CENTRALE NANTES

LS2N LABORATOIRE DES SCIENCES DU NUMÉRIQUE DE NANTES

Introduction to Python

Prof. Diana Mateus

With contribution from Marco Esposito @ TUM

What is Python?

An interpreted dynamic programming language

Interpreted: does not require an entire program to run instructions

Dynamic: allows to change object structure at runtime

Why Python?

- Mature and popular, in particular for scientific computing
- … and yet rapidly evolving
- Large set of libraries and library bindings (SciPy, OpenCV, tensorflow, VTK,…)
- Focused on productivity and readability
- Reasonable performance, easy to integrate with C/C++

On the downside

- Usually slower than C or C++ (but is of course a higher level language)

- Not appropriate for
  - mobile development (although Raspberry Pi)
  - Memory intensive tasks
  - Graphics
  - Multi-processor/multi-core work

Python 2 or Python 3?

Short answer: Python 3 ALWAYS!

With Python 3 a lot of compatibility-breaking changes were introduced to “clean up” the language

Python 2 has been deprecated for years and will soon be removed as system default

Using version 3 now will avoid a migration later

Print 3/2

```
# python 2

print 3/2  # 1

from __future__ import division

print 3/2  # 1.5



# python 3

print(3/2)  # 1.5
```

Getting Python

- Quick and painless: **Anaconda 3**
  - (download and always click “yes”)
  - Includes the Python interpreter, many useful libraries and an IDE (Spyder)
  - Available for all Operating Systems

- On **Mac:** python 2 installed by default, not python 3

Use Anaconda 3 or Homebrew for better control

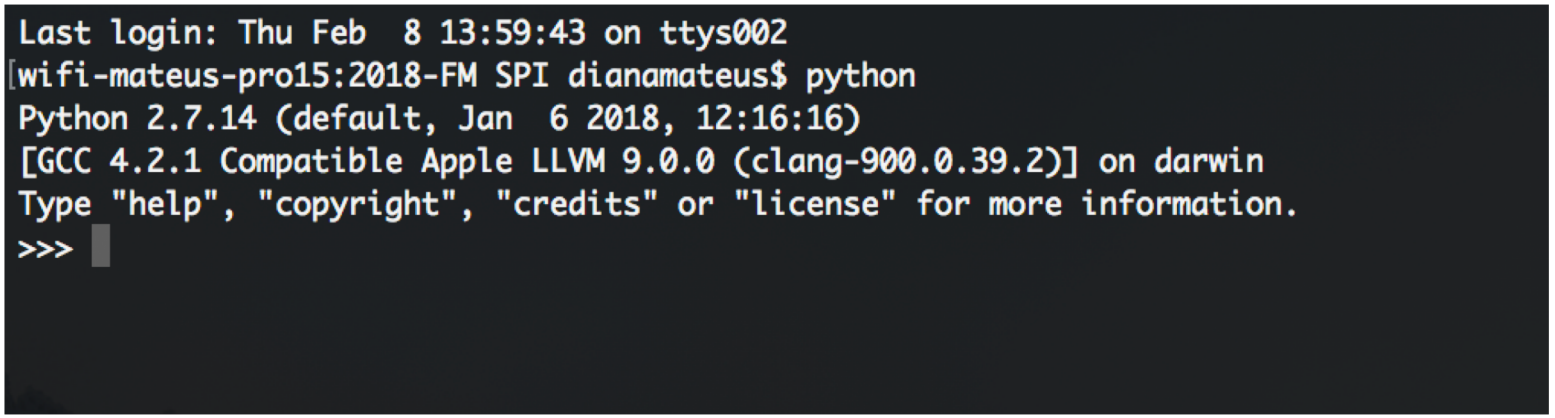
- On **Linux:** python 2 by default, sometimes python 3

Use package manager or pip

The Python command

The "**`python`**" command is a Python interpreter:
takes Python code as input and executes it

If run without arguments, it will open a Read-Evaluate-Print-Loop (REPL) console

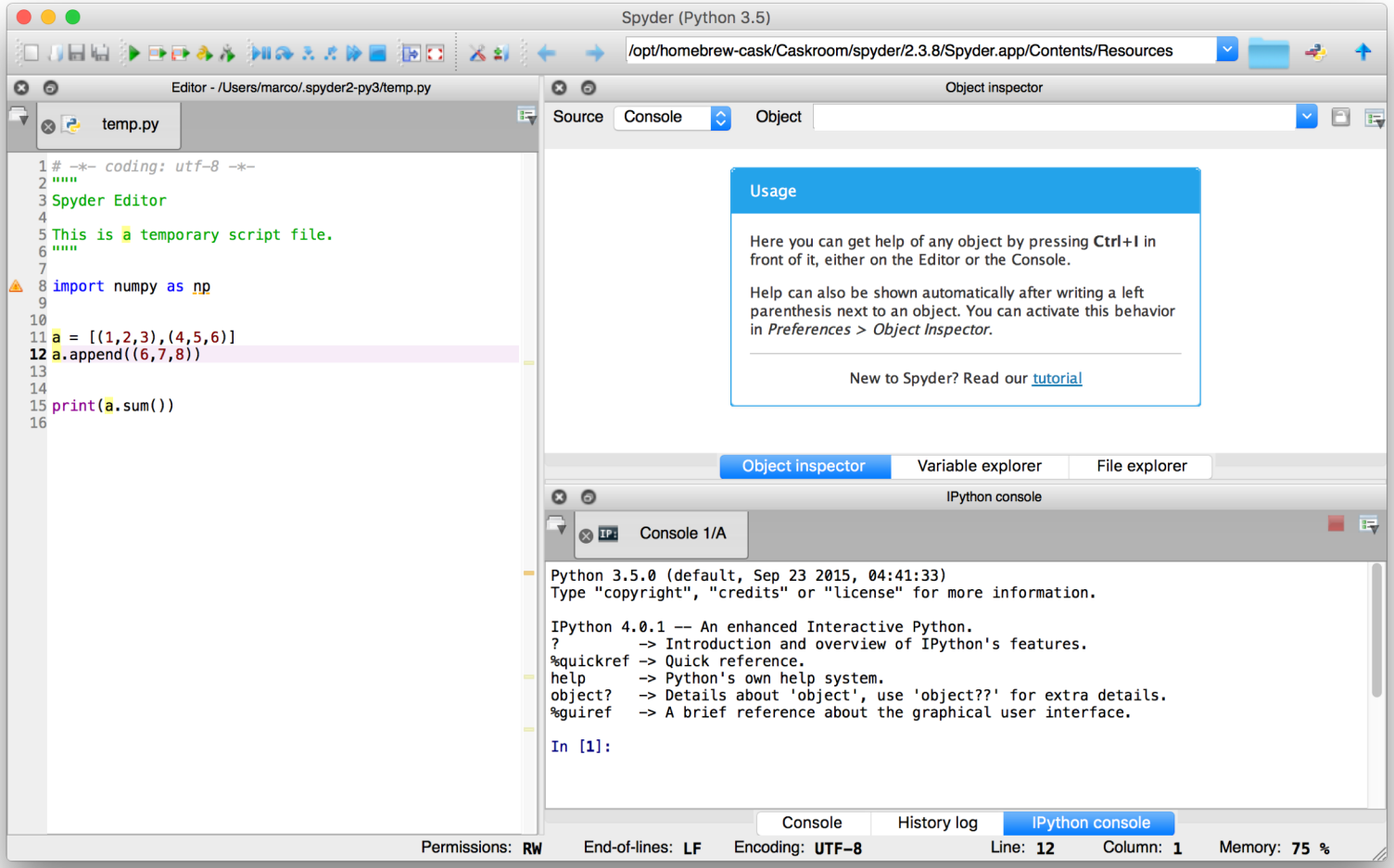


If the path to a file is given as argument, the content of the file will be interpreted as sequential Python instructions

Exit with Ctrl-D or `quit()`

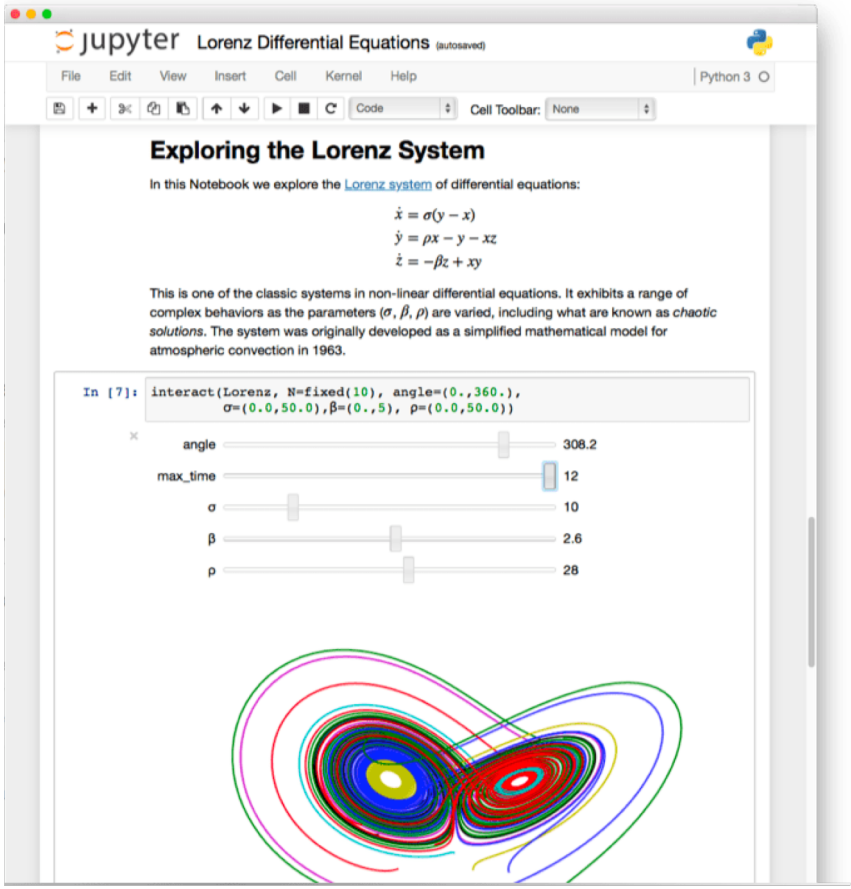
Spyder

MATLAB-like environment, good starting point



Jupyter Notebooks

- an **open-source web** application
- Useful to create and share **documents** that contain **live code**, equations, visualizations and narrative text.
- Uses: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.
- has support for over **40 programming languages**, including Python, R, Julia, and Scala.



Source: jupyter.org/

Disclaimer

What follows is just a brief intro to Python to get you up and running

We won't cover all characteristics that make Python special

You are wholeheartedly advised to continue exploring on your own:

- http://www.diveintopython3.net is a very good introduction
- http://learnpythonthehardway.org/book/ is also a good tutorial
- PyCon YouTube channel https://www.youtube.com/channel/UCgxzjK6GuOHVKR_08TT4hJQ
  - In particular Raymond Hattinger's and Brandon Rhodes' videos for starting
- https://www.codingame.com/ is a fun way to practice
- For everything else, there's your favorite web searcher

Contents

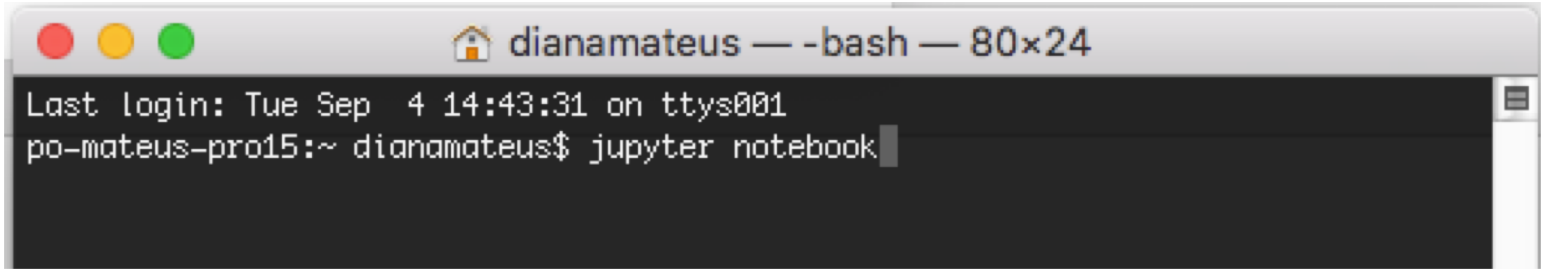
- Language Fundamentals
- Program structure
- Numpy
- Games

Language Fundamentals

Getting started with a Jupyter Notebook

Running Jupyter

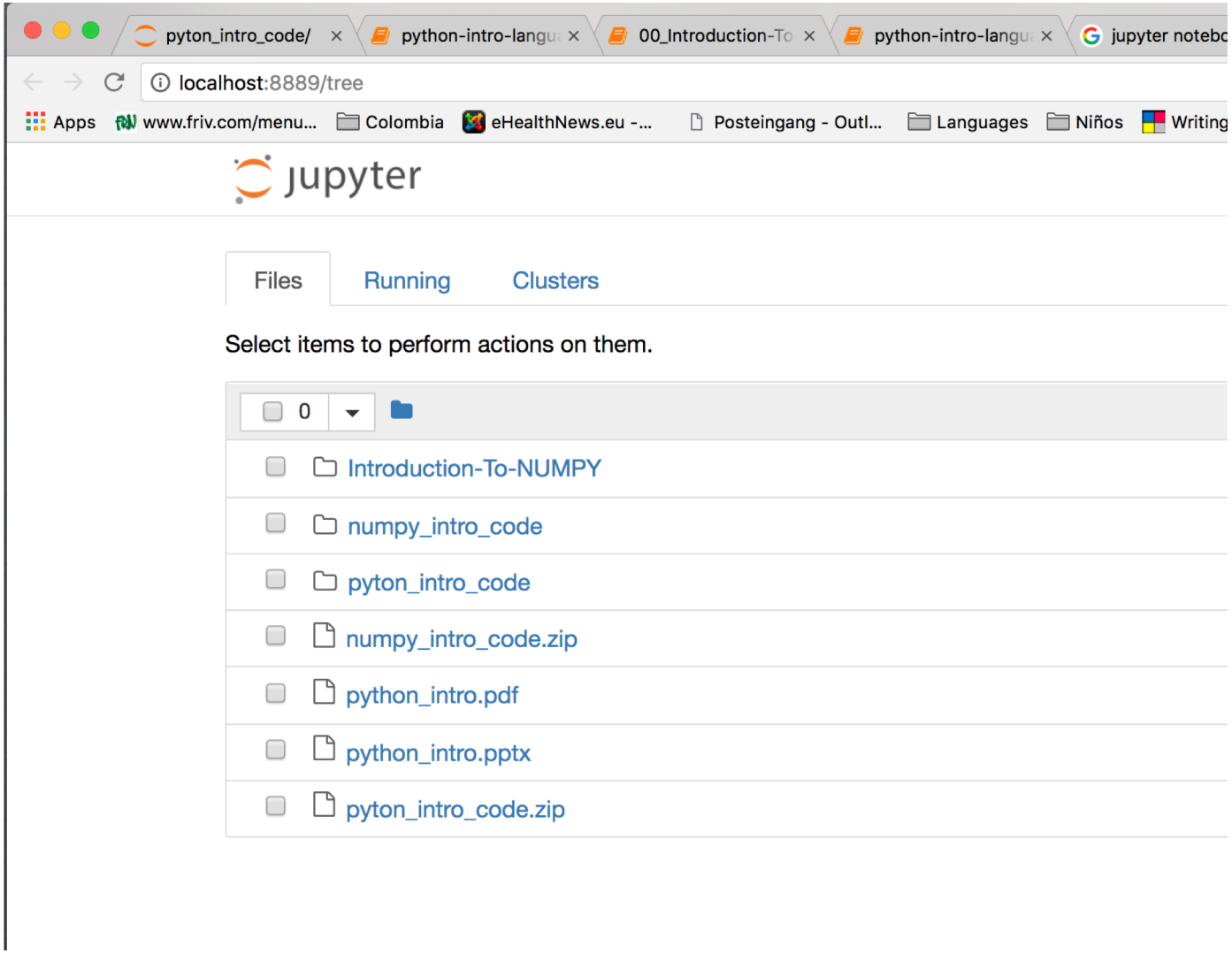
- run from the terminal the command « `jupyter notebook` »



- This should open a web browser automatically and show you the contents of your current directory.

- If this does not happen automatically, open a browser and navigate to http://localhost:8888

Getting started with a Jupyter Notebook



Python is object-oriented

Everything in Python is an object

An object holds a **value** and offers a set of **operations** on it

An object can be bound to a **name**

An object can be bound to more than one name

When no name refers to an object any more, it is deleted by the garbage collector

```
5.0.is_integer()  # True
5.1.is_integer()  # False

a = [1, 2]
b = a
a.append(3)

b  # [1, 2, 3]

a = { 'name': 'Diana' }
b = a  # [1, 2, 3] will be
deleted
```

Python is dynamic

~~**Statically typed**:~~ when a name is **introduced**, its type is specified and cannot be changed

Dynamically typed: a name can refer to any object

Dynamic: an object is instantiated with a certain type, but its properties can be added / changed / removed at runtime

```
a = 'hello'
a = 5.1

a = SomeClass()
a.my_property = 'hello'
a.bit_length = 3
delattr(a, 'my_property')
```

Fundamental types

Types do exist in Python, even if they are not visible at a first glance

Selected **native types**:

- **NoneType**: None
- **Boolean**: True / False
- **Integer**: 1, 2, … (with sign, **never overflow**)
- **Float:** 1.0, 2.5, … (accurate up to 15 positions, usually C++'s "double")
- **Fraction**: ½, Complex, Decimal, …
- **String**

Composed types:

- **Tuple**
- **List**
- **Dictionary**

Absence of a value: None

`None` can be used to indicate the absence of a value

```
def naive_sqrt(n):
    if n < 0:
        return None
    else:
        return sqrt(n)

s = naive_sqrt(9)
if s is not None:
    print(s)
```

`None` is actually not a type but **a single static object**
So it is usually checked against with **`is not None`**

```
a = [1, 2]
b = [1, 2]

a == b   # True
a is b   # False
```

Numeric operations:

Usual operators: `+, -, *, /  #  / from hell in Python 2`

Unusual operators:

- Power `**`:

```
3**2   # 9
```

- Division rest `%`:

```
11 % 2   # 1
```

- Square root `math.sqrt` :

```
from math import sqrt
sqrt(9)   # 3
```

The **`math`** package includes anything you should need (pi, sin, cos, …)

Strings

Sequences of characters, can be delimited by ' or "

```
greeting = 'hello ' + "world"
```

The best way to print an object as a string can be inferred using "format":

```
print('file: {}/{}.{}'.format(dir, filename, extension))
# There are better ways to build a file path
```

```
message = 'the square root of {0} is {1:0.2f}; yes, of
{0}'.format(3, sqrt(3))'
```

```
print('Na'*6 + ' Batman!')  # NaNaNaNaNaNa Batman!
```

Tuples

Immutable sequences of values of any type

Tuples are constructed by enclosing variables within `()` (can also be omitted)

Support pattern matching and splicing:

```
x, y = y, x  # swap x and y

x, (y, z, w) = 3, (4, 5, (6, 7, 8))
               # w is assigned the tuple (6, 7, 8)
```

```
func_args = 1, 'hello', (), tuple()

a, b, c = fun_4_args(*func_args)
              #  returns a tuple, e.g.: return 1, 2, 3
```

Lists

Mutable sequences of values; created with [ ]

```
my_list = [1, 'hello', from_the, OtherSide()]
len(my_list)  # 4
'hello' in my_list  # True
```

Support many operations: appending, inserting, extending, removing, sorting, finding, filtering, …

```
interesting_list = [1, 5, 3] + [2, 6]
naive_backup_list = interesting_list
backup_list = interesting_list.copy()
interesting_list.sort()
print(naive_backup_list)  # [1, 2, 3, 5, 6]
print(backup_list)  # [1, 5, 3, 2, 6]
```

Slicing I

A portion of a tuple/list can be accessed through a **slice**

Indices **start at zero**;

the **left** index is **inclusive** and the **right** is **exclusive**

```
my_list = [1, 2, 3, 4, 5, 6, 7]
my_list[0:3]  # [1, 2, 3]
split_index = 4
my_list[:split_index] + my_list[split_index:] == my_list #True
```

Slicing II

A slice is not a copy of the original data, so it is fast and allows to change a list

```
my_list[3:5] = 'a', 3.5
        # my_list == [1, 2, 3, 'a', 3.5, 6, 7]
```

Supports fancy indexing

```
my_list[0:5:2]  # [1, 3, 5]
my_list[0:5:2]  = [-1]*3
       # my_list == [-1, 2, -1, 4, -1, 6, 7]
my_list[::-1]  # [7, 6, 5, 4, 3, 2, 1]
```

Dictionaries

Comparable to maps in other languages: hold a value for a given key (or map keys to values)

```
world_cups = { 'Brazil': 5, 'Italy': 4, 'Germany': 4 }
world_cups['Antarctica']  # KeyError
world_cups['France'] = 2
```

Keys and values can be of almost any type and inhomogeneous

```
weird_dict = {}
weird_dict[None] = 'foo' # None ok
weird_dict[(1,3)] = 'baz' # tuple ok
import sys
weird_dict[sys] = 'bar' # wow, even a module is ok !
weird_dict[(1,[3])] = 'qux' # oops, lists not allowed
crazy_dict = { 1: 'a', (3, 4, 5): {'hello': 5.4}}
```

Almost any object in Python is actually a dictionary

Program Structure

The minimal Python Program

```
#!/usr/bin/env python3
print('Hello World!')
```

The first line is not required; it makes sure that on Unix systems the program is executed with the correct interpreter

The program can be run with:

```
python program.py
```

Or on Unix systems also just with:

```
path/to/program.py
```

Given that the file has execution privileges; if not, the following is required:

```
chmod +x path/to/program.py
```

Reading parameters from the shell

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

program_name, person = sys.argv[0:2]

print('hello {} from {}!'.format(person, program_name))
```

`import` is used to use a module with external code; the specified module name can be used after the import statement (more on that later)

`sys.argv` is a list containing the arguments passed to the program from the shell

As in C/C++, the first argument is the name of the program

Defining functions

```
def boring_fun():
    print('hello world!')
    return 4

def maybe_boring_fun(obj='world'):
    print('hello {}!'.format(obj))

ret = maybe_boring_fun() # ret == None; prints "hello world!"
maybe_boring_fun('Python')  # prints "hello Python!"
```

A function always returns a value
if there is no **`return`** statement, it returns **`None`**

Arguments can have a default value, which is used if no value is specified for it

The body is delimited by **indentation** alone (must remain consistent!)

Specifying only certain parameters

```
def initially_boring_fun(obj='world', greeting='hello'):
    print('{} {}!'.format(greeting, obj))

initially_boring_fun(greeting='hola')  # prints "hola world!"
```

Python supports **named parameters**: arguments with a default value can be "skipped" by specifying which arguments to pass with their name

However, arguments without a default value ("positional arguments") cannot be left out

```
def tricky_fun(obj, greeting='hello'):
    print('{} {}!'.format(greeting, obj))

tricky_fun(greeting='hola')  # results in a TypeError
```

Control flow: if

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

rand_num = random.randint(0, 10)

user_guess = input('Guess a number (0 – 10): ')  # returns a string!

if int(user_guess) == rand_num:
    print('you won!')
else:
    print('you lost! the number was: {}'.format(rand_num))
```

The **"else"** clause can be omitted

As for functions, the clauses are delimited by **indentation only**

Control flow: while

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

rand_num = random.randint(0, 10)

user_guess = None # names must be introduced before they are used

while user_guess != rand_num:

    user_guess = int(input('Guess a number between zero and ten: '))

    if user_guess == rand_num:
        break

    print('try again!')

print('you won!')
```

Control flow: for

More advanced and versatile than in most languages

```
my_list = [1, 2, 3, 4, 5]
for item in my_list:
    print(item)

for index, item in enumerate(my_list):
    print('item number {}: {}'.format(index, item))

for i in range(2):
    print('Jump')
```

`for-in` is actually faster than accessing a list through `[ ]`, so prefer it!

Blocks and namespaces

Function bodies and clauses (**blocks**) are delimited **only by indentation**, which should be consistent (same number of spaces or tabs, not mixed)

```
a = 5

def useless_fun():
    a += 1
    b = 3

useless_fun()  #Error
```

```
a = 5

def useless_fun():
    global a
    a += 1
    b = 3

useless_fun()

print(a)  # 6
print(b)  # 3
```

```
a = 5

if True:
    a += 1
    b = 3

print(a)  # 6
print(b)  # 3
```

Differently than in other languages, **blocks not always introduce a new namespace/scope**; only functions (and classes) do

Passing by value or by reference?

In python a third option is used: **pass-by-object reference**:

```
b = 5

def innocuous_fun(a):
    a += 1
    print(a)

innocuous_fun(b)  # 6
print(b)  # 5
```

```
b = [5]

def innocent_fun(a):
    a.append(1)
    print(a)

innocent_fun(b)  # [5, 1]
print(b)  # [5, 1]  <- UPS!!
```

When mutable objects are passed and changed in the function, there are side-effects (the change is also visible from outside the function)

List comprehensions

A list can be derived from an existing one making use of a **list comprehension**:

```
import os
dir_path = os.path.expanduser('patterns/')
print(os.listdir(dir_path))  # 1.jpg, 2.jpg, ...

image_numbers = range(2, 22)

image_strings = ['{0:01d}'.format(i) for i in image_numbers]

filenames = [s+'.jpg' for s in image_strings]

image_filepaths = [os.path.join(dir_path, f) for f in filenames]
```

List comprehensions

What if we want to exclude a couple of images?

```
image_nums = range(2, 22)
excl_num = [4, 19]
image_str = ['{0:05d}'.format(i) for i in image_nums if i not in
excl_num]
```

Flow control: error handling

Some resources (files, network sockets…) should be closed properly after using them, even if an exception arises

Other languages make use of strategies like try/catch or RAII, which are invasive and/or verbose

```
try:
    f = open('some_file.txt')
    some_line = f.readline()
    raise ValueError('because I hate you :)')
    other_line = f.readline()  # this statement is never executed

except ValueError as e:
    print(e)  # because I hate you :)
    f.close()
```

Flow control: with

Python offers a lighter option: **context managers**

```
with open('some_file.txt') as f:
    some_line = f.readline()
    raise ValueError('because I hate you :)')
    other_line = f.readline()  # statement not executed

# f is properly closed even if the exception fires
```

You may want to create your own

Reading and writing files

Reading content of `in_file.txt` and writing it into `out_file.txt` after converting it to UPPERCASE:

```
with open('in_file.txt') as in_f,
open('out_file.txt', 'w') as out_f:

    for line in in_f.readlines():
        out_f.write(line.upper())
```

The files are automatically closed after exiting the `with` clause

Splitting code in more files

It is always good practice to divide big amounts of code into smaller files

A file containing Python code is called a **module**

Its contents can be accessed after **importing** its name into the current namespace

```
# my_library.py

def greet():
    print('hello world!')
```

```
# program1.py

import my_library

my_library.greet()
```

```
# program2.py

from my_library import greet

greet()
```

Finding modules

When a module is imported, a file with the corresponding name is looked for:

- If in the global namespace:
  in the current working directory of the interpreter
- If in a module being imported:
  in the directory of the module being imported
- In directories specified by the user in configuration files (PYTHONPATH)
- In the current Python interpreter's system library directory

Typically you will place your files somewhere in your home directory

If you cannot find your own modules, make sure that:

- If you are working from the command line:
  - you started the interpreter in the same directory as the modules
  - OR you added the directory where the modules are to your PYTHONPATH
- If you are running a program:
  - look up the (complicated) rules and double check the import paths

Importing a module

When a module is imported, its content is executed

```python
# my_program.py

import greetings as g  # hello everyone!

g.greet('world')  # hello world!
```

```python
# greetings.py

def greet(target):
    print('hello {}!'.format(target))

greet('everyone')
```

This can be useful, but it is typically undesired

It is possible to execute instructions only when the file is used as a program

```python
# my_program.py

import greetings as g

g.greet('world')  # hello world!
```

```python
# greetings.py

def greet(target):
    print('hello {}!'.format(target))

if __name__ == '__main__':
    greet('everyone')
```

What about classes?

Usually you do not need them in Python

Other languages make it necessary to write a struct/class just to pass data around; you can do that with tuples, lists or dictionaries

In particular, try to avoid this:

```
some_filter = SomeFilter(my_image)
filtered_image = some_filter.filter(some_param=5.0)
```

This is actually a disguised function:

```
filtered_image = some_filter(image=my_image, some_param=5.0)
```

Functions are easier to read and to test

But I really need a class!

This is true when:

- some data logically belong together and represent a “state”
- some operations manipulate them at the same time to pass from a state to another

In that case it makes sense to declare a class:

```
class VendingMachine:
    CHOCOLATE_BAR_PRICE = 1.30  #Class variable (~“static”)
    HOTDOG_PRICE = 3.00

    # __init__ is the (unique) constructor
    def __init__(self):
        self.chocolate_bars = 0 # Instance variable
        self.hotdogs = 0
        self.money = 0
        self.inserted_money = 0
```

Class and instance variables

```
class VendingMachine:
    CHOCOLATE_BAR_PRICE = 1.30  # Class variable (~"static")
    HOTDOG_PRICE = 3.00
    def __init__(self, chocolate_bars=0, hotdogs=0):
        self.chocolate_bars = chocolate_bars
        self.hotdogs = hotdogs
        self.money = 0
        self.inserted_money = 0
```

Since the price is the same for all vending machines, it is a class variable

Instance variables (changing for each instance) should all be declared in the **`__init__`** method, which is called at object construction

Only one **`__init__`** method is allowed; not a problem thanks to named arguments

Class and instance variables

```
class VendingMachine:
    CHOCOLATE_BAR_PRICE = 1.30  # Class variable (~"static")
    HOTDOG_PRICE = 3.00
    def __init__(self, chocolate_bars=0, hotdogs=0):
        self.chocolate_bars = chocolate_bars
        self.hotdogs = hotdogs
        self.money = 0
        self.inserted_money = 0
```

```
empty_vm = VendingMachine()
hd_vm = VendingMachine(hotdogs=20)
```

Defining methods

Instance methods must take self as first parameter

```
class VendingMachine:
    (…)
    def refill(self):
        self.chocolate_bars = 10
        (...)

    def insert_money(self, ins_money):
        self.inserted_money += ins_money

    def buy_chocolate_bar(self):
         if self.inserted_money >= VendingMachine.CHOCOLATE_BAR_PRICE:
              self.inserted_money -= VendingMachine.CHOCOLATE_BAR_PRICE
              self.money += VendingMachine.CHOCOLATE_BAR_PRICE
              return True
          else:
               return False
    (…)
```

Time's up!

This was just a brief look at the language; there's much more to it:

- Class static methods
- Class inheritance
- super()
- Interfaces
- Decorators
- Generators
- Regular expressions
- Core libraries: itertools, os, sys, glob, pickle, multiprocessing, …

Continue exploring, and before doing something tricky and/or boring look for a solution online! Probably there is a simple and elegant solution ready.

Numeric computing: Numpy

How do I do what I was doing in MATLAB?

Scientific libraries for Python:

- Matrices, vectors, dot products, … : Numpy
- Symbolic calculus: Sympy
- Plots: matplotlib
- Reading images, filtering, ... : Scipy, OpenCV
- Manipulating and visualizing medical data: VTK, ITK, MITK, …

Machine Learning libraries (yay!):

- Scikit-learn: lots and lots of algorithms ready to be used out-of-the-box
- Pylearn2: same, but allows more control
- Pandas: utilities for data analysis (parsing files, normalizing data, ...)
- Caffe: deep learning framework for vision applications
- TensorFlow: Google's library for data flow graphs
- Pythorch

Last but not least: most of them get along well!

Numpy Arrays

Similar to MATLAB's matrices

Homogeneous multidimensional array of values

Support same slicing techniques as lists (and most other operations)

The number of dimensions (**axes**) is called **rank**

A comprehensive introduction can be found https://docs.scipy.org/doc/numpy-dev/user/quickstart.html

Broadcasting

Many operations require the input arrays to have the same **shape** (length in each dimension), for example addition

This can quickly become a nightmare (if you know MATLAB's `repmat` you know about it)

Numpy makes it easier to avoid this problem by "stretching" an array along the missing dimensions, provided that the ones with value > 1 coincide

You can think of it as an automatic `repmat`

The precise rules can be found in: https://docs.scipy.org/doc/numpy-dev/user/basics.broadcasting.html

Matlab<=>Python

For common equivalences check Numpy for Matlab Users:

`a.'`	`a.transpose()` or `a.T`	transpose of `a`
`a'`	`a.conj().transpose()` or `a.conj().T`	conjugate transpose of `a`
`a * b`	`a.dot(b)`	matrix multiply
`a .* b`	`a * b`	element-wise multiply
`a./b`	`a/b`	element-wise divide
`a.^3`	`a**3`	element-wise exponentiation

https://docs.scipy.org/doc/numpy/user/numpy-for-matlab-users.html

Your turn to try!!!