- Materiale didattico
  - slides
  - esercizi risolti

- Esame: Esercizi di programmazione da svolgere al computer
  - 1 ora
  - Esercizi di programmazione in python/matlab. Tipicamente 3 esercizi python e 3 esercizi matlab

- DATE esami
  - 12/6 ore 9
  - 27/6 ore 14
  - 22/7 ore 14
  - 9/9 ore 14

- [ITA] Introduzione a python, Tony Gaddis
- Think Python, by Allen B. Downey

- The Coder's Apprentice Learning Programming with Python 3, by Pieter Spronck [Free PDF]

- A Whirlwind Tour of Python, by Jake VanderPlas
- Python Data Science Handbook, by Jake VanderPlas
- Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, by Aurélien Géron

Python Basics

- variables
- elementary data types
- expressions
- data structures
- conditional statements
- cycles
- functions
- files
- exceptions
- classes

python

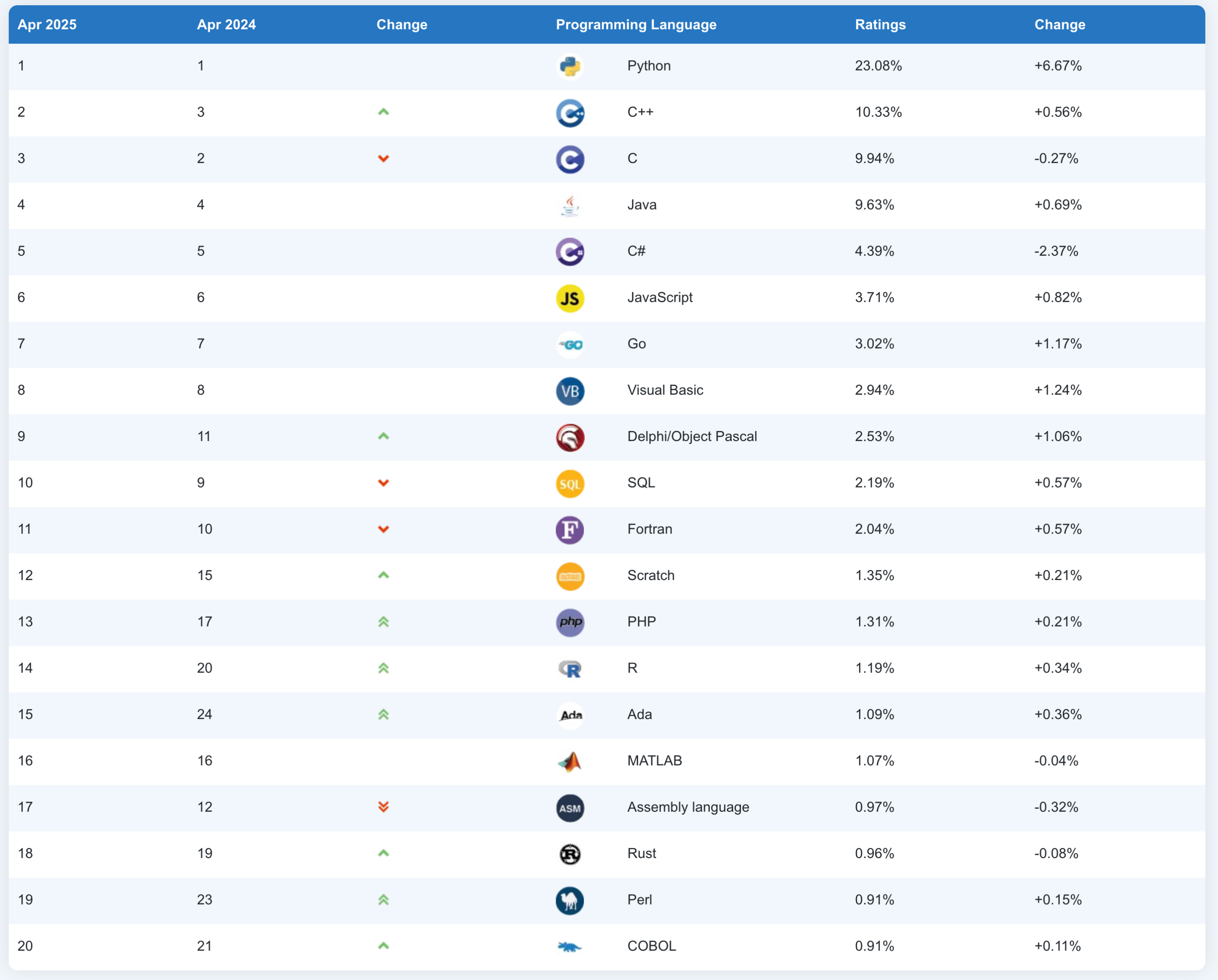
- First public release in 1991 by Guido Van Rossum (the Benevolent Dictator for Life)
- Python2 released in 2000, and python3 in 2008
- Interpreted
- Dynamically typed (you don't need to declare variables)
- Object oriented
- Low number of native symbols

- Widely used for data analyses, scientific computing, and Artificial Intelligence (many libraries)

- Not particularly efficient (compared to C++/Fortran)
- But good performance for data analysis can be achieved using libraries

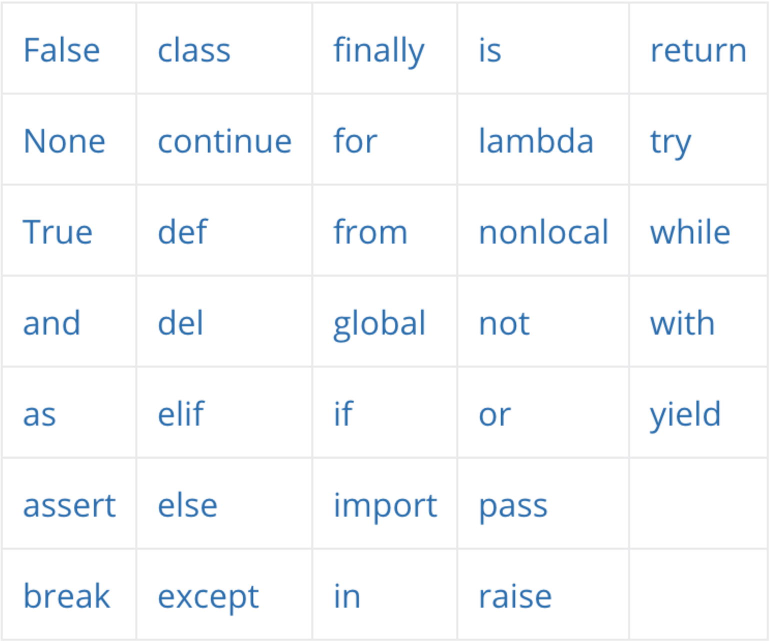
Why python ?

https://www.tiobe.com/tiobe-index/

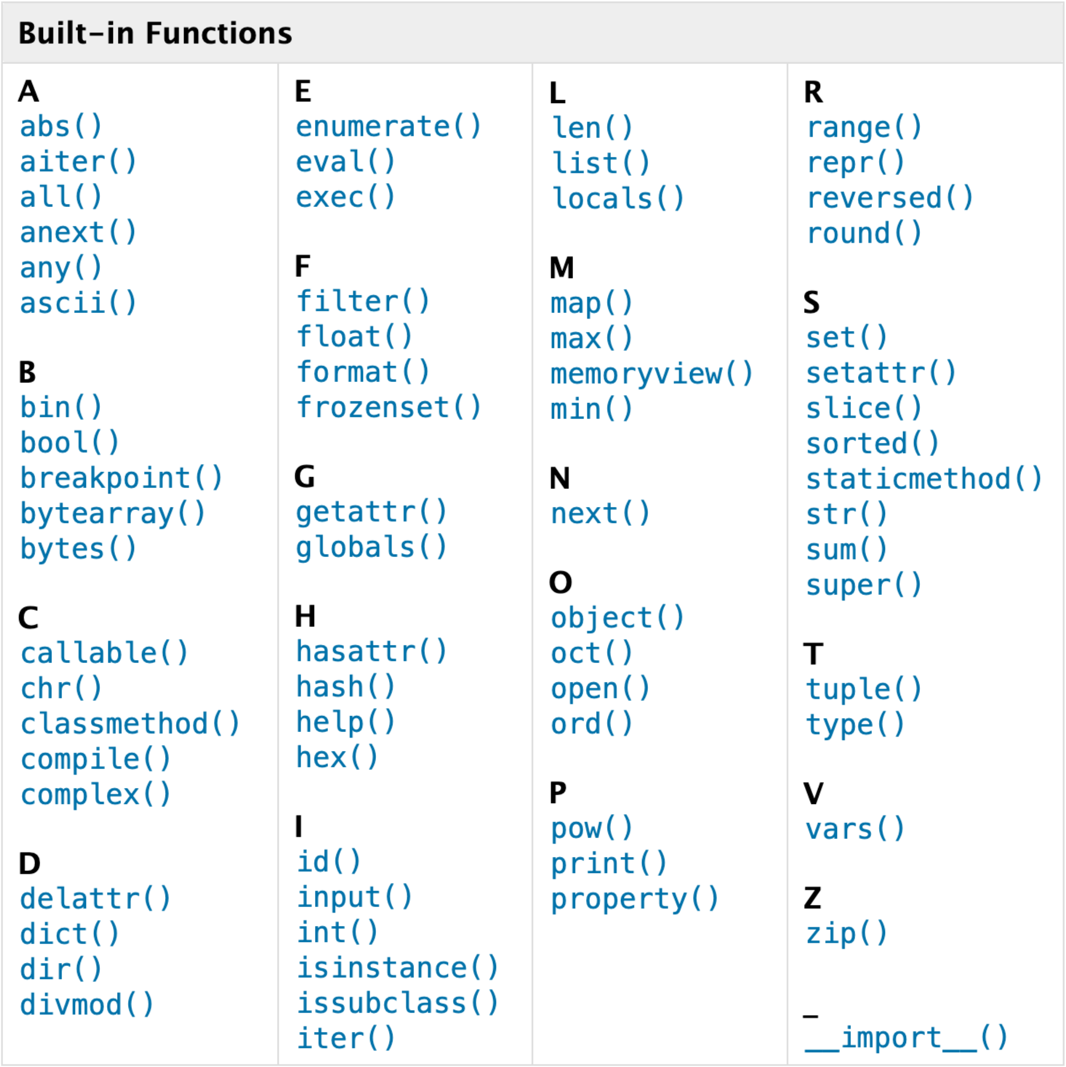


Apr 2025	Apr 2024	Change	Programming Language	Ratings	Change
1	1		Python	23.08%	+6.67%
2	3	↑	C++	10.33%	+0.56%
3	2	↓	C	9.94%	-0.27%
4	4		Java	9.63%	+0.69%
5	5		C#	4.39%	-2.37%
6	6		JavaScript	3.71%	+0.82%
7	7		Go	3.02%	+1.17%
8	8		Visual Basic	2.94%	+1.24%
9	11	↑	Delphi/Object Pascal	2.53%	+1.06%
10	9	↓	SQL	2.19%	+0.57%
11	10	↓	Fortran	2.04%	+0.57%
12	15	↑	Scratch	1.35%	+0.21%
13	17	⇈	PHP	1.31%	+0.21%
14	20	⇈	R	1.19%	+0.34%
15	24	⇈	Ada	1.09%	+0.36%
16	16		MATLAB	1.07%	-0.04%
17	12	⇊	Assembly language	0.97%	-0.32%
18	19	↑	Rust	0.96%	-0.08%
19	23	⇈	Perl	0.91%	+0.15%
20	21	↑	COBOL	0.91%	+0.11%

Keywords in Python programming language



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



Built-in Functions

A
abs()
aiter()
all()
anext()
any()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()
classmethod()
compile()
complex()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()

Where to get python from

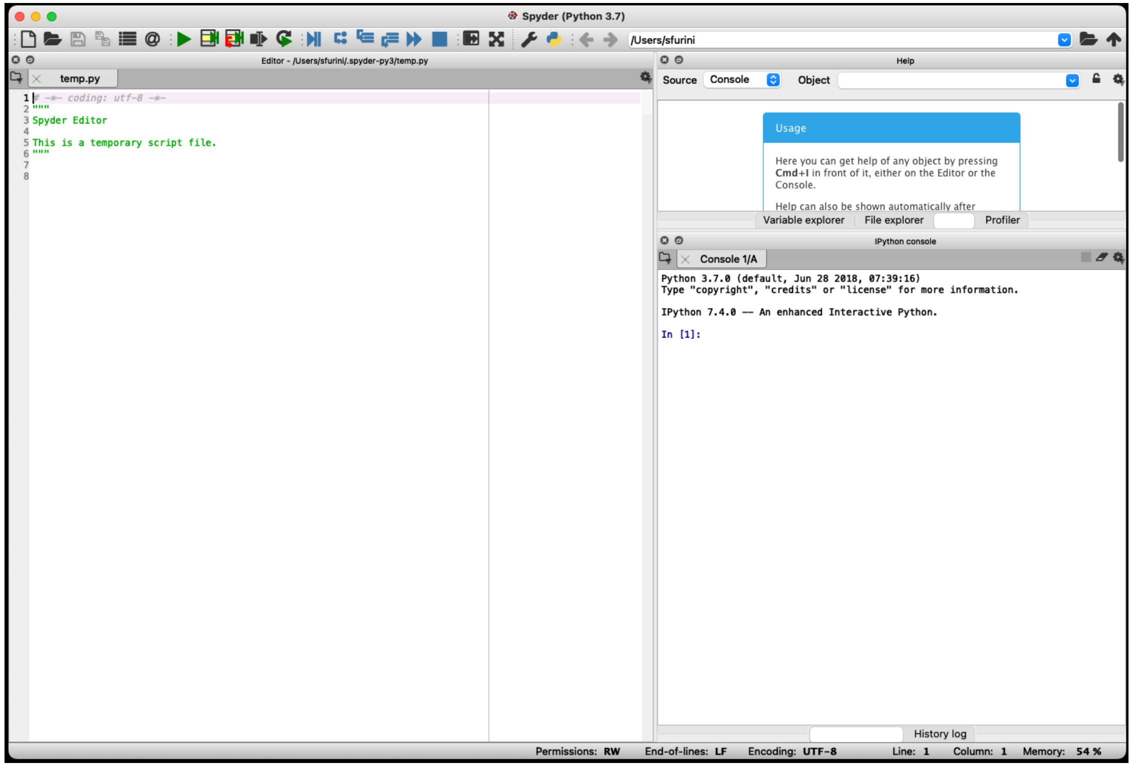
Freely available (for any platform) at http://www.python.org/

Suggestion: Download python within the anaconda platform https://www.anaconda.com/distribution/

- – package manager (conda)
- – text editor (spider)
- – iterative interpret (ipython)

How to execute python code

- Using a GUI (like spider)



```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""
```

Usage

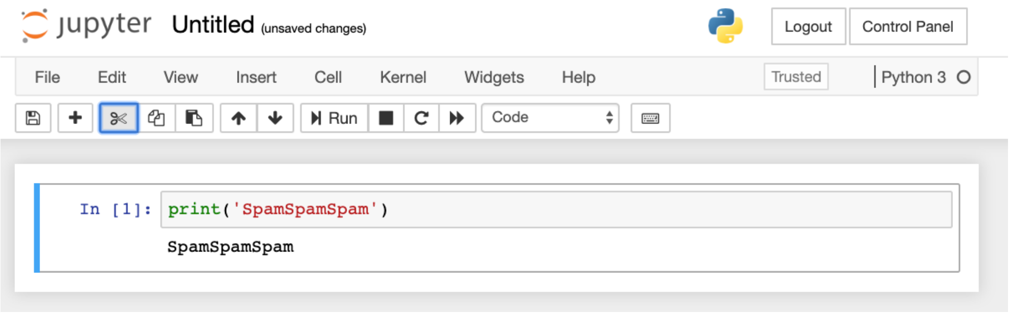
Here you can get help of any object by pressing **Cmd+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after

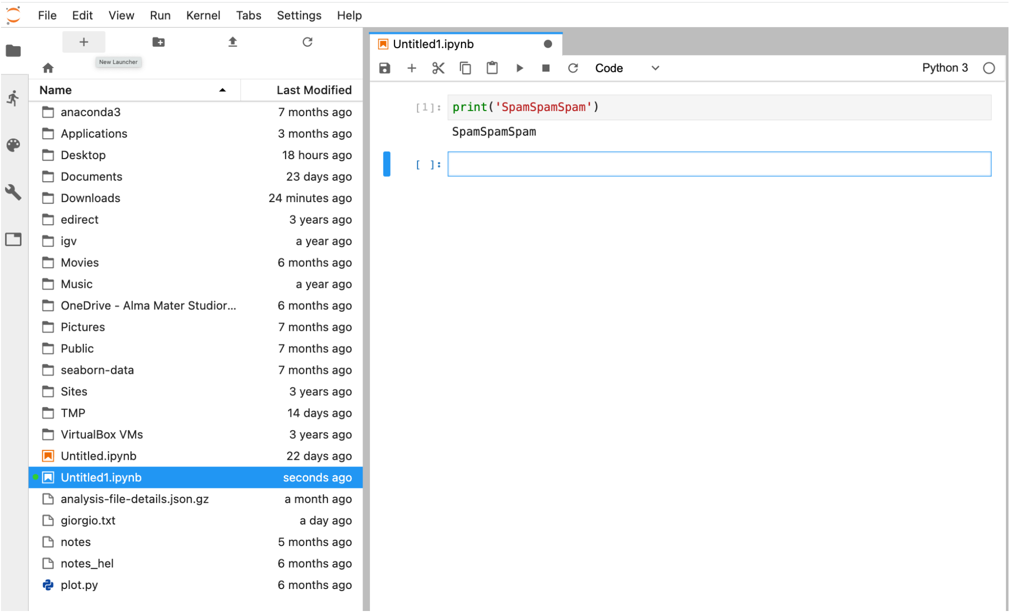
```
Python 3.7.0 (default, Jun 28 2018, 07:39:16)
Type "copyright", "credits" or "license" for more information.

IPython 7.4.0 -- An enhanced Interactive Python.

In [1]:
```

How to execute python code

- Using *jupyter notebook / jupyter lab*



```
In [1]: print('SpamSpamSpam')
        SpamSpamSpam
```



Name	Last Modified
anaconda3	7 months ago
Applications	3 months ago
Desktop	18 hours ago
Documents	23 days ago
Downloads	24 minutes ago
edirect	3 years ago
igv	a year ago
Movies	6 months ago
Music	a year ago
OneDrive - Alma Mater Studior...	6 months ago
Pictures	7 months ago
Public	7 months ago
seaborn-data	7 months ago
Sites	3 years ago
TMP	14 days ago
VirtualBox VMs	3 years ago
Untitled.ipynb	22 days ago
Untitled1.ipynb	seconds ago
analysis-file-details.json.gz	a month ago
giorgio.txt	a day ago
notes	5 months ago
notes_hel	6 months ago
plot.py	6 months ago

```
[1]: print('SpamSpamSpam')
     SpamSpamSpam

[ ]:
```

Comments

- Everything after the ‘#’ character is treated as a comment, and it is not executed
- Use comments to document what you’re doing

```
# this is not executed
print(“SpamSpamSpam”) # … this is also not executed
```

One instruction = One line

- But long instructions can be broken over multiple lines by:
  - using the \ character
  - just going to the next line if the text is included in between parenthesis
- But it is possible to write more instructions on a line, if separated by ;

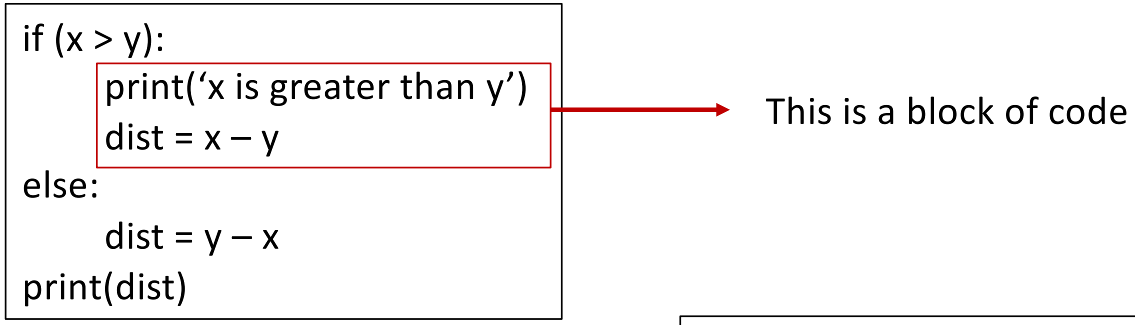
```
x = 3 * 2 \
(5 + 4)
```

```
x = 3 * (2 +
6)
```

```
x = 0; y = 0
```

BLANK SPACES

- Indentation is used to define blocks of code
- The number of blank spaces at the beginning of the line is arbitrary (multiple of 4 is the standard) but it needs to be consistent within the block
- Spaces within a line are meaningless



```
if (x > y):
        print('x is greater than y')
        dist = x – y
else:
        dist = y – x
print(dist)
```

This is a block of code

C++

```
if (x > y) {
        cout << 'x is greater than y' << endl;
        dist = x – y;
} else {
        dist = y – x;
}
cout << dist << endl;
```

```
print('SpamSpamSpam')
print    (  'SpamSpamSpam'    )
```

These two lines are identical

VARIABILE_NAME = EXPRESSION

Binding: a logical name is associated with an object (the result of the expression)

```
movie = “Life of Brain”
print(movie) # it’s the same as print(“Life of Brain”)
```

```
x = 1
print(x)
```

- The logical name is similar to a pointer to an object
  - Different names can point to the same object
  - The type is associated with the object not with the name (there’s an overhead compared to statically typed languages as C)

Python names

- Sequences of arbitrary length of alphanumeric characters and underscores
  - The first character cannot be a number
  - Names that start with underscore have special meanings
  - Do not use spaces inside names
  - Names are case sensitive

Elementary data types (they are objects)

Data type	Values	Examples
bool	True, False	
int	Integer numbers (no limits in length) It is possible to use bases different from 10 with the prefixes: 0b, 0B (binary); 0o, 0O (octal); 0x, 0X (hexadecimal)	1234 **0b**10 **0x**1a
float	floating numbers lowest absolute value: 5e-324 highest absolute value: 1e308	0.01 .01 1e-2 1E-2
complex	complex numbers	2.1+3.4j complex(2.3, 3.4)
str	string of characters Both single or double apexes can be used Use three apexes for strings spanning more than one line	“A” ‘ATCG’ “””ATC CFT”””
NoneType	None	

Basic data types are immutable

Data type

- The type of the variable is not explicitly defined
- It is associated with the object itself and not with its name
- There's an overhead compared to programming languages like C
  - E.g.: an int is actually an object, that stores the actual integer number plus other information (the data type)

C

```
int a;
float b;
bool c;
a = 1;
b = 1.0;
c = True;
```

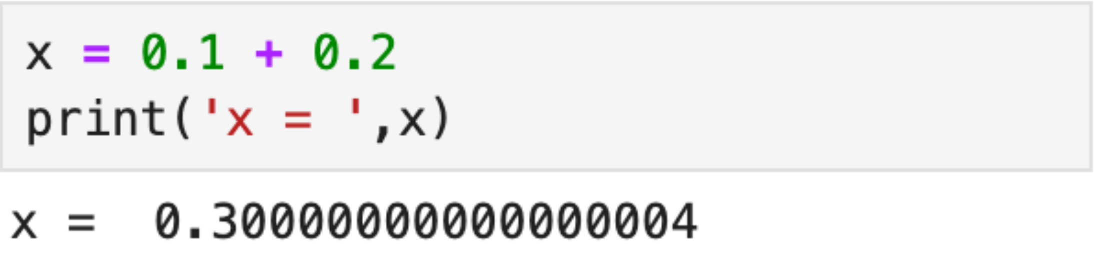
python

```
a = 1 # int
b = 1.0 # float
c = True # bool
```

- Variables still have a type, it's just not explicitly defined by the programmer

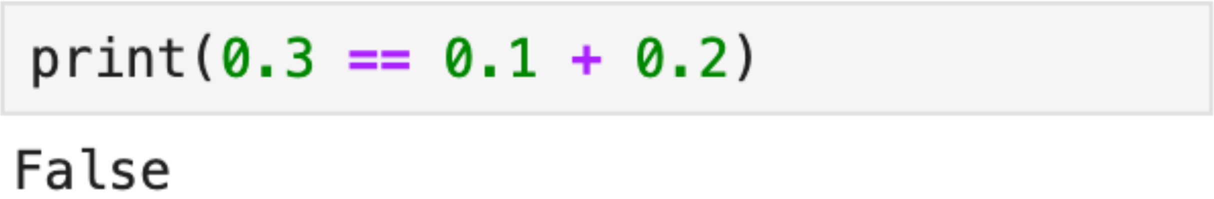
Beware of rounding

Floating points are stored as binary numbers → rounding are possible
It is usually a bad idea to test if two floating numbers are the same



```
x = 0.1 + 0.2
print('x = ',x)
```

```
x =  0.30000000000000004
```



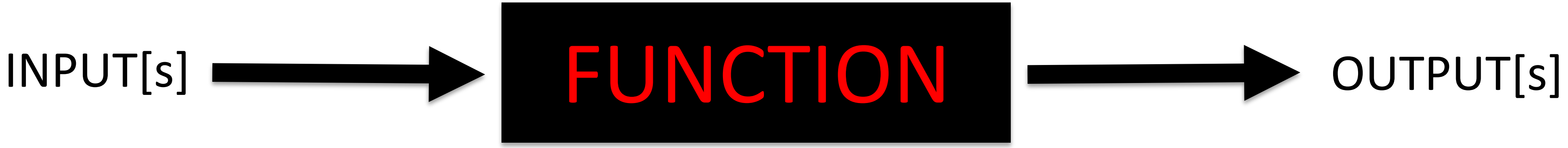
```
print(0.3 == 0.1 + 0.2)
```

```
False
```

Functions

- Block of codes that perform a specific operation
- They can receive inputs
- They can return an output
- Some input parameters might have default values (keyword parameters)

- When using a function you don't need to know anything about how it works



```
return_value = function_name(parg1, parg2, ….,
    karg1 = val1, karg2 = val2, ….)
```

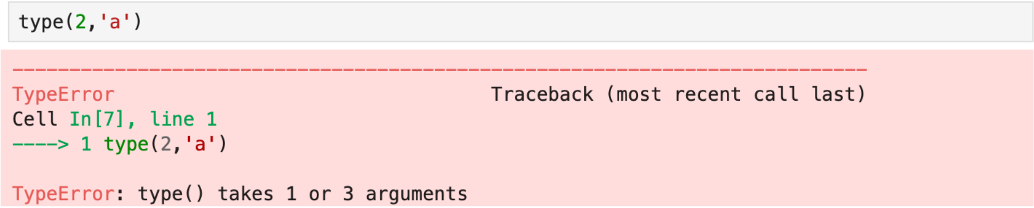
- everything in blue is not compulsory

- positional arguments:
  - the position is important
  - some positional argument might be compulsory
  - an arbitrary number of positional arguments might follow

- keyword arguments:
  - the position is not important
  - if a keyword argument is not defined, the default value is used
  - there can be an arbitrary number of keyword arguments

type(object): return the type of an object

type(True) → bool
type(1) → int
type(1.0) → float

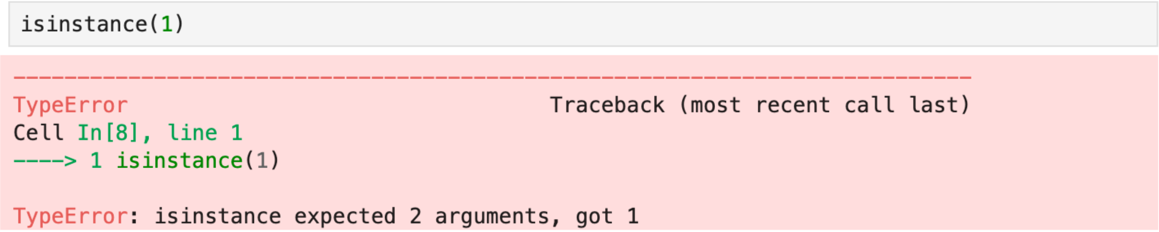


```
type(2,'a')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[7], line 1
----> 1 type(2,'a')

TypeError: type() takes 1 or 3 arguments
```

isinstance(object, type): return True if the object is of that type

isinstance(False, bool) → True
isinstance(1.0, int) → False

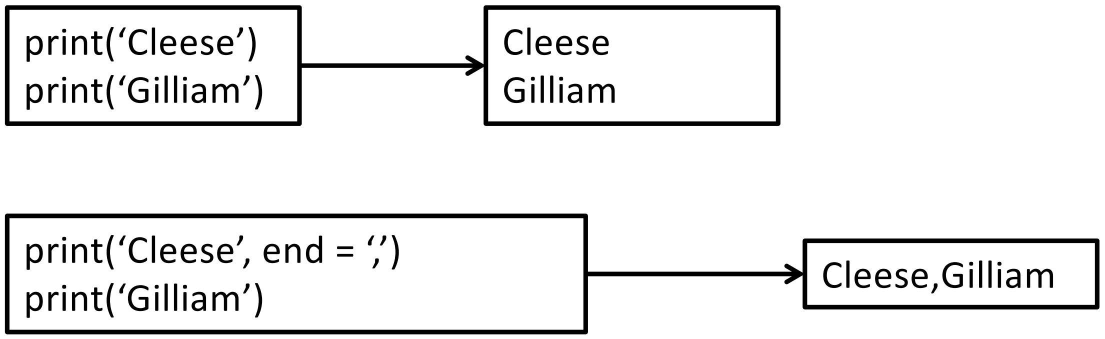


```
isinstance(1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[8], line 1
----> 1 isinstance(1)

TypeError: isinstance expected 2 arguments, got 1
```

print()

- It shows in the standard output the input argument(s)
- It can take an arbitrary number of arguments
  - print(‘John’)→ John
  - print(‘John’, ‘Cleese’) → John Cleese
- At default, the input parameters are printed separated by a space, but it is possible to change this behaviour with the argument *sep*
  - print(‘John’, ‘Cleese’, sep = ‘-’) → John-Cleese
- At default, the last printed character is the newline, but it is possible to change it with the argument *end*
- With *flush = True*, immediate writing to stdout is forced



```
print(‘Cleese’)
print(‘Gilliam’)
```

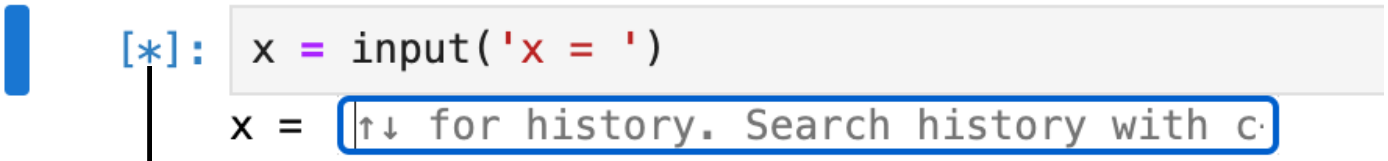
```
Cleese
Gilliam
```

```
print(‘Cleese’, end = ‘,’)
print(‘Gilliam’)
```

```
Cleese,Gilliam
```

input()

- It shows in the standard output the input argument (if any)
- Then it waits for input from the standard input up to a newline character
- It returns the input from standard input as a str



```
[*]:  x = input('x = ')
      x = ↑↓ for history. Search history with c
```

it means that the cell is being executed. In this case the execution ends when typing the new line character

help()

- For any object it returns the help string

help(print)

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Try:

```
x = 1
help(x)
```

help(object)	Return the help string for the object	
abs(number)	Return the absolute value	abs(-3.2) → 3.2
pow(base, exp)	Return base^exp	pow(2,3) → 8
round(number)	Round the number to the closest integer	round(1.9) → 2
bin(int), hex(int), oct(int)	Convert an integer number into a string containing its binary/hexadecimal/octal representation	bin(4) →'0b100’
bool(object), int(object), float(object), str(object)	Convert the object into a bool/int/float/str (when possible)	
complex(real [, imag])	Return a complex number with real and imag part	complex(1,2) → 1+2j
chr(int)	Return the unicode character with that index	chr(8364) → € chr(0x1F92F) →🤯
ord(chr)	Return the unicode index of the character	ord(‘€’) → 8364
isinstance(object, type)	Return True is the object is of class type	isinstance(1,int) → True
type(object)	Return the type of the object	type(1) → int

bool(), int(), float(), complex(), str()

- Convert input to an object of the corresponding type

bool(1) → True
bool(0) → False
bool(‘any not empty string’) → True
bool(‘’) → False

complex(1) → 1+0j
complex(1,2) → 1+2j
complex(‘1+2j’) → 1+2j

int(‘12’) → 12
int(‘12.1’) → Error

These are not functions, they are classes

Python Expressions

- set of commands that when evaluated by the interpret return an object
- Examples: arithmetic expressions

(3 * 2) + 4

4 – 7*5

Since expressions are evaluated by the interpret, they can be used on the right side of binding operations

x = 1 + 4

x = 1 + y

x = 1 + x

Arithmetic operators

Operator		Examples
+	Sum	1+7 = 8 True + 1 + False = 2 *# conversion rules are automatically applied*
-	Subtraction	
*	Product	
/	Division	When mixing int with float, everything is converted to float (in python3, it's the opposite in python2)
//	Quotient	5 // 2 = 2 1 // 0.3 = 3
%	Module	5 % 2 = 1 1 % 0.3 = 0.1
**	Power	3**2 = 9

Operators can be bool, int, float, complex
In numeric expressions, True is treated as 1, False as 0
The results always belongs to the wider set (bool < int < float < complex)

Combining binding with operators

All the arithmetic operators can be combined with the binding operator into a single expression

```
x = x + 1  → x += 1
x = x – 2  → x -= 2
x = x * 3  → x *= 3
x = x / 4  → x /= 4
x = x // 5 → x //= 5
x = x % 7  → x %= 7
x = x ** 4 → x **= 4
```

Elementary types are immutable, so x += 1 defines a new object

+/* Operators with strings

“A” + “C”	“AC”
“ATCG”*4 4*“ATCG”	“ATCGATCGATCGATCG”

Both can be combined with the binding operator

```
s = ‘Spam’
s += ‘Spam’ → s = ‘SpamSpam’
```

```
s = ‘Spam’
s *= 3 → s = ‘SpamSpamSpam’
```

in Operator

object1 **in** *object2*

True if *object1* is included in *object2*

It can be used any time *object2* is iterable

“TATA” in “ACGTACGC**TATA**CG”	True
“TATA” in “AT”	False
1 in 12	ERROR
‘1’ in ‘12’	True

not in Operator

Logical not of the in operator

“TATA” in “ACGTACGC**TATA**CG”	False
“TATA” in “AT”	True

Logical operator

	NOT
True	False
False	True

AND	True	False
True	True	False
False	False	False

OR	True	False
True	True	True
False	True	False

Rules for Boolean conversions

- int/float/complex are True if not zero
- strings are True if not empty
- None is considered False

Python logical operator

Operator		Example	Result
not	Return a boolean value	not False not 0 not 1 not “ATCG” not ””	True True False False True
and	Return: • the second operand when True • the False operand when only one is False • the first operand when both are False	True and False 1 and 0 1 and 3 0 and “”	False 0 3 0
or	Return: • the first operand when both are True • the True operand when only one is True • the second operand when both are False	True or False 1 or 0 0 or 1	True 1 1

Relational operators

Operator		Example	Result
==	Equal	1 == 0 “ATC” == “ATC”	False True
!=	Different	“ATGC” != “ATC” “A” != 1	True True
<		“AC” < “AT” “AT” < “AT”	True False
>			
<=		“AT” <= “AT”	True
>=			

Bitwise operators

Operator		Example	Result
&	and	2 & 6 2 = 0b10 6 = 0b110	2
|	or	2 | 6	6
^	xor	2 ^ 6	4
~	not		
<<	left shift	2<<1	4
>>	right shift	2>>1	1

They can be combined with the binding operator

Priority when evaluating expressions

Operator
()
function call
Slicing
accessing attributes
**
*, /,//, %
+, -
relational operators
in, not in
not
and
or

When in doubt, use parentheses !

IF (1/3)

```
…
if expression:
    <Block of code>
…
```

- Indentation is used to define the block of code that is executed only when the expression is True
- Be consistent in using tabs/spaces for indentation throughout the code

IF (2/3)

```
if expression1:
    <Block of code 1>
else:
     <Block of code 2>
```

IF (3/3)

```
if expression_1:
    <Block of code 2>
elif expression_2:
    <Block of code 2>
[…]
elif expression_N:
    <Block of code N>
else:
    <Block of code N + 1>
```

Only the block of code corresponding to the first True expression is executed

The final else might be missing

if on a single line

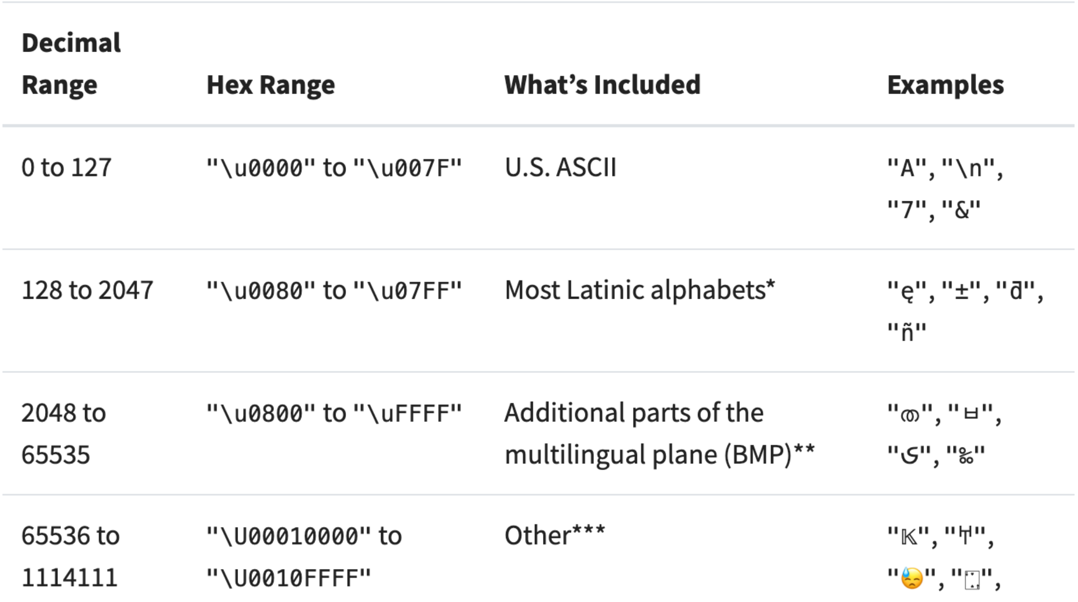
expression if condition else expression

```
x if x > 0 else -x
```

```
x-y if x > y else y-x
```

UNICODE

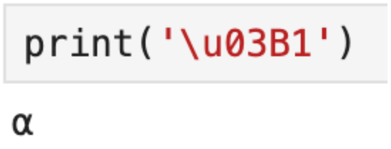
- Python uses the UNICODE standard for characters
- The default encoding is UTF-8 (ASCII is a subset of UNICODE)
- Variable number of bytes is used for coding (depending on the character)



Decimal Range	Hex Range	What's Included	Examples
0 to 127	"\u0000" to "\u007F"	U.S. ASCII	"A", "\n", "7", "&"
128 to 2047	"\u0080" to "\u07FF"	Most Latinic alphabets*	"ę", "±", "ƌ", "ñ"
2048 to 65535	"\u0800" to "\uFFFF"	Additional parts of the multilingual plane (BMP)**	"ത", "ㅂ", "ی", "‰"
65536 to 1114111	"\U00010000" to "\U0010FFFF"	Other***	"𝕂", "ᅱ", "😓", "🂡",

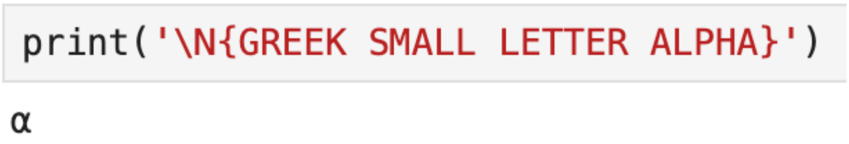
Special characters

\n = Newline
\t = tab
\xNN = hexadecimal ASCII code
\uNNNN = UTF-8 code
\N{name} = UTF-8 name



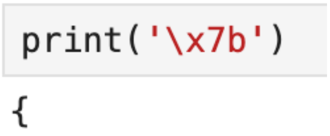
```
print('\u03B1')
```

```
α
```



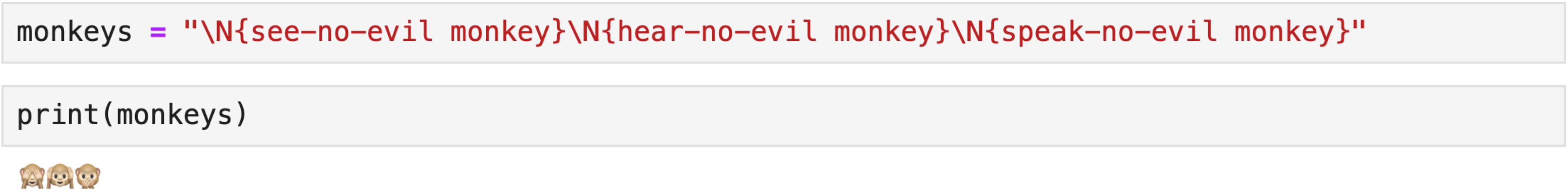
```
print('\N{GREEK SMALL LETTER ALPHA}')
```

```
α
```



```
print('\x7b')
```

```
{
```



```
monkeys = "\N{see-no-evil monkey}\N{hear-no-evil monkey}\N{speak-no-evil monkey}"
```

```
print(monkeys)
```

```
🙈🙉🙊
```

str Indexing

s = ‘abcdefgh’

a	b	c	d	e	f	g	h
0	1	2	3	4	5	6	7
-8	-7	-6	-5	-4	-3	-2	-1

len(s)	Return the number of elements in s
s[i]	i-th element
s[i:j]	from the i-th element to the one before the j-th element
s[i:]	from the i-th element to the last one
s[:j]	from the first element to the one before the j-th element
s[:]	all the elements in the str
s[i:j:k]	from the i-th element to the one before the j-th element, selecting elements with step k
s[-i]	i-th element reading from the end, the index of the last element is -1

LOOPS

```
…
while expression:
   <Block of code>
…
```

```
for variabile in iterable:
   <Block of code>
```

break: stop the execution of the loop

continue: move straight to the next execution of the loop

else: the following block of code is executed only if the loop terminates without a break

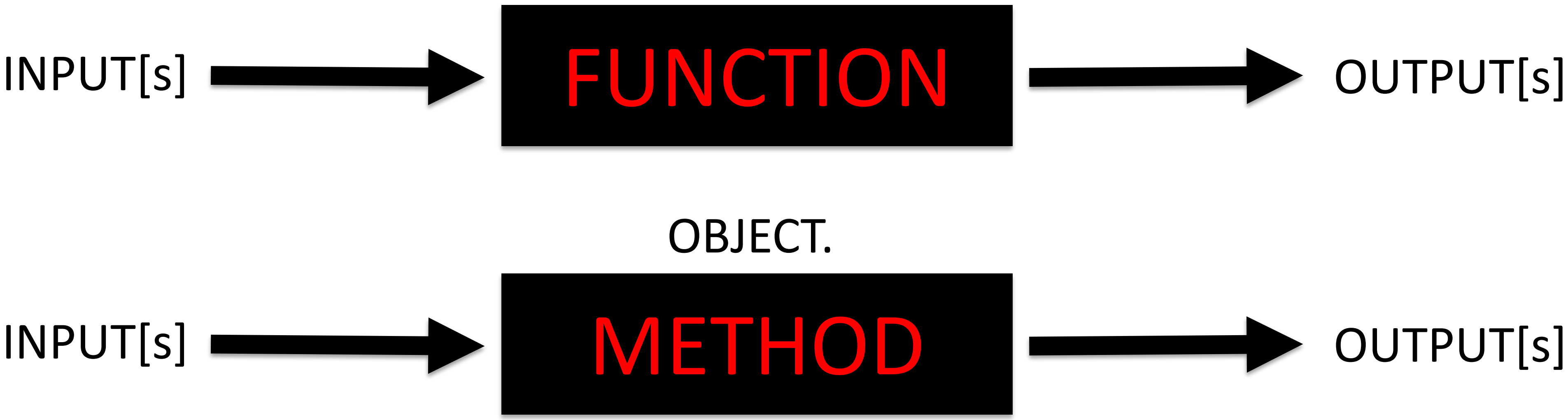
```
while True:
    x = input(‘Provide a positive number: ‘)
    x = int(x)
    if x > 0:
        print(‘x = ‘,x)
        break
```

```
import math
l = [1.1, 3.2, -0.1]
for x in l:
    if x < 0:
        continue
    print(‘log(x) = ‘,math.log(x))
```

```
i = 0
while i < 10:
    x = input(‘Provide a positive number: ‘)
    x = int(x)
    if x > 0:
        print(‘x = ‘,x)
        break
    i += 1
else:
    print(‘I surrender !’)
```

Methods

- Similar to functions but they operate on a specific object
- They can change the object itself
- Positional and keyword parameters work as in function



```
x = 1 + 2j
x.conjugate() → 1-2j
```

```
s = ‘a usEleSs sTRIng’
s.lower() → ‘a useless string’
S = s.upper() → ‘A USELESS STRING’
S.replace(’STRING’, ‘EXAMPLE’) → A USELESS EXAMPLE
```

s.lower()	Return a string with all the characters converted to lower case
	s1 = ‘Wanda’
	s2 = s1.lower() → s2 = ‘wanda’

s.upper()	As before, but converting to upper case

s.find(substring)	Search for substring into s. It returns the position of the first occurence (indexes start at zero). If substring is not included in s, it return -1.
	s1 = ‘SpamSpamSpam’
	i = s1.find(‘am’) → i = 2

s.strip(remove_chr)	Remove all the characters in remove_chr at the begin/end of s. At default it removes blank spaces and new lines.
	s1 = ‘ ATCG ‘
	s2 = s1.strip() → s2 = ‘ATCG’

s.replace(old,new)	Return a new string where any occurance of old is replaced by new
	dna = ‘ATCGTCG’
	rna = dna.replace(‘T’,’U’)

s.join(list)	Create a string by merging the elements in seq using s as a separator
s.split(sep)	Split a string into a list using sep as separator

	s = ‘deaD paRrot’
s.capitalize(), s.title()	Dead parrot, Dead Parrot

s.center(20)
s.ljust(20)
s.rjust()
	' deaD paRrot ‘
	'deaD paRrot ‘
	' deaD paRrot'

s.count(‘ea’)	2
s.endswith(‘ot’), s.startswith(‘de’)	True, True
s.find(‘a’), s.rfind(‘a’)	2, 6 (-1 if not found)
s.index(‘a’), s.rindex(‘a’)	2, 6 (Error if not found)
s.replace(‘ea’,’e’)	deD paRrot
s.isalpha()	True

s.isdecimal(), s.isdigit(), s.isalnum()
s.isnumeric(), s.isspace(), s.istitle()
	False

s.isupper(), s.islower()	False
s.upper(), s.lower(), s.swapcase()	DEAD PARROT, dead parrot, DEAd PArROT
s.strip(‘D’), s.rstrip(), s.lstrip()	eaD paRrot

Strings are immutable → the methods cannot change the string, they return a new one

format strings

It creates formatted strings by substituting the parts between curly brackets by the actual values provided to the method

```
‘This {} weights {} kg’.format(’dog’, 16)
→ This dog weights 16 kg
```

it is possible to enumerate the items

```
‘This {0} weights {1} kg’.format(’dog’, 16)
→ This dog weights 16 kg
```

```
‘This {1} weights {0} kg’.format(16, ’dog’)
→ This dog weights 16 kg
```

… or to give names to them

```
‘This {animal} weights {w_kg} kg’.format(animal = ’dog’,  w_kg = 16)
→ This dog weights 16 kg
```

… and to define the format style

```
‘This {animal:s} weights {w_kg:f} kg’.format(animal = ’dog’,  w_kg = 16)
→ This dog weights 16.0000 kg
```

```
‘This {animal:s} weights {w_kg:4.2f} kg’.format(animal = ’dog’,  w_kg = 16)
→ This dog       weitghs 16.00 kg
```

s	string
c	character
d	integer, base 10
f	floating
e	exponential notation
b	binary
o	octal
x	hexadecimal

<	left-justified
^	centered
>	right-justified

```
{:justificationfield_witdhfield_type}
```

```
s = '{:<10s}'.format('dog')
print(s)
s = ‘{:^10s}'.format('dog')
print(s)
s = ‘{:>10s}'.format('dog')
print(s)
```

```
dog
   dog
      dog
```

```
s = ‘{:3.1f}'.format(1.0)
print(s)
s = ‘{:4.2f}'.format(1.0)
print(s)
```

```
1.0
1.00
```

Built-in data structures

list	Ordered collection	mutable
tuple	Ordered collection	immutable
dict	Unordered collection with keyword access	mutable
set	Unordered collection	mutable
frozenset	Unordered collection	immutable
bytes	Ordered collection of bytes	immutable
bytearray	Ordered collection of bytes	mutable

LIST

- Ordered collection of elements → each element has an index
- The elements can be of any kind
- Elements of different kinds can be mixed in the same list
- Lists can be modified

Examples of list definition

L = [‘A’, ‘C’, ‘T’, ‘G’]	**[‘A’, ‘C’, ‘T’, ‘G’]**
L = [‘AA’, ‘AT’, ‘AC’, ‘AG’]	[‘AA’, ‘AT’, ‘AC’, ‘AG’]
L = [‘A’, ‘C’, ‘T’, ‘G’, ’A’, ’T’]	[‘A’, ‘C’, ‘T’, ‘G’, ’A’, ’T’]
L = list(‘TCAATGCG’)	[‘T’, ’C’, ‘A’, ‘A’, ‘T’, ‘G’, ‘C’, ‘G’] This syntax can be used with any iterable object

L1 = [‘A’, ‘T’]
L2 = [‘C’, ‘G’]
L3 = [L1, L2]
	[[‘A’, ‘T’], [‘C’, ‘G’]]

quando fai un assegnamento a una lista, è come se inizializzassi un nuovo puntatore, quindi modificando una lista si modifica anche l’altra

Indexing

3.2	‘A’	‘ATCG’	7.8	True	3.4	3.2	3.2
0	1	2	3	4	5	6	7
-8	-7	-6	-5	-4	-3	-2	-1

len(L)	Return the number of elements in L
L[i]	i-th element
L[i:j]	from the i-th element to the one before the j-th element
L[i:]	from the i-th element to the last one
L[:j]	from the first element to the one before the j-th element
L[:]	all the elements in the list
L[i:j:k]	from the i-th element to the one before the j-th element, selecting elements with step k
L[-i]	i-th element reading from the end, the index of the last element is -1

Using indexing to modify a list

L[i] = x	Change the i-th element to the value x
L[i:i] = M	Add the elements of M before the i-th element of L (M needs to be an iterable) L = [1,2,3] L[1:1] = [4,5] → [1,4,5,2,3]
L[i:j] = M	Change the elements of L from i to j-1 with the elements of M (M is an iterable) L = [1,2,3] L[1:2] = [4,5] → [1,4,5,3]
L[len(L):len(L)] = M	Add the elements of M at the end of L
del L[i]	Remove i-th element from the list. It changes the list itself
del L[i:j]	Remove elements for i to j-1
del L[i:j:k]	Remove elements from i to j-1 with step k

```
l = [32, 17, 1, 8, 21, 9]

l[1] = 18 → [32, 18, 1, 8, 21, 9]

l[1:1] = [89,] → [32, 89, 18, 1, 8, 21, 9]

l[1:3] = [17,] → [32, 17, 1, 8, 21, 9]

del l[3:5] → [32, 17, 1, 9]
```

Methods of lists

L.append(x)	Add the element x at the end of L
L.extend(M)	Add all the elements in M (iterable) at the end of L
L.insert(i,x)	Add the element x to L at position i
L.remove(x)	Remove the first occurrence of x from x Error if L does not include x
L.pop([i])	Remove and return the i-th element of the list (last element by default)
L.index(x)	Return the index of the first occurrence of x in L Error if L does not include x
L.count(x)	Count the occurrences of x in L
L.reverse()	Reverse the order of the elements in L
L.sort([reverse = False])	Sort the elements in the list in ascending order (descending if reverse is True)

Lists are mutable → these methods can modify the list itself

```
l = [32, 17, 1, 8, 21, 9]

l.append(8) → [32, 17, 1, 8, 21, 9, 8]

l.extend([17,35]) → [32, 17, 1, 8, 21, 9, 8,17,35]

L.remove(8) → [32, 17, 1, 21, 9, 8,17,35]

L.sort() → [1, 8, 9, 17, 17, 21, 32, 35]
```

Operators +,*,in, not in

```
l1 = [3,2,4]
l2 = 2*l1 → 3,2,4,3,2,4
l3 = [7,8] + l2 → 7, 8, 3, 2, 4, 3, 2, 4
8 in l3 → True
```

Common functions for lists

	L = [0,1,2,3]	
len(L)	4	Number of elements

min(L)
max(L)
	0
	3	

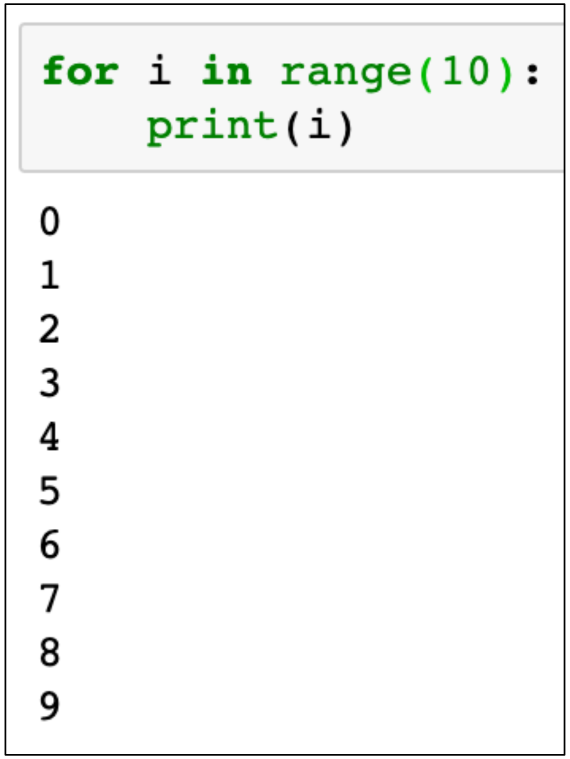
sum(L)	6	
any(L)	True	True if any element is True
all(L)	False	True is all the elements are True

They work for any iterable
If the corresponding operator is defined
(e.g. + for sum, > for max, etc)

la deviazione standard si calcola facendo sqrt(sum((l[i] - media)**2)/len(l))

uccidetemi
vi prego

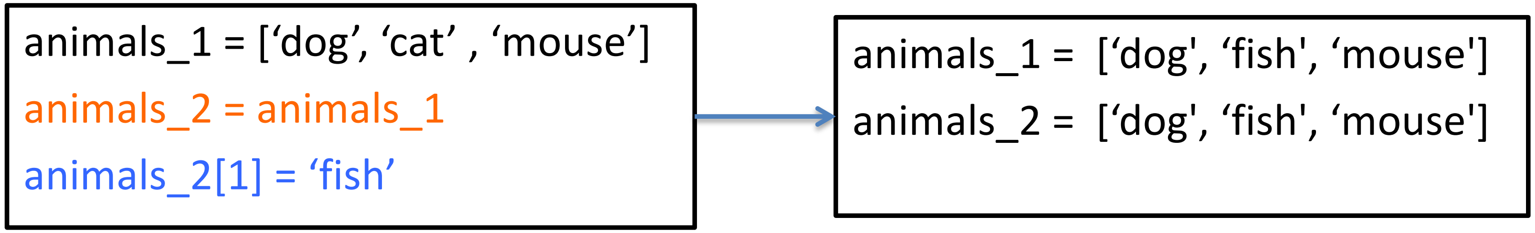
- range(N): return an iterable to the first N-1 integer values
  - range(N, M): return an iterable to the interger values from N to M-1



```
for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

Aliasing



```
animals_1 = [‘dog’, ‘cat’ , ‘mouse’]
animals_2 = animals_1
animals_2[1] = ‘fish’
```

```
animals_1 =  [‘dog', ‘fish', ‘mouse']
animals_2 =  [‘dog', ‘fish', ‘mouse']
```

- The orange command does not create a new list
- It creates a new label for the same object pointed by animals_1
- Thus, any change to animals_2 also affects animals_1

is operator

It checks if two names point to the same object (not if they are equal)

```
l1 = [1,2,3]
l2 = l1
l1 is l2 → True
```

```
l1 = [1,2,3]
l2 = [1,2,3]
l1 is l2 → False
```

```
l1 = [1,2,3]
l2 = [1,2,3]
l1 == l2 → True
```

copy method

```
l1 = [1,2,3]
l2 = l1.copy()
l1 is l2 → False
```

It creates a copy of the list

```
l1 = [1,2,3]
l2 = [4,5,6]
l3 = [l1, l2]
l4 = l3.copy()
l4 is l3 → False
l4[0] is l3[0] → True
```

But it does not copy the elements of the list !

```
from copy import deepcopy

l1 = [1,2,3]
l2 = [4,5,6]
l3 = [l1, l2]
l4 = deepcopy(l3)
l4 is l3 → False
l4[0] is l3[0] → False
```

Aliasing and binding operator

```
l1 = [1,2,3]
l2 = l1
l1 = l1 + [4,5,6]
```

The l1 defined in the last line points to a new object, so l2 still points to [1,2,3]

```
l1 = [1,2,3]
l2 = l1
l1 += [4,5,6]
```

Here, the l1 defined in the last line points to the same object defined in the first line, so in this case also l2 points to [1,2,3,4,5,6]

list Vs strings

strings are similar to lists, but with two important differences:

- all the items are characters
- strings are immutable

```
s = ‘dog’
s[0] = ‘f’ → ERROR
```

```
l = list(‘dog’)
l[0] = ‘f’ → now it works
```

tuples

- Similar to list but immutable

```
t1 = (1,2,3)
t2 = 4,5,6
t1[0] = 2 → ERROR
```

with or without parenthesis is the same

```
t1 = () # empty tuple
t2 = 2, # one element tuple
```

```
l1 = [1,2,3]
l2 = [4,5,6]
t1 = l1, l2
```

elements can be mutable objects

```
l1 = [1,2,3]
t1 = tuple(l1) # it works with any iterable
```

SET

- Unordered collection of elements

- Elements can be of different types

- Being unordered, indexing is not possible (and each object is included only once)

- Elements need to be hashable objects (immutable objects are hashable)

S = {‘A’, ‘C’, ‘T’, ‘G’}	{‘A’, ‘C’, ‘T’, ‘G’}
S = {‘AA’, ‘AT’, ‘AC’, ‘AG’}	{‘AA’, ‘AT’, ‘AC’, ‘AG’}
S = {‘A’, ‘C’, ‘T’, ‘G’, ’A’, ’T’}	{‘A’, ‘C’, ‘T’, ‘G’}
S = set(‘TCAATGCG’)	{‘A’, ‘C’, ‘T’, ‘G’} It works with any iterable object

S = { {‘A’, ’T’}, {‘C’, ’G’} }
S = { [‘A’, ’T’], [‘C’, ’G’] }
S = { (‘A’, ’T’), (‘C’, ’G’) }
	Error: sets are not hashable
	Error: lists are not hashable
	OK: tuples are hashable

S = {‘A’, 1, True, 2.7}	{‘A’, 1, True, 2.7}

Set operations

Operator	Method	

SET1 | SET2	SET1.union(SET2,…)	SET1 = {1,2,3}
		SET2 = {1,4,5)
		SET1 | SET2 → {1,2,3,4,6}

SET1 & SET2	SET1.intersection(SET2,…)	SET1 & SET2 → {1}
SET1 – SET2	SET1.difference(SET2,…)	SET1 – SET2 → {2,3}
SET1 ^ SET2	SET1.symmetric(SET2,…)	SET1 ^ SET2 → {2,3,4,5}

They do not modify SET1, they return new objects

Adding/removing elements

OPERATORE	METODO	EFFETTO

	SET1.add(item)	SET1 = {1,2,3}
		SET1.add(4) → SET1 = {1,2,3,4}

	SET1.remove(item)	SET1.remove(4) → SET1 = {1,2,3} Error if item is missing
	SET1.discard(item)	As before, but without error if item is missing

SET1 |= SET2	SET1.update(IT)	Add all the elements in IT to SET1, where IT is an iterable
		SET1.update([4,5,6])
		→ SET1 = {1,2,3,4,5,6}

They directly modify SET1

Set comparisons

Operator	Method	
	SET1.isdisjoint(SET2)	True if there's no element in common
SET1 <= SET2		True if all the elements in SET1 are also in SET2
SET1 < SET2	SET1.issubset(SET2)	True if SET3 is bigger than SET1, and all the elements in SET1 are also in SET2
SET1 >= SET2		
SET1 > SET2	SET1.issuperset(SET2)	

FROZENSET

- As sets but hashable/immutable

```
s1 = {1,2,3}
s1.add(4)
s2 = frozenset(s1)
s2.add(5) → ERROR
s3 = { frozenset({1,2}), frozenset({3,4}) }
```

it works with any iterable

it would not work with normal sets

DICTIONARY

- set of key:value pairs
- key needs to be hashable
- values can be anything

D = {‘A’: ‘adenine’, ‘T’: ‘thymine’,
‘C’:’cytosine’, ‘G’: ‘guanine’}

key	value
‘A’	‘adenine’
‘T’	‘thymine’
‘C’	‘cytosine’
‘G’	‘guanine’

posso usare solo dati non modificabili in un dizionario

D = {}	Empty dictionary
D[k]	Get the value corresponding to the key k Error if key does not exist
D.get(k, default = None)	Return the value corresponding to the key k If k is not a key, it returns the default value
D[k] = v	Add the k:v pair (or redefine it, if already there)
del D[k]	Remove the element with key k Error if k does not exist
D.keys()	Return an iterator to the keys
D.values()	Return an iterator to the values
D.items()	Return an iterator to key, value pairs
k in D (k not in D)	True if k is (is not) a key of D
len(D)	number of key:value pairs in D

List comprehensions

[expression for value in iterator if expression]

- The if part might be missing
- More than one for cycle (each one eventually including an if) can also be used

```
l = [n**2 for n in range(10)] → [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[x for x in l if (x % 3) == 0] → [0,9,36,81]
```

range(stop): generates a sequence of
numbers from 0 to stop (not included).
More to follow…

Other comprehensions

- If {} are used instead of [], a set is created

```
[n%2 for n in range(10)] → [0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
{n%2 for n in range(10)} → [0, 1]
```

- When using {}, a dict is created if the expression includes :

```
{n:(n%2 == 0) for n in range(10)} → {0:True, 1: False, 2:True, 3:False,…}
```

Function Definition

def** FunctionName**(…):

   “””doc string”””

   <Block of code>

```
def add_one(x):
   “””just adding 1”””
   y = x + 1
   return y
```

```
def make_even(x):
   “””return the even number >= x”””
   if x % 2 == 0:
      return x
   else
      return x + 1
```

- Function arguments are separated by commas
- return is used to stop execution and return a value (it can be None)
- If the function ends without a return, None is returned
- The doc_string might be missing (bad idea)
- When the doc_string is defined, it is returned by help(FunctionName)
- Functions are first class objects

Positional and keyword arguments

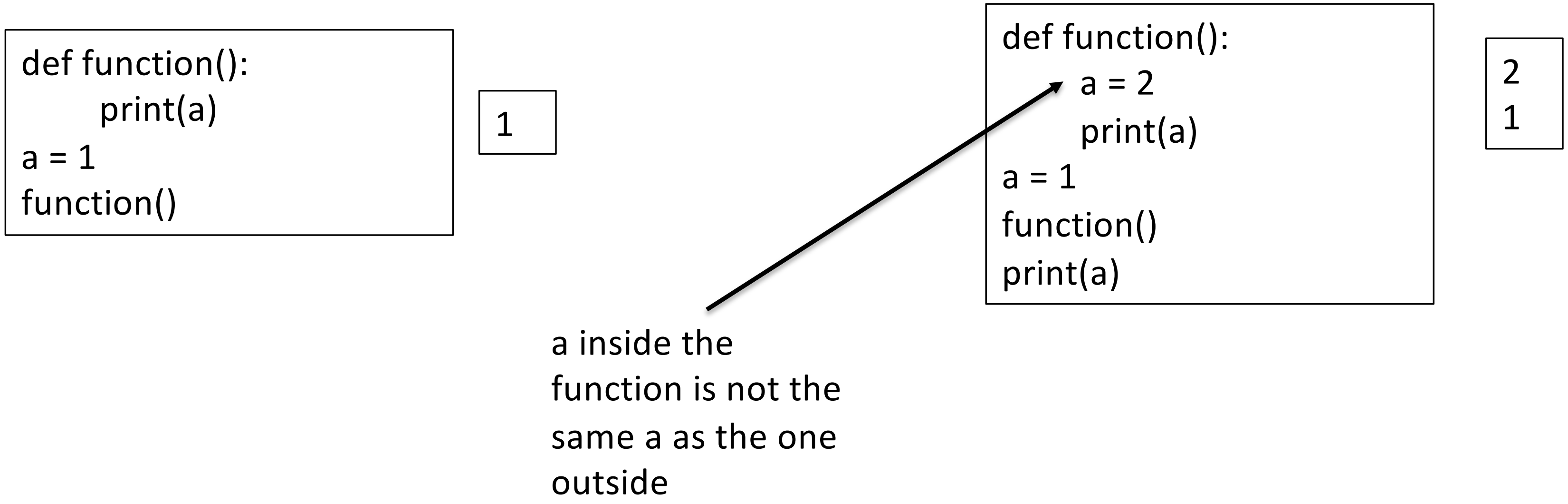
def Function(parp1, parp2,…, parpN, park1 = X1, , park2 = X2, …, parkM = XM):

- keyword arguments follow positional arguments
- When keyword arguments are not defined at function call, the default value is used

```
def function(p1, p2, k1 = 2, k2 = 3):
      print('p1 = ',p1,', p2 = ',p2,', k1 = ',k1,', k2 = ',k2)

function(0) → ERROR
function(0,1) → p1 =  0 , p2 =  1 , k1 =  2 , k2 =  3
function(0,1, k1 = 4) → p1 =  0 , p2 =  1 , k1 =  4 , k2 =  3
function(0,1, k2 = 5) → 1 =  0 , p2 =  1 , k1 =  2 , k2 =  5
function(0,1, k1 = 4, k2 = 5) → p1 =  0 , p2 =  1 , k1 =  4 , k2 =  5
function(k1 = 4, k2 = 5, 0, 1) → ERROR
function(k1 = 4, k2 = 5, p1 = 0, p2 = 1) → It works, but it’s horrible
```

Scope and Lifetime of variables

- Variables defined in the main body of the file are global
- Variables defined inside a block are local to that block
- more local variables override less local variable



```
def function():
        print(a)
a = 1
function()
```

```
1
```

```
def function():
        a = 2
        print(a)
a = 1
function()
print(a)
```

```
2
1
```

a inside the function is not the same a as the one outside

global

- The keyword global is needed when we want to change a global variable inside a function

```
def function():
        global a
        a = 2
        print(a)
a = 1
function()
print(a)
```

```
2
2
```

Now, we're saying that the variable a inside the function is the same variable a defined outside

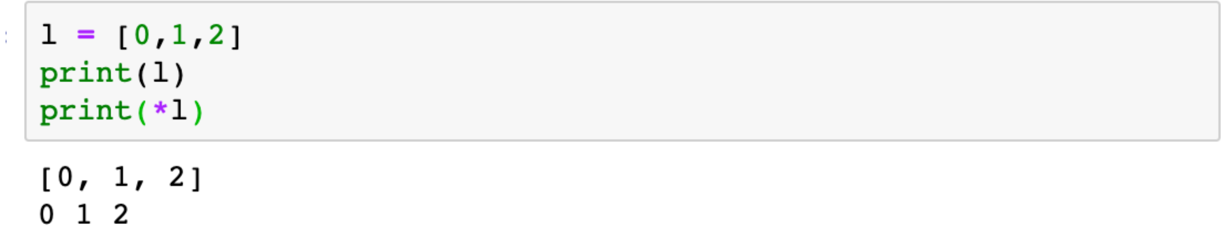
Beware: it's not possible to refer both to a local and a global variable with the same name
These programs raise an exception UnboundLocalError

```
def function():
        print(a)
        a = 2
        print(a)
a = 1
function()
```

```
def function():
        a = a + 1
a = 1
function()
```

*

*: expand a sequence in its components



```python
l = [0,1,2]
print(l)
print(*l)
```

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

```
def function(*args):
        print('Number of arguments: ',len(args))
        print('Arguments: ',args)
```

```
function(1)
```

```
Number of arguments: 1
Arguments: (1,)
```

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

```
Number of arguments: 2
Arguments: (1,2)
```

```
function(1,2,3)
```

```
Number of arguments: 3
Arguments: (1,2,3)
```

This syntax can be used to pass an arbitrary number of positional arguments to a function
It is convention to call the sequence of arbitrary positional arguments args (but anything would work, it’s the * making the trick)

```
def welcome(message,*args):
       print(message)
       for other_message in args:
            print(other_message)
```

```
welcome(‘Hello !’)
```

```
Hello !
```

```
welcome(‘Hello !’, ’Ciao !’, ‘Hola !’)
```

```
Hello !
Ciao !
Hola !
```

**: expand a dictionary in its key = value components

```
def function(**kwargs):
       print('Number of keyword arguments: ',len(kwargs))
       print('Arguments: ‘,kwargs)
```

```
function(k1 = 1)
```

```
Number of arguments: 1
Arguments: {‘k1’:1}
```

```
function(k1 = 1, k2 = 2)
```

```
Number of arguments: 2
Arguments: {‘k1’:1, ‘k2’:2}
```

This syntax can be used to pass an arbitrary number of keyword arguments to a function
It is convention to call the sequence of arbitrary keyword arguments kwargs (but anything would work, it’s the ** making the trick)

```
def welcome(message,**kwargs):
    print(message)
    for language, greeting in kwargs.items():
        print(language,’: ‘,greeting)
```

```
welcome(‘Greetings in different languages’, uk = ‘hello’, ita = ‘ciao’, spain = ‘hola’)
```

```
Greetings in different languages
uk: hello
ita: ciao
spain: hola
```

Remember: Functions are objects

So they can be assigned to variables

```
sum_all = sum
sum_all([1,2,3]) # it’s the same as sum([1,2,3])
```

… or they can be passed to other functions

```
def print_int_other_base(x, convert):
   print(convert(x))

print_int_other_base(21, bin) → 0b10101
print_int_other_base(21, oct) → 0o25
print_int_other_base(21, hex) → 0x15
```

… or it’s possible to define list of functions, dictionaries that have functions as values or keys, etc…

… or they can be returned by other functions

```
def convert2kelvin(input_scale):
   if input_scale == 'fahrenheit':
      def convert(temperature):
         return (temperature - 32) * 5/9 + 273.15
      return convert
   else:
      def convert(temperature):
         return 273.15 + temperature
      return convert

kelvin = convert2kelvin('celsius')
kelvin(0) → 273.15
kelvin = convert2kelvin('fahrenheit')
kelvin(32) → 273.15
```

Example: sort of list (iterable, key = None)

- Method that sort a list in place
- If key is defined, the elements are ordered using the result of this function

```
def age(person):
        return person[1]
def height(person):
        return person[2]
l = [ ['mario',54,173], ['vittorio',12,145], ['bruno',74,164], ['giorgio',23,187] ]

l.sort() → [['bruno', 74, 164], ['giorgio', 23, 187], ['mario', 54, 173], ['vittorio', 12, 145]]
l.sort(key = age) → [['vittorio', 12, 145], ['giorgio', 23, 187], ['mario', 54, 173], ['bruno', 74, 164]]
l.sort(key = height) → [['vittorio', 12, 145], ['bruno', 74, 164], ['mario', 54, 173], ['giorgio', 23, 187]]
```

Anonymous lambda functions (functions on one-line)

lambda variable(s): expression

It’s a method to define simple functions without name

```
square = lambda x: x**2
square(10) → 100
```

```
add = lambda x, y: x + y
add(1,2) → 3
```

```
l = [ ['mario',54,173], ['vittorio',12,145], ['bruno',74,164], ['giorgio',23,187] ]

l.sort() → [['bruno', 74, 164], ['giorgio', 23, 187], ['mario', 54, 173], ['vittorio', 12, 145]]
l.sort(key = lambda x: x[1]) → [['vittorio', 12, 145], ['giorgio', 23, 187], ['mario', 54, 173], ['bruno', 74, 164]]
l.sort(key = lambda x: x[2]) → [['vittorio', 12, 145], ['bruno', 74, 164], ['mario', 54, 173], ['giorgio', 23, 187]]
```

File Objects

file_object = open(*Name_of_the_file*, *mode*)

- Name_of_the_file is a string with the file name (full path, or with respect to the current path)
- *mode* is a string that defines the kind of the file and how you want to open the file

r	**Read from the beginning**	**Error if the file does not exist**
w	**Write from the beginning**	**If the file exists, it is overwritten**
a	**Write from the end**	**if the file does not exist, it is created**
r+	Read/write from the beginning	
w+	Read/write from the beginning	
a+	Read/write from the end	

t	text
b	binary

	text	**binary**
.read([N])	Read all the characters, or a maximum of N characters if N is defined	Read all the bytes (or N bytes if defined)
.readline([N])	Read up to the new line character (included), or up to a maximum of N characters, if N is defined	
.readlines()	Read all the lines and return a list of strings	
.write(item)	write the sequence of characters	write the sequence of bytes
.writelines(list)	write the elements of the list (strings)	

file.txt

```
Line 1
Line 2
Line 3
Line 4
```

f = open(‘file.txt’, ‘r’)	**x**
x = f.read(1)	“L”
x = f.read(1)	“i”
x = f.read(1)	“n”
x = f.read(7)	“e 1\nLin”
x = f.readline()	“ea 2\n”
x = f.readline()	“Linea 3\n”
x = f.readline()	“Linea 4\n”
x = readline()	“”
x = read()	“”

file_object.close()

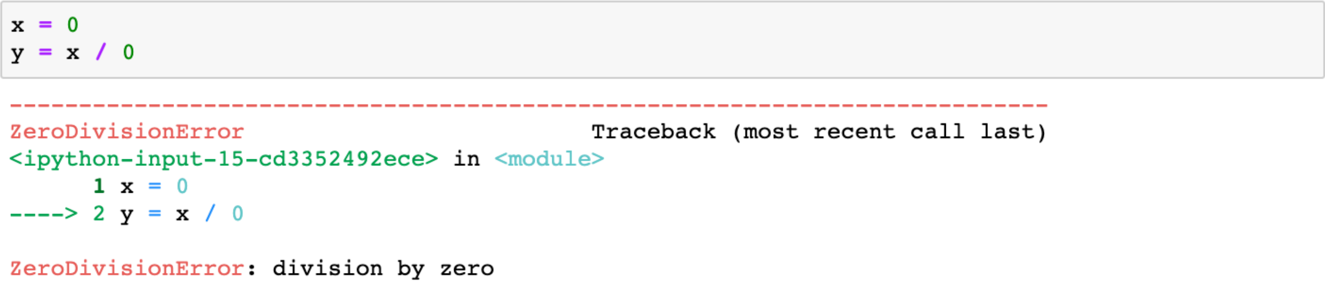
- Do it after the work with the file as finished
- Remember that writing operations are not guarantee to be synchronized until the file is closed
- To force writing to the disk, it is possible to use the method flush()

with … as

```
with open(’file.txt’, ‘rt’) as fin:
    s = f_in.readline()
    while s:
        print(s)
        s = f_in.readline()
```

A more compact syntax to not forget the close command

Exceptions



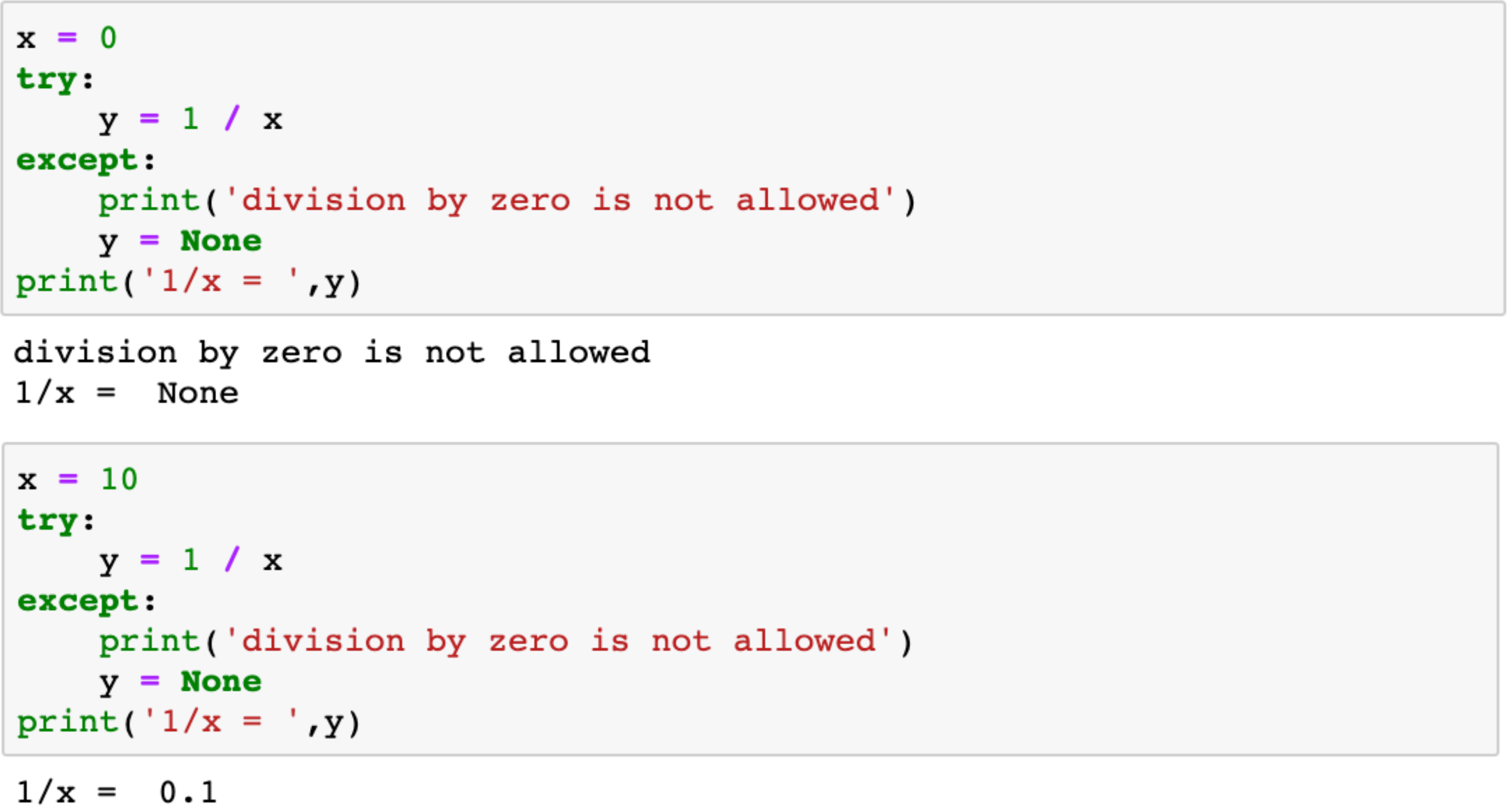
```
x = 0
y = x / 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-15-cd3352492ece> in <module>
      1 x = 0
----> 2 y = x / 0

ZeroDivisionError: division by zero
```

- python as a built-in system for dealing with run-time errors

Basic syntax

```
try:
   <Block of code>
except:
   <Block of code to execute if anything went wrong in the block above>
```



```
x = 0
try:
    y = 1 / x
except:
    print('division by zero is not allowed')
    y = None
print('1/x = ',y)
```

```
division by zero is not allowed
1/x =  None
```

```
x = 10
try:
    y = 1 / x
except:
    print('division by zero is not allowed')
    y = None
print('1/x = ',y)
```

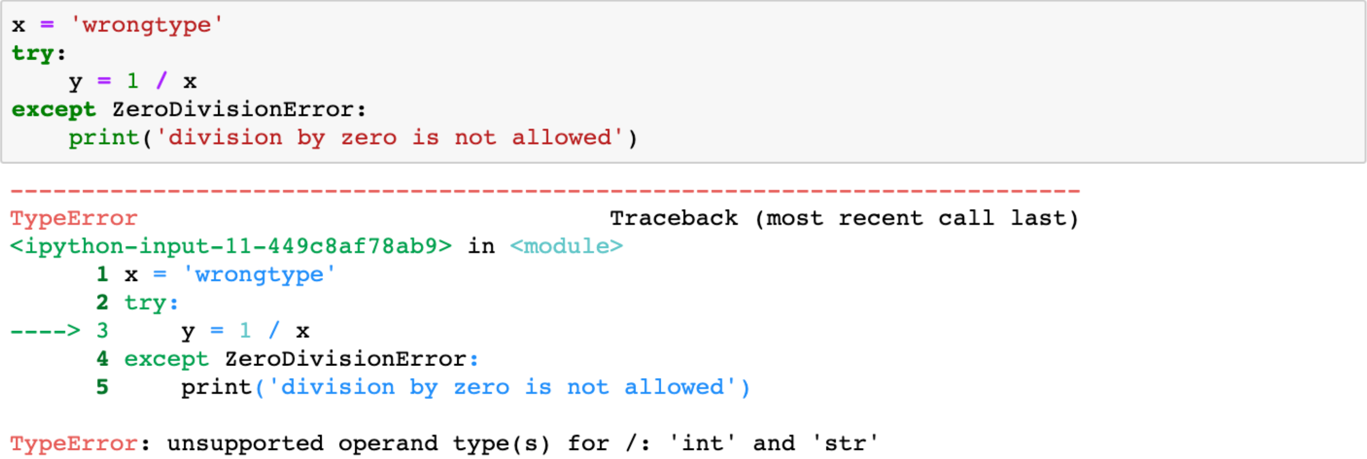
```
1/x =  0.1
```

- When catching the expression with the try statement, it is possible to deal with the error, and to decide how the program should continue
- Exceptions that are not catched cause the program to exist
- python has many types of exceptions

ZeroDivisionError	Raised when second operand of division or modulo operation is zero
ValueError	Raised when a function gets argument of correct type but improper value
TypeError	Raised when a function or operation is applied to an object of incorrect type
ImportError	Raised when the imported module is not found
KeyError	Raised when a key is not found in a dictionary
AttributeError	Raised when attribute assignment or reference fails
…	…

- It is possible to choose the kind of exception to capture

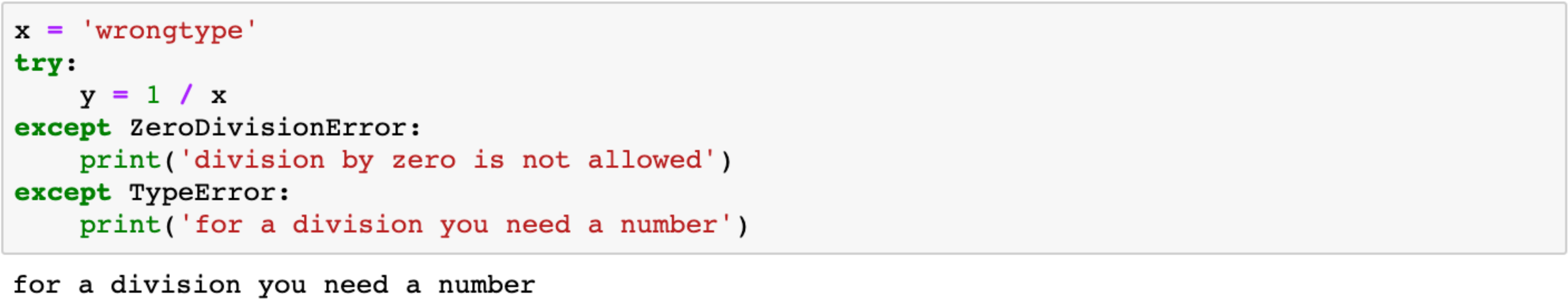
```
try:
    <Block of code>
except Exception1:
    <Block of code to execute if Exception1 occurred>
```



```python
x = 'wrongtype'
try:
    y = 1 / x
except ZeroDivisionError:
    print('division by zero is not allowed')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-449c8af78ab9> in <module>
      1 x = 'wrongtype'
      2 try:
----> 3     y = 1 / x
      4 except ZeroDivisionError:
      5     print('division by zero is not allowed')

TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

- It is possible to handle different exceptions in different ways



```
x = 'wrongtype'
try:
    y = 1 / x
except ZeroDivisionError:
    print('division by zero is not allowed')
except TypeError:
    print('for a division you need a number')
```

```
for a division you need a number
```

```
try:
    <Block of code>
except Exception1:
    <Block of code to execute if Exception1 occurred>
else:
    <Block executed if no exception occurred>
finally:
    <Block executed in any case>
```

Raising exceptions

- sometimes it is useful to raise a particular exception in your own code (so that another part of the program can handle it)
- Exceptions are raised by the raise function

```
def convert_kelvin_to_fahrenheit(temperature):
    if (not isinstance(temperature, float)) and (not isinstance(temperature, int)):
        raise TypeError('temperature is a number')
    if temperature < 0:
        raise ValueError('temperature in kelvin is >= 0')
    return (temperature - 273.15) * 9/5 + 32
```

Import

import MODULE
import the entire module, objects of the module are accessible with the syntax MODULE.object

import MODULE as NEW_MODULE_NAME
same as before, but now the module is accessible as NEW_MODULE_NAME

from MODULE import OBJECT
only the requested object is imported, and it is accessible directly as OBJECT

*from MODULE import **
all the objects of the module are imported, and they are accessible with their own name

Standard library

- os, sys, shutils: tools for interacting with the operative system
- glob: search the file system
- math, cmath: some basic mathematical tools
- pickle: writing/reading objects to disk
- collections: common data structures
- time, datetime: tools to handle time-related tasks
- csv: reading/writing csv files

https://docs.python.org/3/library/

Package and Environment Managing

- ANACONDA: default package and environment manager for python
  - Different versions for python2/3 (but each one can build environments for the other python version)
  - miniconda is a lightweight version

A conda environment is a set of system variables and libraries

- **conda update conda** update conda to the current version
- **conda env list** list all the environments available
- **conda create --name NAME [--clone NAME] [python=VERSION]**
  - create a new environment
  - it is possible to choose the python version, e.g. 2.7
  - with clone it is possible to inheritfrom a previous environment
- **conda activate NAME** activate the environment
- **conda deactivate** close the current environment
- **conda env remove --name NAME** delete the environment

- **conda list** List all the installed packages
- **conda list WHAT** List installed packages with name containing WHAT
- **conda search WHAT** List available packages with name containing WHAT
- **conda install PACKAGE** install the package and any dependency
- **conda install PACKAGE=VERSION** install that specific version of the package
- **conda uninstall PACKAGE** remove it (and everything depending on it)

- **conda config --show** show the entire configuration
- **conda config --get channels** which channels are used for getting packages
- **conda config --add channels NAME** add a new channel
  - the last added it he highest priority one
- **conda install -c CHANNEL1 [-c CHANNEL2, …] PACKAGE**
  - install the package using channel1 as highest priority

Object programming

- Imperative programming: Solve the task with a sequence of commands
- Object programming: Solve the task using objects and corresponding operations

- class = an abstract model for a group of entities (persons, vehicles, random number generators, …) → The data type
- object = an entity of a particular class → The variable
- attribute = a characteristic of the object (random number generator: mean, std, …)
- method = function offered to the outside world (random number generator: generate new sample)
- inheritance = relationship among classes (generators of random numbers with gaussian/uniform/… distributions are a special kind of random number generators)

```
class Point:
    “””A point in a 2D-dimensional space”””
   def __init__(self, x = 0.0, y = 0.0):
      self.x = x
      self.x = y

p1 = Point(2.3,6.1) # here an object of class Point is created
```

- The class can be named as any python object (but it is quite common to use names starting with a capital letter)
- __init__ is the constructor of the object. It’s called every time an object of the class Point is created
- The first argument of __init__ is always the object itself. It’s not compulsory to call it self (but it is highly recommended)
- For the other arguments of the __init__ method, the same rules discussed for function arguments apply
- Try help(Point) or help(p1)
- Try type(p1) or isinstance(p1, Point)

```
class Point:
  “””A point in a 2D-dimensional space”””
  def __init__(self, x = 0.0, y = 0.0):
    self.x = x
    self.y = y

p1 = Point(2.3,6.1)
print(p1) → <__main__.Point object at 0x102862be0>
```

__repr__: method that is used every time it is necessary to convert the object into a string

__str__: method that is used to convert the object into a string for printing; __repr__ is used if __str__ is not defined

```
class Point:
  “””A point in a 2D-dimensional space”””
  def __init__(self, x = 0.0, y = 0.0):
    self.x = x
    self.y = y
  def __repr__(self):
    return 'x = ‘+str(self.x)+’ y = ‘+str(self.y)

p1 = Point(2.3,6.1)
print(p1) → x = 2.3 y = 6.1
```

self is the first argument of all the methods

- Add to the class Point a method for calculating the distance from the origin

```
class Point:
     “””A point in a 2D-dimensional space”””
    def __init__(self, x = 0.0, y = 0.0):
        self.x = x
        self.y = y
    def __repr__(self):
        return 'x = ‘+str(self.x)+’ y = ‘+str(self.y)
def norm(self):
        return (self.x**2 + self.y**2)**0.5



p1 = Point(2.3,6.1)
print(‘distance of p1 from origin = ‘,p1.norm())
```

All the rules discussed for function definition apply to method definition

- Add a method that returns a new point rotated by teta degrees

```
class Point:
    def __init__(self, x = 0.0, y = 0.0):
        self.x = x
        self.y = y
    def __repr__(self):
        return 'x = ‘+str(self.x)+’ y = ‘+str(self.y)
    def norm(self):
        return (self.x**2 + self.y**2)**0.5
    def rotate(self, teta):
            import math
            x = math.cos(teta)*self.x - math.sin(teta)*self.y
            y = math.sin(teta)*self.x + math.cos(teta)*self.y
            return Point(x,y)


p1 = Point(2.3,6.1)
p2 = p1.rotate(3.14)
```

- Change the method rotate so that it changes the object itself, instead of returning a new object

```
class Point:
    def __init__(self, x = 0.0, y = 0.0):
        self.x = x
        self.y = y
    def __repr__(self):
        return 'x = ‘+str(self.x)+’ y = ‘+str(self.y)
    def norm(self):
        return (self.x**2 + self.y**2)**0.5
    def rotate(self, teta):
        import math
        x = math.cos(teta)*self.x - math.sin(teta)*self.y
        y = math.sin(teta)*self.x + math.cos(teta)*self.y
        self.x = x
        self.y = y
```

Remember aliasing

```
p1 = Point(1,2)
p2 = p1
print('p1: ', p1) → p1:  x = 1 y = 2
print('p2: ', p2) → p2:  x = 1 y = 2
p1.x = 3
print('p1: ', p1) → p1:  x = 3 y = 2
print('p2: ', p2) → p2:  x = 3 y = 2
```

This is a new link to the same object

So here, we’re changing the object pointed both by p1 and p2

Operator overloading

- It is possible to define how operators (+,-,==,…) work on object of your own classes

<	__lt__
<=	__le__
>	__gt__
>=	__ge__
==	__eq__
!=	__ne__

For binary operators, it is convention to call the second operand in the method definition *other*

+ +=	__add__ __iadd__
- -=	__sub__ __isub__
* *=	__mul__ __imul__
/ /=	__truediv__ __idiv__
// //=	__floordiv__ __ifloordiv
% %=	__mod__ __imod__
** **=	__pow__ __ipow__

unary operators

+	__pos__
-	__neg__
abs()	__abs__
int()	__int__
float()	__float__
round()	__round__
bool()*	__bool__

* This is also used when checking if an object is True

- Define the > operator of the class Point, so that p1 > p2 returns True if p1 is further away from the origin than p2

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return 'x = ‘+str(self.x)+’ y = ‘+str(self.y)
    def norm(self):
        return (self.x**2 + self.y**2)**0.5
    def __gt__(self, other):
        return self.norm() > other.norm()
```

Now sort knows what to do !

```
import random
points = [Point(random.uniform(0,1),
        random.uniform(0,1)) for i in range(10)]
points.sort()
```

- Define the == operator to check if two objects of the class Point are the same

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

p1 = Point(1,2)
p2 = Point(1,2)
print(p1 == p2) → False
```

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return (self.x == other.x) and (self.y == other.y)

p1 = Point(1,2)
p2 = Point(1,2)
print(p1 == p2) → True
```

- Define a class Polygon to represent polygon objects, where a polygon is defined as a sequence of objects of the class Point

```
class Point:
        def __init__(self, x, y):
            self.x = x
            self.y = y
class Polygon:
        def __init__(self, *args):
            self.vertexes = []
            for vertex in args:
                if not isinstance(vertex, Point):
                    raise TypeError()
                self.vertexes.append(vertex)

triangle = Polygon(Point(0,1), Point(0,1), Point(0.5,1))
```

Inheritance

- New classes are created inheriting from previous classes. Only differences between the parent class and the new class need to be implemented
- Methods defined in the new class override methods defined in the parent class

```
class New(Parent):
```

With this syntax, the class New inherits from the class Parent

```
class New:
```

```
class New(object):
```

Actually when the parent class is not defined, it is assumed equal to the class object, where object is a completely generic python class

- Define the classes Triangle and Rectangle as classes inheriting the class Polygon

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
class Polygon(object):
    def __init__(self, *args):
        self.v = []
        for point in args:
            self.v.append(point)
    def __getitem__(self, ind):
        return self.v[ind]
class Triangle(Polygon):
    def area(self):
        return 0.5*self.v[2].y*(self.v[1].x - self.v[0].x)
class Rectangle(Polygon):
    def area(self):
        return (self.v[1].x - self.v[0].x) * (self.v[2].y - self.v[0].y)
```

What if I want to check that the number of vertexes is 3 when a triangle is defined ?

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
class Polygon(object):
    def __init__(self, *args):
        self.v = []
        for point in args:
            self.v.append(point)
    def __getitem__(self, ind):
        return self.v[ind]
class Triangle(Polygon):
    def __init__(self, *args):
        if len(args) != 3:
            raise ValueError()
        self.v = []
        for point in args:
            self.v.append(point)
    def area(self):
        return 0.5*self.v[2].y*(self.v[1].x - self.v[0].x)
class Rectangle(Polygon):
    def area(self):
        return (self.v[1].x - self.v[0].x) * (self.v[2].y - self.v[0].y)
```

Here, part of the code is repeated between the __init__ method of Polygon and the __init__ method of Triangle
→ Difficult to maintain the code

But it is possible to call a method of one class from any other class

```
class Triangle(Polygon):
    def __init__(self, *args):
        if len(args) != 3:
            raise ValueError()
        Polygon.__init__(self, *args)
```

The built-in function super() returns the parent class

```
class Triangle(Polygon):
    def __init__(self, *args):
        if len(args) != 3:
            raise ValueError()
        super().__init__(*args)
```

With this syntax, self does not need to be passed as argument

enumerate(iter)

- It creates an iterator that provides the sequence of pairs (index, value) for all the elements in object iter

```
l = [7,3,9]
for i, x in enumerate(l):
        print(‘i = ‘,i,’, x = ’,x)
```

```
i = 0, x = 7
i = 1, x = 3
i = 2, x = 9
```

range(start, stop, step)

Sequence of ordered integer numbers

```
l = list(range(0, 10, 2))→ [0,2,3,6,8]
l = list(range(4))→ [0,1,2,3]
```

```
for value in range(10):
        print(value)
```

The full sequence of elements is never actually created
Thus, it does not occupy the memory