Scientific Programming in Python

Francesco Mannella

Institute of Cognitive Sciences and technologies - CNR

Outline

Outline

Functions and Lists - hands-on code

The **quicksort** algorithm:

```python
from __future__ import print_function
from __future__ import division

def quickSort(arr):
    """
    The QuickSort algorithm is an
    efficient sorting algorithm,
    serving as a systematic method for
    placing the elements of an array
    in order.

    Args:
      :param arr: a list containing the
                  elements to sort
      :return:  the sorted list of arguments
    """
    less = []
    pivotList = []
    more = []
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        for i in arr:
            if i < pivot:
                less.append(i)
            elif i > pivot:
                more.append(i)
            else:
                pivotList.append(i)
        less = quickSort(less)
        more = quickSort(more)
        return less + pivotList + more

if __name__ == "__main__":
    a = [4, 65, 2, -31, 0, 99, 83, 782, 1]
    a = quickSort(a)
    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.

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

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

zip() is often used in for loops:

```
for i, wd in zip(index, weekdays):
    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.

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

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

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

More on functions, abstraction - hands-on code

Code: you can separate code parts in functions

```
epochs = 2
eta = 1.0

weights = [0, 0]

def step_fun(x):

    if x > 0:
        return 1
    else:
        return 0

for epoch in range(epochs):
    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("")
```

More on functions, abstraction - hands-on code

Code: you can separate code parts in functions

```python
epochs = 2
eta = 0.01

weights = [0, 0]

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

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

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

        # potential
        pot = wsum(inp, weights)

        # activation
        act = step_fun(pot)

        # learn
        learn(eta, inp, act, lab, weights)

        print(lab, act, weights)
        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.

```python
a = 1
b = a

a = 34444
```

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

```python
def foo(arg):
    arg += 99

a = 1
foo(a)
```

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

```python
a = [1, 2, 3]
b = a

a[2] = 34444
```

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

```python
def foo(cont):
    for i in range(len(cont)):
        cont[i] += 99

a = [1, 2, 3]
foo(a)
```

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

Outline

Dictionaries - hands-on code

Code: building a function for creating histograms

```python
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

```python
if __name__ == "__main__":

    # Load data from file
    data = []
    with open("hist_data.txt", "r") as datafile:
        for line in  datafile.readlines():
            data.append(float(line))

    # make histogram
    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:

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

Fill up:

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

Get keys:

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

Get values:

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

Iterate through keys-value pairs:

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

Outline

Custom objects - hands-on code

Code: creating new types of objects

```python
epochs = 2
eta = 0.01

weights = [0, 0]

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

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

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

        # potential
        pot = wsum(inp, weights)

        # activation
        act = step_fun(pot)

        # learn
        learn(eta, inp, act, lab, weights)

        print(lab, act, weights)
        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
```