to objects, but:
- Class-wide methods are methods related to the class.
- A method can be indicated as a class-wide methode with a decorator **@classmethod** or **@staticmethod**.

```
class Mathematics:
    @staticmethod
    def power1(x, n):
        result = 1
        for __ in range(n):
            result *= x
        return result
    @classmethod
    def power2(cls, x, n):
        return cls.power1(x, n)

print(Mathematics.power1(2, 4))
print(Mathematics.power2(2, 4))
```

Example

```
class Person:

        __slots__ = ('__name', '__residence')

   def __init__(self, name, residence = 'unknown'):
      self.__name = name
      self.__residence = residence

   def tell(self):
      return('I am {} and I live in {}'\
            .format(self.__name, self.__residence))

   def move(self, new_residence):
      self.residence = new_residence


p = Person('Albert', 'Amsterdam')
print( p.tell() )
p.move('Eindhoven')
print( p.tell() )
```

Special Methods

- A class can have many different special methods.
- Also called magic methods.
- A magic method is called by Python in all kind of situations, typically when operators are used

__init__	__eq__	__gt__
__del__	__ne__	__ge__
__str__	__lt__	__add__
__repr__	__le__	__sub__

computrain

Inheritance

- Classes can be reused by inheritance
- The original class is called the parent class, the superclass or the base class
- The new class is called the child class, the subclass or the derived class
- Enclose the parent in parentheses after the new class name.
- All the attributes and methods of the parent class are avaiable in the child class.
- In the __init__ method of the child class always first call de __init__ method of the parent class with the **super** method

```
class Vector(object):
```

```
class ChildClass(ParentClass):
    def __init__(self, name):
        super().__init__(name)
```

Let’s make classes together

- The trainer will make with you a class diagram in a tool or on a blackboard.
- Then, Python code will be made by forward engineering.
- After that, you have to do it yourself in exercises 1.6 Bank Account and 1.7 Class Car.

Democodes

- There are more examples of democode for OO programming in Python

- Animal.py
- Auto.py
- DemoOO.py
- Dier.py
- Person.py
- Person2.py

Exercise 1.6

Bankaccount

Create a Bankaccount class, create serveral Bankaccount objects and demonstrate that you can deposit and withdraw amount to the account.

Tips:

- Create a class Bankaccount
- Add attributes in the **__init__** method. Attributes should be __balance and __holder.
- Add the methods: **deposit** and **withdraw** that take an amount argument and a third method **info** that returns information about the account.
- Instantiate serveral bankaccount objects and demonstrate the working of the class

Exercise 1.7

Class Car

- Create a class named **Car**
- Add the __init__ method and set several attributes like _**make**, _**type** and _**color**
- Set the _**mileage** attribute to 0
- Create a method **info** that describes the car and the mileage
- Create a method **drive** that takes an amount of kilometers and adds that to the mileage.
- Test you class by instantiating a car and calling the methods

Exercise 1.8

Vector class

Create a 2d-Vector class. Also add operator overloading for the + sign to add two vectors together.

Tips:

- Build a class called Vector
- Add two attributes: x and y
- Implement the __init__ method that takes two arguments: x and y
- Implement the __str__ and __repr__ methods.
- Implement the __add__ method the define the adding of two vectors.
- Test your class by creating two vectors and adding these together.

Exercise 1.9

Make your own classes

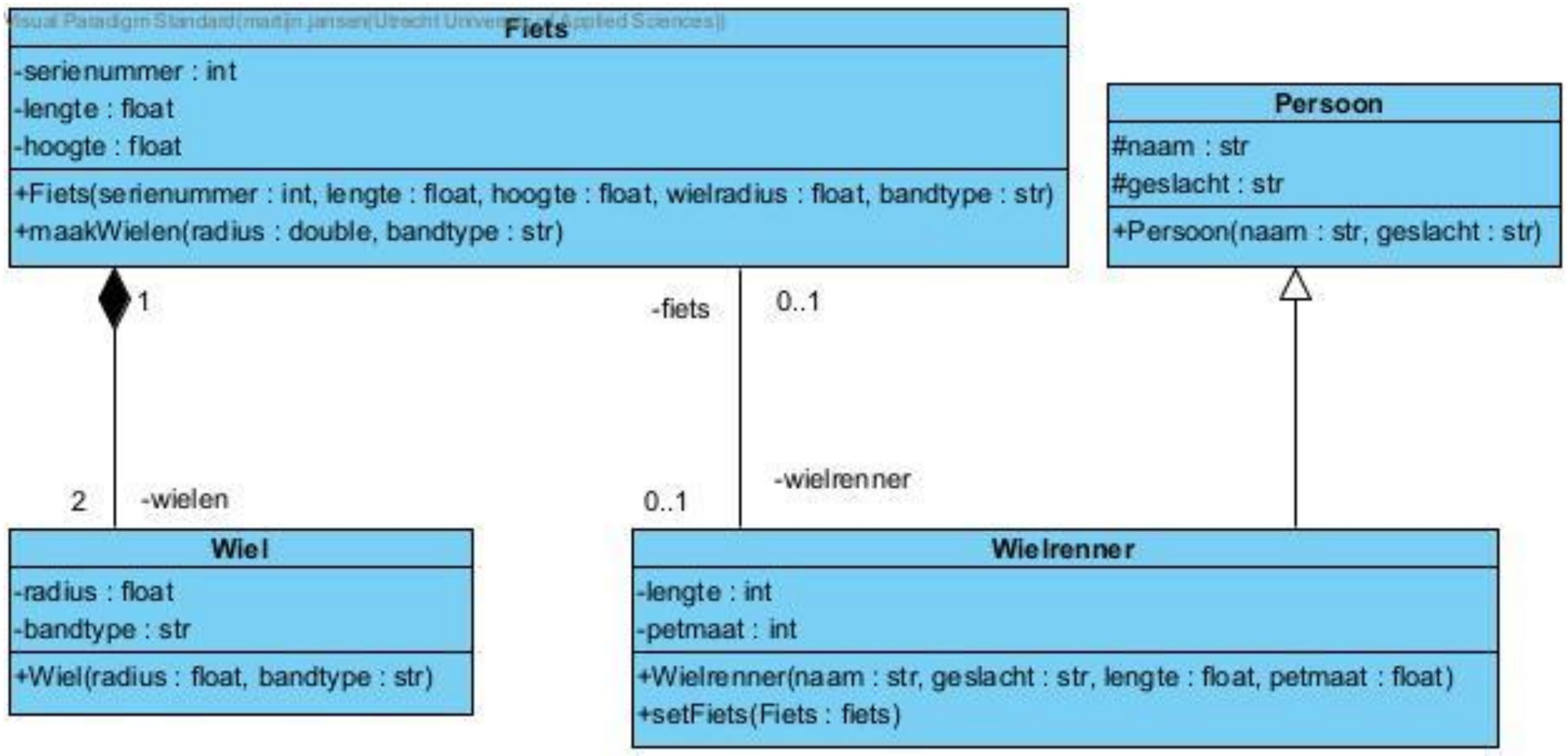
- Create your own classes for your own work.
- First, think about the classes you need for your work
- Make UML class diagram.
- Ask the trainer if everything is correct in UML what you are thinking about to make.
- If the trainer gives you a GO, you can begin programming in Python.

Exercise 1.9

Make your own classes

- Example for the trainer:

Forward engineering Python file: Fiets.py


Visual Paradigm Standard(martijn jansen(Utrecht University of Applied Sciences))
Fiets
-serienummer : int
-lengte : float
-hoogte : float
+Fiets(serienummer : int, lengte : float, hoogte : float, wielradius : float, bandtype : str)
+maakWielen(radius : double, bandtype : str)
Persoon
#naam : str
#geslacht : str
+Persoon(naam : str, geslacht : str)
1
2
-wielen
-fiets
0..1
-wielrenner
0..1
Wiel
-radius : float
-bandtype : str
+Wiel(radius : float, bandtype : str)
Wielrenner
-lengte : int
-petmaat : int
+Wielrenner(naam : str, geslacht : str, lengte : float, petmaat : float)
+setFiets(Fiets : fiets)

Exercise 1.9: Results

Presentations of classes

- Each group has made their own classes in Python according a class diagram.
- Each group has to present their work to each others.
- The trainer gives feedback during and after the presentations.
- These presentations can also be given in the morning of lesson day 2.

computrain

Python Standard Library

- The Python Standard Library consists of more than 200 modules and packages
- The Python has "batteries included"

os	csv	json	subprocess
os.path	collections	xml	socket
sys	array	sqlite3	asyncio
string	decimal	zipfile	urllib
re	fractions	time	http
math	statistics	argparse	tkinter
random	pathlib	logging	doctest
datetime	pickle	threading	unittest
calendar	shelve	multiprocessing	timeit

sys - System-specific

- System-specific parameters and functions

version	argv
version_info	exit
path	stdin / stdout / stderr

```
import sys

# get Python version
print(sys.version)

# add directory to sys.path
sys.path.append(r'c:\pythondev')
```

os - Operating system interfaces

- The **os.path** module provides a portable way of using operating system dependent functionality.

rename	listdir	system
remove	path	getenv
mkdir	pipe	getpid
makedirs	scandir	getppid
chdir	walk	kill
getcwd	fork	wait
rmdir	exec	nice

```
import os

# set current working directory
os.chdir(r'c:\pythondev')
print(os.getcwd())
```

pathlib - Object-oriented filesystem paths

This module offers classes representing filesystem paths with semantics appropriate for different operating systems.

Path	**parts**	**cwd()**
PurePath	**drive**	**home()**
WindowsPath	**root**	**chmod()**
PureWindowsPath	**anchor**	**exists()**
	parent	**is_dir()**
operator /	**parents**	**is_file()**
		glob()

```
from pathlib import Path

p = Path('.')

list(p.glob('**/*.py'))
```

shutil

High-level file operations

- **copy**
- **copytree**
- **rmtree**
- **move**
- **disk_usage**
- **chown**

glob

- The glob module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell.

```
import glob
glob.glob('./file[0-9].*')
```

subprocess

- The subprocess module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.

run

```
import subprocess

subprocess.run(["ls", "-l"])

subprocess.run(["ls", "-l", "/dev/null"], capture_output=True)
```

tempfile

This module generates temporary files and directories.

It works on all supported platforms.

- **TemporaryFile**
- **NamedTemporaryFile**
- **TemporaryDirectory**

```
import tempfile

with tempfile.TemporaryFile() as fp:
    fp.write(b'Hello world!')
    fp.seek(0)
    fp.read()
```

Democodes

- DemoDatetime.py
- DemoGlob.py
- DemoPathlib.py
- DemoSubprocess.py
- DemoSys.py

datetime - Basic date and time types

- The datetime module supplies classes for manipulating dates and times.

```
class datetime.date
class datetime.time
class datetime.datetime
class datetime.timedelta
class datetime.tzinfo
class datetime.timezone
```

strftime
strptime

```python
from datetime import date

d = date(2020, 2, 28)

print(d.strftime('%Y-%m-%d'))
```

Exercise 2.1

sys

- Get the current version of Python
- Return the message you are currently running Python version …

Exercise 2.2

OS

- Use the os library to get the contents of a directory in a list.

Exercise 2.3

datetime

- input a date and print the date in another format

string - Common string operations

- String methods

count	isnumeric	split
find	join	strip
format	lower	title
index	replace	

```
# remove vowels
s = 'an example text'
vowels = "aeiou"
trans = str.maketrans("", "", vowels)
result = s.translate(trans)
```

re - Regular expression operations

- Online: https://www.regex101.com/#python

```
search(pattern, string, flags)
match(pattern, string, flags)
findall(pattern, string, flags)
sub(pattern, repl, string, max=0, flags)
compile(pattern)
```

```
import re
match = re.search(r'@([\w\.]+)\b', 'albert@gmail.com', re.I)
if match:
    for group in match.groups():
        print('Domain: ', group)
```

math - Mathematical functions

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

random - Pseudo-random numbers

- Generate pseudo-random numbers

```
random.seed()
random.randrange(start, stop)
random.randint(a, b)
random.choice(sequence)
random.choices(sequence, k=1)
random.shuffle(sequence)
random.sample(sequence, n)
random.random()
```

```
import random
items = 'abcdefghiklmnopqrstuwvxyz0123456789'
sample = random.sample(items, 3)
```

json - JavaScript Object Notation

- JSON encoder and decoder

```
json.dump(object, file)
json.dumps(object)
json.load(file)
json.loads(string)
```

```
import json

s = json.dumps([1,2,3,{'4': 5, '6': 7}])

with open('bestand.json', 'w') as f:
json.dump([1,2,3,{'4': 5, '6': 7}], f)

json.loads('[1,2,3,{"4":5,"6":7}]')
```

pickle - Python object serialization

- A **shelf** is a persistent, dictionary-like object that stores any arbitrary Python that can be pickled.

pickle.dump(object, file)
pickle.dumps(object)
pickle.load(file)
pickle.loads(string)

```
import pickle

class User:
   def saveToPickle(self):
      with open('user.pickle','wb') as f:
         pickle.dump(self, f)
   def loadFromPickle(self):
      with open('user.pickle','rb') as f:
         self.__dict__.update(pickle.load(f).__dict__)
   @classmethod
   def createFromPickle(cls):
      with open('user.pickle','rb') as f:
         return pickle.load(f)
```

xml

- The **xml.etree.ElementTree** module implements a simple and efficient API for parsing and creating XML data.
- This module provides limited support for **XPath** expressions for locating elements in a tree. https://www.w3schools.com/xml/xpath_intro.asp
- ElementTree provides a simple way to build XML documents and write them to files.

```python
import xml.etree.ElementTree as ET

tree = ET.parse('data.xml')
root = tree.getroot()

print(root.attrib)

for element in root.findall('//name'):
   print(element.text)
```

statistics

- This module provides functions for calculating mathematical statistics of numeric (Real-valued) data.

bisect_left	**median**	**pstdev**
bisect_right	**median_grouped**	**pvariance**
collections	**median_high**	**stdev**
groupby	**median_low**	**variance**
harmonic_mean	**mode**	
mean	**numbers**	

```
import statistics
numbers = [23, 64, 86, 23, 54, 76, 98, 21]
print('Median:', statistics.median(numbers))
print('Mean:', statistics.mean(numbers))
print('St.Dev.:', statistics.stdev(numbers))
```

doctest

- The doctest module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown.

```
def square(n):
    """Calculate the square of n.

    >>> [square(n) for n in range(6)]
    [0, 1, 4, 9, 16, 25]
    """
    return n ** 2


if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

Exercise 2.4

pickle

- Create a datastructure and store this in a pickle file. Create a second python script that reads pickle file and restores the data in the data structure.

Exercise 2.5

Statistics

Create a function that calculates and returns the mean, median and mode of a list of numbers.

Tips:

- Define a function as **def central_measures(numbers)**
- Calculate the measures:
  - The **mean** is the sum of the values divided by the number of values
  - The **median** is middle value of the sorted list of values
  - The **mode** is the most frequently occuring value
- Return the measures as a tuple with **return mean, median, mode**
- Call the function with a list of arbitrary numbers
- Print the result

Exercise 2.6

doctest

- Create a function and add a docstring with doctests for the function.

unittest - Unit testing framework

- There is also an **assert** statement

```
import unittest


class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())


if __name__ == '__main__':
    unittest.main()
```

```
assertEqual(a, b)
assertNotEqual(a, b)
assertTrue(x)
assertFalse(x)
assertIs(a, b)
assertIsNot(a, b)
assertIsNone(x)
assertIsNotNone(x)
assertIn(a, b)
assertNotIn(a, b)
assertIsInstance(a, b)
assertNotIsInstance(a, b)
```

csv – Comma Seperated Values

- CSV File Reading and Writing

reader
writer
DictReader
DictWriter

```
import csv

filename = 'data.csv'

with open(filename) as f:
    reader = csv.DictReader(f, delimiter=';')
    for row in reader:
        print(row['first_name'], row['last_name'])
```

decimal

- The decimal module provides support for fast correctly-rounded decimal floating point arithmetic.

```
from decimal import Decimal

d1 = Decimal('0.1')
d2 = Decimal('0.2')

result = float(d1 + d2)
```

fractions

- The fractions module provides support for rational number arithmetic.

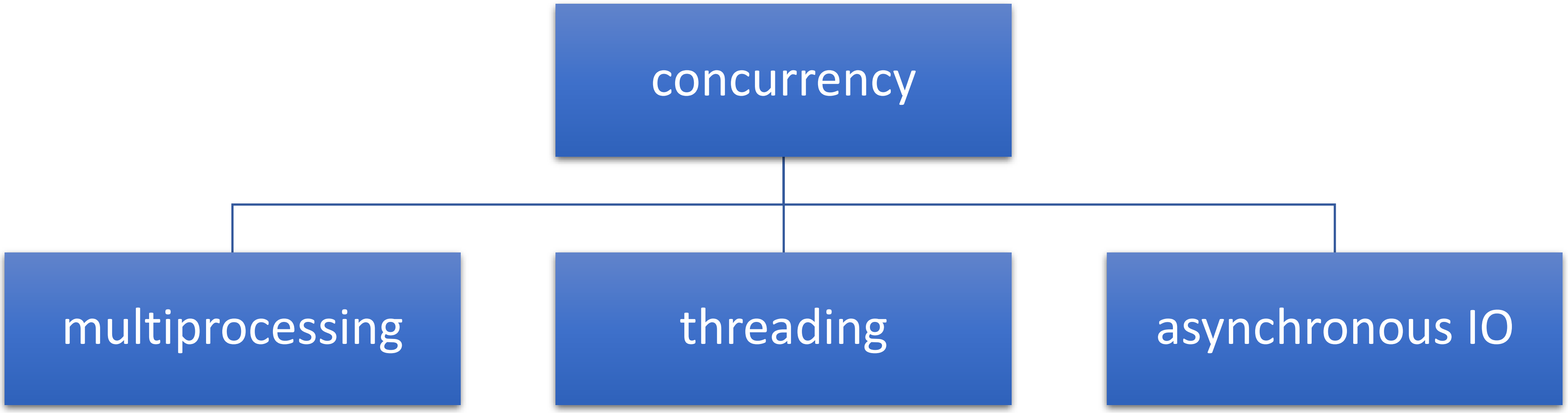
```
from fractions import Fraction

d1 = Fraction(1, 3) # => 1/3
d2 = Fraction(1, 2) # => 1/2

result = d1 + d2 # => 5/6
```

sqlite3

- DB-API 2.0 interface for SQLite databases
- PEP 249 - Database API Specification 2.0

```
import sqlite3

conn = sqlite3.connect('example.db')
c = conn.cursor()

c.execute("""CREATE TABLE stocks
        (date text, trans text, symbol text, qty real, price real)""")

c.execute("""INSERT INTO stocks
        VALUES ('2006-01-05’, 'BUY’, 'RHAT’, 100, 35.14)""")

conn.commit()

for row in c.execute('SELECT * FROM stocks ORDER BY price'):
    print(row)

conn.close()
```

Concurrency



Python Standard Library:

- **multiprocessing** package
- **threading** package
- **asyncio** package and **async/await** keywords (introduced in Python 3.4)

Comparison

	Multiprocessing	Threading	Asynchronous IO
Package	multiprocessing	threading	asyncio
Class	Proces	Thread	Coroutine
Python	Class Proces	Class Thread	Keywords async, await
Data sharing	Message	Shared data	
Usage	CPU intensive	IO intensive	IO intensive

Proces versus Thread

- True parallelism in Python is achieved by creating multiple processes, each having a Python interpreter with its own separate GIL.

Process	**Thread**
processes run in separate memory (process isolation)	threads share memory
uses more memory	uses less memory
children can become zombies	no zombies possible
more overhead	less overhead
slower to create and destroy	faster to create and destroy
easier to code and debug	can become harder to code and debug

Python GIL

- A global interpreter lock (GIL) is a mechanism used in Python interpreter to synchronize the execution of threads so that only one native thread can execute at a time, even if run on a multi-core processor.
- The C extensions, such as numpy, can manually release the GIL to speed up computations. Also, the GIL released before potentionally blocking I/O operations.
- Note that both Jython and IronPython do not have the GIL.

threading - Thread-based parallelism

- **Thread**
  - start
  - run
  - join
  - name
- **active_count**
- **current_thread**
- **main_thread**

```
import time
from threading import Thread

def myfunc(i):
    print "sleeping 5 sec from thread %d" % i
    time.sleep(5)
    print "finished sleeping from thread %d" % i

for i in range(10):
    t = Thread(target=myfunc, args=(i,))
    t.start()
```

asyncio

- At the heart of async IO are **coroutines**. A coroutine is a specialized version of a Python generator function.

```
import asyncio

async def count():
   print("One")
   await asyncio.sleep(1)
   print("Two")

async def main():
   await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
   import time
   s = time.perf_counter()
   asyncio.run(main())
   elapsed = time.perf_counter() - s
   print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

multiprocessing

- The multiprocessing library is based on spawning Processes.
- A process starts a fresh Python interpreter thereby side-stepping the Global Interpreter Lock
- The multiprocessing module allows the programmer to fully leverage multiple processors on a given machine.

```
from multiprocessing import Process

def f(name):
   print('hello', name)

if __name__ == '__main__':
   p = Process(target=f, args=('bob',))
   p.start()
   p.join()
```

logging - Logging facility for Python

- Setup with **basicConfig**
- Logging Levels: DEBUG, INFO, WARNING, ERROR, CRITICAL

```
import logging

logging.basicConfig(
    filename = None, # or to a file 'example.log',
    level = logging.ERROR,
    format = '%(asctime)s.%(msecs)03d - %(message)s',
    datefmt = '%Y-%m-%dT%H:%M:%S')

logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
logging.error('Watch out!')
logging.critical('ERROR!!!!!')
```

timeit

- Measure execution time of small code snippets.

```
from timeit import timeit

timeit('"-".join(str(n) for n in range(100))', number=10000)
timeit(lambda: "-".join(map(str, range(100))), number=10000)
```

zipfile

- The ZIP file format is a common archive and compression standard. This module provides tools to create, read, write, append, and list a ZIP file.

```
import zipfile
import pandas as pd

with zipfile.ZipFile("FinalExam.zip") as z:
   with z.open("AdvWorksCusts.csv") as f:
      df_Customers = pd.read_csv(f)

   with z.open("AW_AveMonthSpend.csv") as f:
      df_AveMonthSpend = pd.read_csv(f)

   with z.open("AW_BikeBuyer.csv") as f:
      df_BikeBuyer = pd.read_csv(f)
```

tarfile

- The tarfile module makes it possible to read and write tar archives, including those using gzip, bz2 and lzma compression.

```
import tarfile

t = tarfile.open('example.tar.gz', 'r')
print("Files in TAR file:")
print(t.getnames())
```

Democodes

- DemoAsyncio.py
- DemoDocstring.py
- DemoDocstringPPT.py
- DemoSubprocess.py
- DemoThreading.py

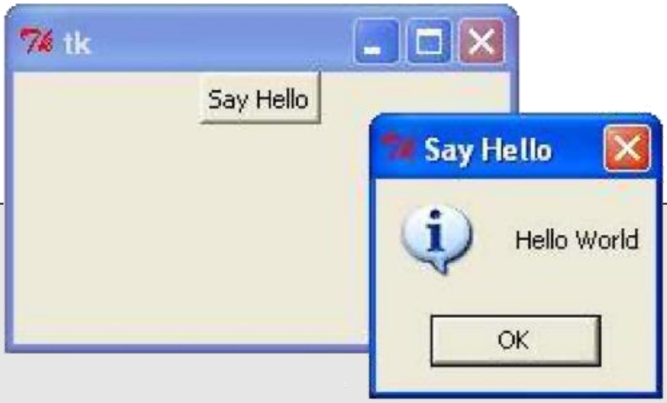
GUI Frameworks

- **TkInter** - The traditional Python user interface toolkit.

- **PyQt** - Bindings for the cross-platform Qt framework.
- **PySide** - PySide is a newer binding to the Qt toolkit
- **wxPython** - a cross-platform GUI toolkit that is built around wxWidgets
- **Win32Api** - native window dialogs
- **PyMsgBox**

tkinter

- The tkinter package ("Tk interface") is the standard Python interface to the Tk GUI toolkit.
- There are also Standard Dialogs



```
import tkinter as tk
import tkMessageBox

top = tk.Tk()

def hello():
   tkMessageBox.showinfo("Say Hello", "Hello World")

btn1 = tk.Button(top, text = "Say Hello", command = hello)
btn1.pack()

top.mainloop()
```

The Python Package Index - PyPI

- The official third-party software repository for the Python programming language
- The Python Package Index is a repository of software for the Python programming language. There are currently > **300000** packages.
- Install packages with the **pip** command.

```
pip list
pip search

pip install numpy
pip install scipy
pip install matplotlib
pip install pandas
pip install requests
pip install pyodbc
```

pyodbc - Accessing ODBC databases

PEP 249 -- Python Database API Specification v2.0

```
import pyodbc

conn = pyodbc.connect(
    'DRIVER={SQL Server};'
    'SERVER=localhost\SQLEXPRESS;'
    'DATABASE=mijndatabase;'
    'UID=username; PWD=pa55w0rd')

sql = 'SELECT customers.* FROM customers'

cursor = conn.cursor()
for row in cursor.execute(sql):
    print("{}, {}".format(row.name, row.residence)

cursor.close()
conn.close()
```

computrain

requests – HTTP for Humans

- **Requests** is an elegant and simple HTTP library for Python, built for human beings.

```
import requests

url = "http://api.openweathermap.org/data/2.5/weather"
url += "?appid=d1526a9039658a6f76950cff21823aff"
url += "&units=metric"
url += "&mode=json"
url += "&q=New York"

response = requests.get(url)
if (response.status_code == 200):
    body = response.text
    decoded = response.json()
    temperature = decoded['main']['temp']
else:
    print("Error for city %s" % (city))
```

Exercise 2.7

Get the weather in New York

Use requests to query openweathermap.org for the weather in a specified city.

Tips:

- import **requests**
- build the url (see https://openweathermap.org/current)
  use: **appid=d1526a9039658a6f76950cff21823aff**
- use the following code to get the response:
  response = requests.get(url)
- use json to decode the response into a Python dictionary
  response.json()
- get and print the temperature

The Python Package Index - PyPI

- Go to the website: https://pypi.org/
- The trainer will give a demonstration how to install a Python Package Index.
- One of the following examples can be used:
  - Open Weather Map
  - Requests
  - HTML Converter / Generator
  - PDF Converter / Generator
  - Excel Converter to …
- Can you give more examples for demonstration?

Exercise 2.8

Find Python packages yourself

- *This exercise is a group exercise!*
- Groups will be made. Each group exists of 2 or 3 people at most. Each group will have to find and install Python packages in the Python Package Index.
- First, each group chooses a subject; something that is related to someone's work.
- Second, between 30 and 60 minutes, each group will find and install several Python packages about the chosen subject.
- At the end, each group has to present their findings to the trainer and other groups. The trainer will give feedback.

Python for Data Science

- Python is used very much in Data Science.
- Therefore, Aanconda has been made. Anaconda is a big program / Python version with all the data science libraries already in it.
- So, you don't have to install all the data science libraries from the Python Package Index.
- The trainer will tell something about it and he will show some examples.

https://pandas.pydata.org/docs/getting_started/10min.html#min

numpy

- NumPy is the fundamental package for scientific computing with Python.
- NumPy's main object is the homogeneous multidimensional array.
- Vectorized operations

```
import numpy as np

a = np.array([1,2,3,4])
b = np.array( [ (1.5,2,3), (4,5,6) ] )
c = np.narray( [ [1,2], [3,4] ], dtype=complex )

np.zeros( (3,4) )
np.arange( 0, 2, 0.4 )    # array([ 0., 0.4, 0.8, 1.2, 1.6, 2.0])
np.linspace( 0, 2*pi, 100 ).  # 100 numbers from 0 to 2*pi
```

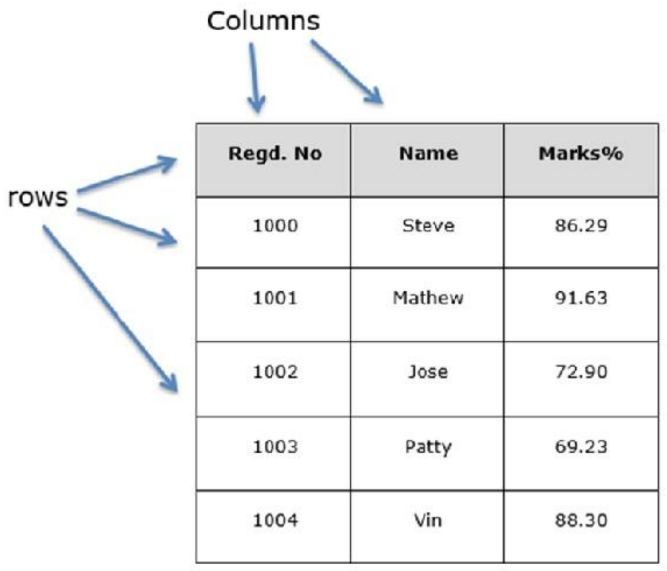
scipy

- It provides many user-friendly and efficient numerical routines, such as routines for numerical integration, interpolation, optimization, linear algebra, and statistics.

Clustering
Constants
Discrete Fourier transforms
Integration
Interpolation
Input and output
Linear algebra
Miscellaneous routines
Multi-dimensional image processing
Orthogonal distance regression
Optimization and Root Finding
Signal processing
Sparse matrices
Sparse linear algebra
Compressed Sparse Graph Routines
Spatial algorithms and data structures
Special functions
Statistical functions
Statistical functions for masked arrays
Low-level callback functions

pandas

- Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool.
- **Series** is a one-dimensional labeled array capable of holding any data type
- **DataFrame** is a 2-dimensional labeled data structure with columns of potentially different types.



Regd. No	Name	Marks%
1000	Steve	86.29
1001	Mathew	91.63
1002	Jose	72.90
1003	Patty	69.23
1004	Vin	88.30

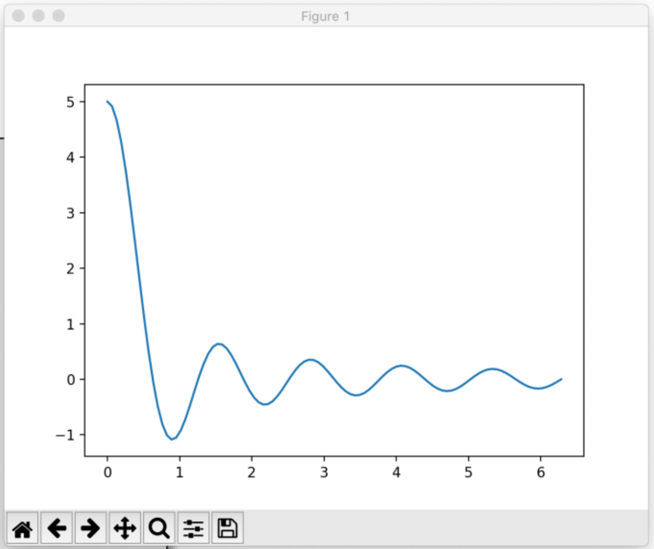
https://pandas.pydata.org/docs/getting_started/10min.html#min

matplotlib

- Matplotlib is a Python 2D plotting library which produces publication quality figures

Line plot	**3D plot**	**Polar plot**
Histogram	**Image plot**	
Scatter plot	**Contour plot**	

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace( 0.000001, 2*np.pi, 100 )
y = 1/x * np.sin(5*x)

plt.plot(x, y)
plt.show()
```



Python for Data Science

- If you want to learn more about this, subscribe to the course:
- **Python voor Data Science**
- At Computrain!

https://pandas.pydata.org/docs/getting_started/10min.html#min

django

MVC Framework

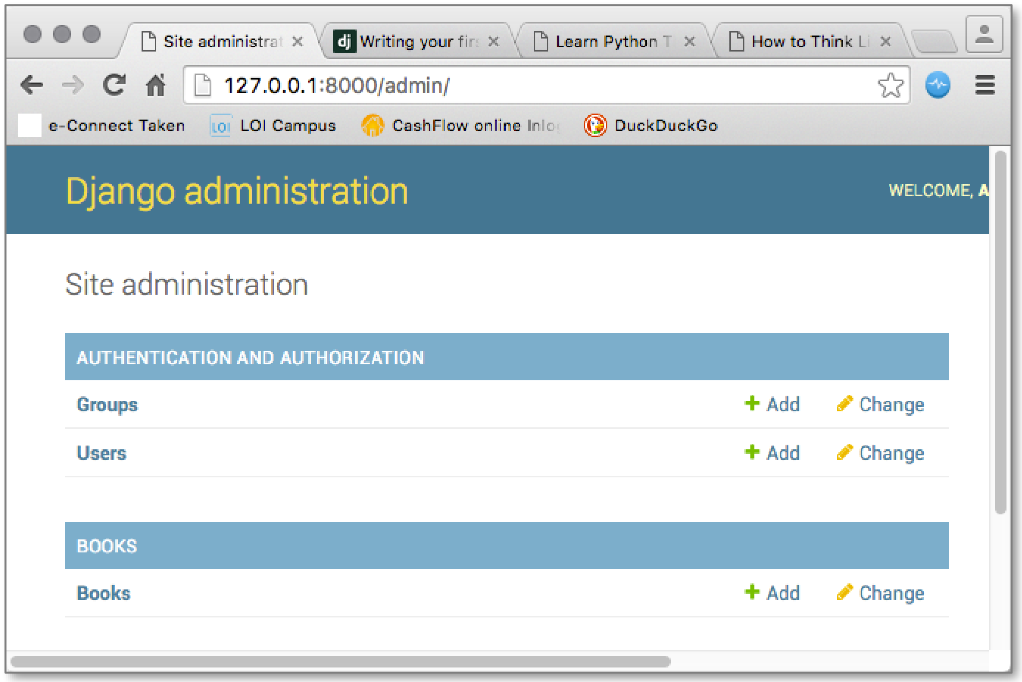
Models

Object-Relational Mapping

URL Mapping

Views

HTML Templates



Django administration

Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change

BOOKS		
Books	+ Add	Change

Command line bootstrap:

```
$ mkdir django-demo
$ cd django-demo
$ virtualenv -p python3.5 venv
$ . venv/bin/activate
$ pip install django
$ django-admin
$ django-admin startproject demo
$ python manage.py createsuperuser
$ python manage.py startapp books
$ python manage.py makemigrations
$ python manage.py migrate
$ python manage.py runserver
```

Python Distributions

- Anaconda
- Active Python
- Python (X,Y)
- IPython
- Enthought Canopy
- Sage
- PyPy
- Pocket Python
- Portable Python

Implementations

- CPython reference implementation
- IronPython (Python running on .NET)
- Jython (Python running on the Java Virtual Machine)
- PyPy (A fast python implementation with a JIT compiler)
- Stackless Python (Branch of CPython supporting microthreads)
- MicroPython (Python running on micro controllers)

Style Guide for Python Code

- PEP 8 - Style Guide for Python Code

Virtual Environment

- Seperated enviroments

```
$ virtualenv -p python3.5 venv
$ . venv/bin/activate
```

- Requirements file

```
$ pip list > requirements.txt
$ pip install -r requirements.txt
```

Ducktyping

- “If it looks like a duck and quacks like a duck, it must be a duck.”

- A programming style which does not look at an object’s type to determine if it has the right interface; instead, the method or attribute is simply called or used

EAFP versus LBYL

- EAFP: "it's easier to ask for forgiveness than permission
- LBYL: "look before you leap"

computrain

Python Advanced

Thank you for your attention!

python™

Goodbye!