Scientific Programming in Python

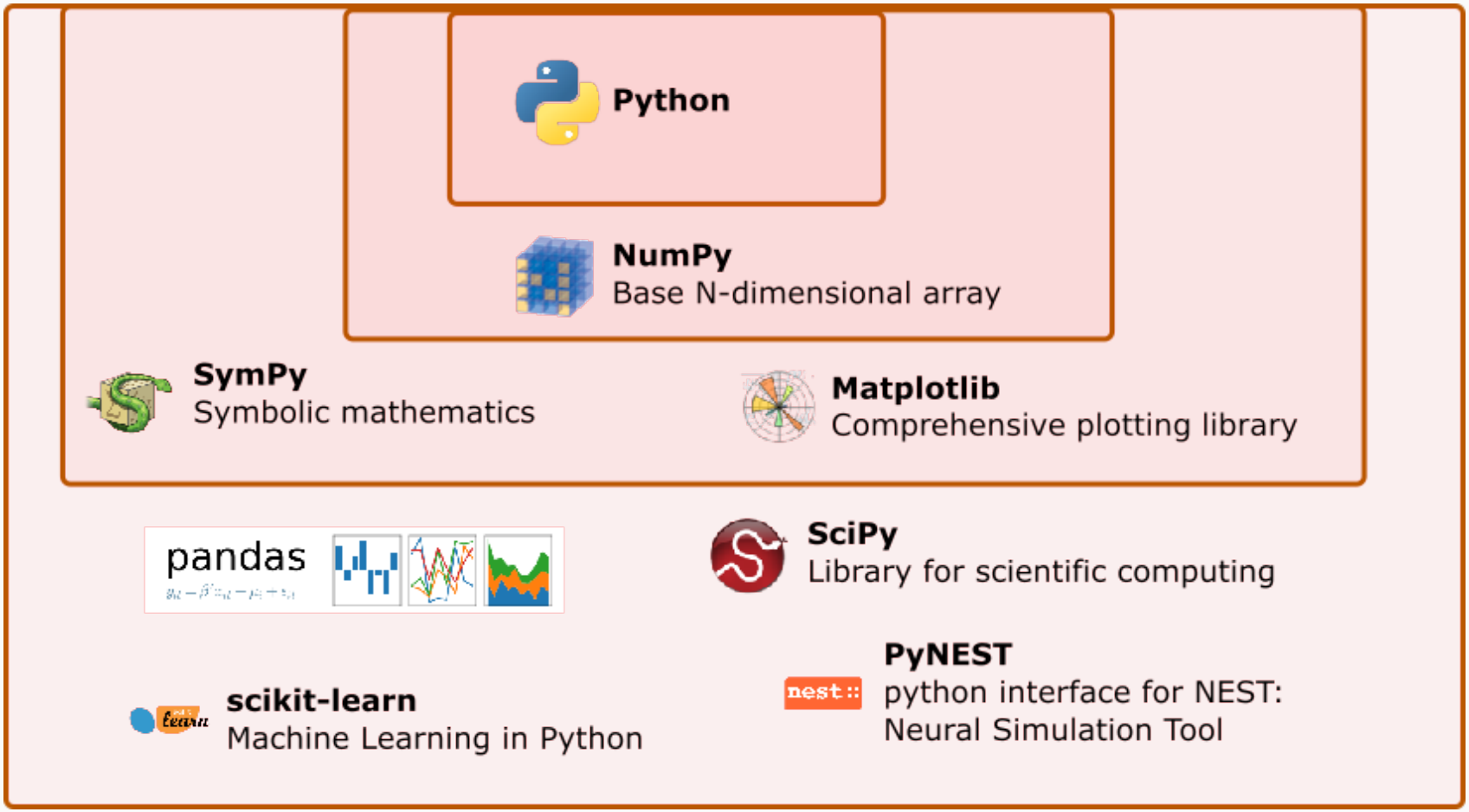
Francesco Mannella

Institute of Cognitive Sciences and technologies - CNR

Outline

Outline

Python basics - Python's scientific ecosystem



Python basics - the interpreter

Just type "python" in the command line to open:

```
$ python
Python 2.7.12 (default, Nov 19 2016,
      06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "
      license"
for more information.
>>>
```

It can be used as a calculator:

```
>>> 4 + 5
9
>>> _
```

You can execute single lines of code

```
>>> a = 5
>>> b = 3
>>> a + b
8
>>>
```

You can also run a control statement:

```
>>> for i in range(10):
...     print i**2
d..
0
1
4
9
16
25
36
49
64
81
>>>
```

Outline

Numeric types - hands-on code

Code: find if a number is a prime number

```
1  # Python program to check if the input number
2  # is prime or not
3
4  from __future__ import print_function
5  from __future__ import division
6
7  num = 17
8
9  # take input from the user
10 # num = int(input("Enter a number: "))
11
12 # prime numbers are greater than 1
13 if num > 1:
14    # check for factors
15    for i in range(2,num):
16        if (num % i) == 0:
17            print(num,"is not a prime number")
18            print(i,"times",num//i,"is",num)
19            break
20    else:
21        print(num,"is a prime number")
22
23 # if input number is less than
24 # or equal to 1, it is not prime
25 else:
26    print(num,"is not a prime number")
```

```
$ python prime.py
407 is not a prime number
11 times 37 is 407
```

Numeric types

```
# Boolean

a = True
b = False

# Integer

a = 34
b = 45//3
c = int(34/2)
d = int(3.14)
e = a + b//d
```

```
# Float

a = 2.3
b = 45/3
c = float(34)
d = b/a

# Complex

a = complex(1, 2)
b = 3 -4.5j
c = a**2 + b
d = c.real() # float
e = c.imag() # float
```

Outline

Operators - arithmetic operators

`a = 10; b = 20`

`a + b == 30`	**Addiction**	Adds values on either side of the operator.
`b - a == 10`	**Subtraction**	Subtracts right hand operand from left hand operand.
`a * b == 200`	**Multiplication**	Multiplies values on either side of the operator.
`b / a == 2` `a / 4.0`	**Division**	Divides left hand operand by right hand operand.
`b % a == 0` `a / 4.0`	**Modulus**	Divides left hand operand by right hand operand and returns remainder.
`a ** b`	**Exponent**	Performs exponential (power) calculation on operators.
`a // 4.0`	**Floor Division**	Division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity)

Operators - assignement operators

`a += b`	**Addiction**	Adds values on either side of the operator and assigns the result to left operand.
`b -= a`	**Subtraction**	Subtracts right hand operand from left hand operand and assigns the result to left operand.
`a *= b`	**Multiplication**	Multiplies values on either side of the operator and assigns the result to left operand.
`b / a == 2` `a / 4.0 == 2.5`	**Division**	Divides left hand operand by right hand operand and assigns the result to left operand.
`b % a == 0` `a / 4.0 == 2.5`	**Modulus**	Divides left hand operand by right hand operand, assigns the result to left operandand returns remainder.
`a ** b == 10`	**Exponent**	Performs exponential (power) calculation on operators and assigns the result to left operand.
`a // 4.0 == 2`	**Floor Division**	Perform floor Division and assigns the result to left operand.

Operators - comparison operators

`a == b`	**False**	Is *a* equal to *b*?
`a != b` `a <> b`	**True**	Is *a* different from *b*?
`a < b`	**True**	Is *a* less than *b*?
`a > b`	**False**	Is *a* greater than *b*?
`a <= b`	**True**	Is *a* less than or equal to *b*?
`a >= b`	**False**	Is *a* greater than or equal to *b*?

Operators - logical operators

```
a = True
b = False
```

`a and b`	**False**	are both *a* and *b* true?
`a or b`	**True**	is *a* or *b* true?
`not a`	**False**	is not *a* true?

Outline

Strings

```
name = "Federico"
surname = "Rossi"
street = 'via' + 'A. De Sanctis'
number = 6
zipcode = 100
city = "Roma"

address = """
Federico Rossi
via A. De Sanctis, 6
00100 Roma
"""
```

```
address0 = "%s %s\n%s, %d\n%05d %s" % (name,
       surname, street, number, zipcode, city)

address1 = "{} {}\n{}, {}\n{:05d} {}".format(
       name, surname, street, number, zipcode,
       city)

address2 = """
{} {}
{} {}
{:05d} {}
""".format(name, surname, street, number,
       zipcode, city )

address3 = r"{} {}\n{}, {}\n{:05d} {}\".format(
       name, surname,  street, number, zipcode,
        city)
```

```
>>> print(address0)
Federico Rossi
via A. De Sanctis, 6
00100 Roma
>>>
>>> print(address1)
Federico Rossi
via A. De Sanctis, 6
00100 Roma
>>>
```

```
>>> print(address2)
Federico Rossi
via A. De Sanctis, 6
00100 Roma
>>> print(address3)
Federico Rossi\nvia A. De Sanctis, 6\n00100
       Roma
>>> _
```

Outline

Control statements

IF statement

```
print_on = True
if print_on :
    print('print this!')

if print_on is True:
    print('print this!')
```

```
if a > 0.5:
    b += a
```

```
a = 7
if a < 5:
    print('less then 5')
elif 5 <= a < 10:
    print('between 5 and 10')
elif 10 <= a <= 15:
    print('between 10 and 15')
else:
    print('greater than 15')
```

FOR loops

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

```
for i in range(10):
    if i == 8:
        continue
    print(i)
```

```
a = []
for i in range(5):
    a += i

a == [0, 1, 2, 3, 4]
```

WHILE loops

```
i = 1
while i < 6:
    print(i)
    i += 1
```

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

Outline

Functions and Lists - hands-on code

The **quicksort** algorithm:

```
1  from __future__ import print_function
2  from __future__ import division
3
4  def quickSort(arr):
5      """
6      The QuickSort algorithm is an
7      efficient sorting algorithm,
8      serving as a systematic method for
9      placing the elements of an array
10     in order.
11
12     Args:
13       :param arr: a list containing the
14                   elements to sort
15       :return:  the sorted list of arguments
16     """
17     less = []
18     pivotList = []
19     more = []
20     if len(arr) <= 1:
21         return arr
22     else:
23         pivot = arr[0]
24         for i in arr:
25             if i < pivot:
26                 less.append(i)
27             elif i > pivot:
28                 more.append(i)
29             else:
30                 pivotList.append(i)
31         less = quickSort(less)
32         more = quickSort(more)
33         return less + pivotList + more
34
35 if __name__ == "__main__":
36     a = [4, 65, 2, -31, 0, 99, 83, 782, 1]
37     a = quickSort(a)
38     print(a)
```

```
$ python quicksort.py
[-31, 0, 1, 2, 4, 65, 83, 99, 782]
```

Lists

Lists:

- are ordered
- can be filled with objects of all types
- can be nested
- are mutable
- are dynamic

```
a = []
b = [1, 2, 3, 4, 5]
c = ['one', 2.0, 3, True, [1,
      2]]
```

You can append new items:

```
a.append(3)
a == [1, 2, 3, 4, 5, 3]
```

Recall elements and slices:

```
b[0] == 1
c[:2] == ['one', 2.0]
c[1:3] == [2.0, 3]
c[-1] == [1, 2]
c[-3:] ==  [3, True, [1, 2]]
```

```
d = range(6)
d == [0, 1, 2, 3, 4, 5]

e = range(2, 7)
e == [2, 3, 4, 5, 6]

f = range(2, 9, 2)
f == [2, 4, 6, 8]


f[::-1] == [8, 6, 4, 2]
d[::-2] == [5, 3, 1]

for i in c:
    print(i)
```

Strings are a special kind of list:

```
g = 'This is a string'
g[-5:] == 'tring'
```

You can split a string into a list of strings:

```
h = f.split(' ')
h == ['This', 'is', 'a', '
      string']
```

Outline

Tuples - hands-on code

Code: A simple perceptron

```
data = [[[0.44, 0.83], 0],
        [[0.83, 0.66], 1],
        [[0.52, 0.83], 0],
        [[0.84, 0.55], 1],
        [[0.71, 0.92], 0],
        [[0.51, 0.15], 1],
        [[0.24, 0.35], 0],
        [[0.34, 0.43], 0],
        [[0.29, 0.81], 0],
        [[0.66, 0.3 ], 1]]

epochs = 2
eta = 1.0

weights = [0, 0]

def step_fun(x):

    if x > 0:
        return 1
    else:
        return 0
```

```
for item in data:
    inp, lab = item

    # potential
    pot = 0
    for i, x in enumerate(inp):
        pot += x * weights[i]

    # activation
    act =  step_fun(pot)

    # learn
    for i, x in  enumerate(inp):
        weights[i] += eta * x * (lab - act)


    print(lab, act, weights)
    print("")
```

Tuples

Tuples are used to initialize many objects at once

```
a, b = (2, 3)
c, _, d = 4, 3, 1
```

A function can return more than one element by packing objects in a tuple

```
def division(numerator, denominator):
    res = numerator // denominator
    remainder = numerator % denominator
    return res, reminder

n, _ = division(23, 4)
print(n) # n == 5

t = division(14, 5)
print(t[0])    # t[0] == 2
print(t[1])    # t[1] == 4
```

Tuples are immutable:

```
a = ['one', 2, 3.0, 'four']
del a[1]    # Correct. a becomes
            # ['one', 3.0, 'four']
a[2] = 45 # Correct, a becomes
          # ['one', 3.0, 45]

b = (0, 45, 'giallo', 6.0)
del b[0]    # Error!! cannot delete
            # elements
b[1] = 'new' # Error!! cannot change values
```

Zip

Both **lists** and **tuples** (and **strings**) are **containers**. A container is an object that contains `references` to other objects. Containers can be iterated upon (they are also called **iterables**), meaning that you can traverse through all the values.

The **zip()** function takes iterables (can be zero or more), makes an iterator that aggregates elements based on the iterables passed, and returns an iterable of tuples.

```
1 days = [27, 28, 30]
2 weekdays = ('Monday', 'Tuesday', 'Thursday')
```

```
>>>zip(index, weekdays)
[(27, 'Monday'), (28, 'Tuesday'), (30, 'Thursday')]
```

zip() is often used in for loops:

```
1 for i, wd in zip(index, weekdays):
2     print i, wd, weekdays[i]
```

Enumerate

When you iterate over a list or tuple you often need to have both the value of each element and its position in the iterable.

```
1 sentence = 'This is a string'
2 for i, ch in zip(range(len(sentence)),
3                  sentence):
4     if ch == ' ':
5         print 'character %d is a space' % i
```

In those case the **enumerate** function simplifies the code:

```
7 for i, ch in enumerate(sentence):
8     if ch == ' ':
9         print 'character %d is a space' % i
```

More on functions, abstraction - hands-on code

Code: you can separate code parts in functions

```
12 epochs = 2
13 eta = 1.0
14
15 weights = [0, 0]
16
17 def step_fun(x):
18
19     if x > 0:
20         return 1
21     else:
22         return 0
23
24 for epoch in range(epochs):
25     for item in data:
26         inp, lab = item
27
28         # potential
29         pot = 0
30         for i, x in enumerate(inp):
31             pot += x * weights[i]
32
33         # activation
34         act =  step_fun(pot)
35
36         # learn
37         for i, x in  enumerate(inp):
38             weights[i] += eta * x * (lab - act)
39
40
41         print(lab, act, weights)
42         print("")
```

More on functions, abstraction - hands-on code

Code: you can separate code parts in functions

```
12 epochs = 2
13 eta = 0.01
14
15 weights = [0, 0]
16
17 def step_fun(x):
18
19     if x > 0:
20         return 1
21     else:
22         return 0
23
24 def wsum(vec, weights):
25     res = 0
26     for i, x in enumerate(vec):
27         res += x*weights[i]
28     return res
29
30 def learn(eta, inp, out, teach, weights):
31     for i, x in  enumerate(inp):
32         weights[i] += eta * x * (teach - out)
```

```
35 for epoch in range(epochs):
36     for item in data:
37         inp, lab = item
38
39         # potential
40         pot = wsum(inp, weights)
41
42         # activation
43         act = step_fun(pot)
44
45         # learn
46         learn(eta, inp, act, lab, weights)
47
48         print(lab, act, weights)
49         print("")
```

Outline

Value vs. reference assignement

- Immutable types (bool, int, float, complex, tuple) can be only passed by value.
- Mutable types (list, dictionary, custom objects) are passed by reference.

```
 7 a = 1
 8 b = a
 9
10 a = 34444
```

```
>>>print(a)
1
>>>print(b)
3444
```

```
 7 a = [1, 2, 3]
 8 b = a
 9
10 a[2] = 34444
```

```
>>>print(a)
[1, 2, 3444]
>>>print(b)
[1, 2, 3444]
```

```
 7 def foo(arg):
 8     arg += 99
 9
10 a = 1
11 foo(a)
```

```
>>>print(a)
1
```

```
 7 def foo(cont):
 8     for i in range(len(cont)):
 9         cont[i] += 99
10
11 a = [1, 2, 3]
12 foo(a)
```

```
>>>print(a)
[100, 101, 102]
```

Outline

Dictionaries - hands-on code

Code: building a function for creating histograms

```
from __future__ import print_function
from __future__ import division

def histogram(data, n_bins=10):
    ''' Create an histogram
        :param data: A list with all values
        :n_bins: Classes of data
    '''

    # Find the minimum value
    min_num = min(data)

    # Find the maximum value
    max_num = max(data)

    # Compute the range of each bin
    gap = (max_num - min_num)/n_bins

    # Compute the limits of bins
    bin_lims = []
    for bin_el in range(n_bins):
        bin_lims.append([
            min_num + bin_el * gap,
            min_num + (bin_el + 1) * gap])

    # Compute the frequence for each bin
    freqs = {}
    for el in data:
        for i, lims in enumerate(bin_lims):
            if lims[0] <= el < lims[1]:
                if i in freqs.keys():
                    freqs[i] += 1
                else:
                    freqs[i] = 1

    # Sum of frequencies
    tot = sum(freqs.values())

    # Plot the histogram
    for idx, freq in freqs.items():
        # Compute the proportion in each bin
        prop = freq / tot
        # Each star in the string is 1% of
      values
        stars = ("*" * int(100*(prop)))
        # Put together all params for printing
        els = bin_lims[idx] + [freq, stars]
        # It must be a tuple (not a list)
        els = tuple(els)
        # Fill the format string and print it
        print("%5.2f <-> %5.2f: %#3d %s" % els)

    return freqs, bin_lims
```

Dictionaries - hands-on code

Code: building a function for creating histograms

```
54 if __name__ == "__main__":
55
56     # Load data from file
57     data = []
58     with open("hist_data.txt", "r") as datafile:
59         for line in  datafile.readlines():
60             data.append(float(line))
61
62     # make histogram
63     histogram(data)
```

```
>>> python hist.py
11.34 <-> 12.00:   1
12.00 <-> 12.66:  10 *
12.66 <-> 13.32:  24 **
13.32 <-> 13.98: 117 ***********
13.98 <-> 14.64: 205 ********************
14.64 <-> 15.30: 260 **************************
15.30 <-> 15.95: 237 ************************
15.95 <-> 16.61:  95 *********
16.61 <-> 17.27:  41 ****
17.27 <-> 17.93:   9
```

Dictionaries

Dictionaries

- are **iterables**
- are maps between keys and values
- keys can be of any non-iterable type
- values can be of any non-iterable type
- Each key is unique

Initializing:

```
a = {}
b = {'one': 232, 'two': 2.3}
```

Fill up:

```
a[1] = 3
a['new'] = 0.04
# a == {1: 3, 'new': 0.04}
```

Get keys:

```
k = a.keys()
# k == [1, 'new']
```

Get values:

```
v = a.values()
# v == [3, 0.04]
```

Iterate through keys-value pairs:

```
for k, v in a.items():
    print('{}: {}'.format(k, v))
```

Outline

Custom objects - hands-on code

Code: creating new types of objects

```
12 epochs = 2
13 eta = 0.01
14
15 weights = [0, 0]
16
17 def step_fun(x):
18
19     if x > 0:
20         return 1
21     else:
22         return 0
23
24 def wsum(vec, weights):
25     res = 0
26     for i, x in enumerate(vec):
27         res += x*weights[i]
28     return res
29
30 def learn(eta, inp, out, teach, weights):
31     for i, x in  enumerate(inp):
32         weights[i] += eta * x * (teach - out)
```

```
35 for epoch in range(epochs):
36     for item in data:
37         inp, lab = item
38
39         # potential
40         pot = wsum(inp, weights)
41
42         # activation
43         act = step_fun(pot)
44
45         # learn
46         learn(eta, inp, act, lab, weights)
47
48         print(lab, act, weights)
49         print("")
```

Custom objects - hands-on code

Code: creating new types of objects

```
epochs = 2

def step_fun(x):
    if x > 0:
        return 1
    else:
        return 0

def wsum(vec, weights):
    res = 0
    for i, x in enumerate(vec):
        res += x*weights[i]
    return res

class Perceptron:

    def __init__(self, eta, out_fun):

        self.eta = eta
        self.out_fun = out_fun
        self.weights = [0, 0]

    def activation(self, inp):

        pot = wsum(inp, self.weights)
        act = step_fun(pot)

        return act, pot

    def learn(self, inp, out, teach):
        for i, x in  enumerate(inp):
            self.weights[i] += self.eta * x * (
      teach - out)

perc = Perceptron(eta=0.1, out_fun=step_fun)

for epoch in range(epochs):
    for item in data:
        inp, lab = item

        act,_ = perc.activation(inp)
        perc.learn(inp, act, lab)

        print(lab, act, perc.weights)
        print("")
```

Custom objects

A class is a declaration of a custom type of objects

```
class NewType:
    def __init__{self}:
        self.a_data_member = []
    def add{self, x}:
        self.a_data_member.append(x)
    def sum(self):
        return sum(self.a_data_member)
```

An object is an element (or instance) of a class:

```
my_object = NewType()

my_object.add(3)
my_object.add(5)
my_object.add(7)
res = my_object.sum()    # res == 15
```