

Python Cheatsheet

Fabrizio Macià and Pablo Angulo for ESTIN@UPM.ES

python

Comments

Lines that start with # are comments, and are not executed

```
a = 1
# a = 2
# => a is still 1
```

Base types

bool *booleans* takes only two values: True and False.

int positive and negative *integers*, not bounded

float *floating point* numbers approximate any real number (e.g. $-1.25e-6$ ($-1.26 \cdot 10^{-6}$))

str A *string* is a sequence of characters. Always represented with quotes or double quotes (e.g. 'Hello, World', or "Hello, World")

Variables

Identifiers start with a letter or _, may contain numbers.

identifier = value binds the value to the identifier

my_number = 1 Binds identifier my_number to the integer 1.

my_number += 10 Equivalent to my_number = my_number + 10.

a=b=1 Binds both identifiers a and b to the integer 1.

a,b = 1,2 Binds a to 1, and b to 2.

a,b = ["one", "two"] *unpacks* list ["one", "two"], binding a to "one", and b to "two"

Inmutable container types

Immutable types can not be modified, but new objects can be built from the old ones.

str Can only hold characters (e.g. a = "Hello, World").

tuple May hold any data type (e.g. b = (12, True, "abc")).

Operations with containers

len(container) Returns the number of elements in container.

container[index] Access an element within the container. Index starts at 0, last element has index -1: (e.g. b[0] => 12, print(b[-1]) => "abc").

container[start:end] Get a subsequence, or slice. start index is included, end index is excluded: (e.g. a[0:5] => "Hello").

container1 + container2 concatenate compatible containers (e.g. a + "!" => "Hello, World!").

container*number repeat container (e.g. "abc"*3 => "abcabcabc").

Formatting strings

Old style formatting (I) Place one **format code** inside the string, follow the string by a % sign, then a variable. The variable will replace the format code.

```
nducks = 7
'...and %d ducks came'%nducks # => '...and 7 ducks came'
```

Old style formatting (II) Place *more than one* format code inside the string, followed by a *tuple* with as many elements as format codes in the string

```
animal = 'duck'
weight = 3.1416
mystring = 'The %s weights %.3f kg'%(animal, weight)
# mystring : 'The duck weights 3.142 kg'
```

New style formatting An **f-string** may have references to any variables defined before.

```
animal = 'duck'
weight = 3.1416
mystring = f'The {animal} weights {weight:.4} kg'
# mystring : 'The duck weights 3.142 kg'
```

Mutable container types

list May hold any data type (e.g. l = [12, True, "abc"]).

l[1:3]=[False, False] Updates a whole slice => l is [12, False, False].

l.append(1e3) Add an element at the end of the list => l is [12, False, False, 1e3].

dict Holds (key, value) pairs (e.g. d = {1:"one", 2:"two"}).

d[1]="uno" Update an existing element d => {1:"uno", 2:"two"}.

d[4]="four" Adds a new (key,value) pair d => {1:"uno", 2:"two", 4:"four"}.

Conversions

1 + 1.0 sum of int and float automatically promotes to float.

int(5.56) (=> 5) int() of a float truncates the decimal part.

round(5.56) (=> 6) round() of a float rounds to the nearest int.

np.round(5.56) (=> 6.0) while method round() from numpy rounds to the nearest integer, but the result is of type float.

int("14") returns the integer 14 (but int('5.56') throws an error!).

float("12.3") returns the float 12.3.

str(2.34) returns the string "2.34".

bool(x) return False if x is None, the boolean False, the number 0, or an empty container.

Conditionals

Exactly one of the indented blocks of an "if/elif/else" statement will be executed (or maybe none of them if there is no "else" clause):

```
if some_condition:
    do_this()
elif other_condition:
    do_that()
else:
    do_whatever()
```

Combine conditions with and, or, and using parenthesis.

```
if (a>0) and (a+b<=1):
    do_this()
    and_this()
# One equal sign = for assignment, two == for
  comparison, != for "not equal"
elif (len(d)!=1) or (a=="Hello, Planet"):
    do_that()
```

```
# next line is not indented, so it will be executed in
  any case
and_do_this_regardless()
```

Loops

for loops

for loops repeat some statements while one variable runs over the elements of an iterable:

```
s = 0
# when the loop starts, s is 0
for x in [1,2,3,4]:
    s = s + x
# when the loop ends, s is 1+2+3+4 = 10
# the final value, 10, will be printed only once
print(s)
```

while loops

while loops repeat some statements while a certain condition is satisfied

```
s = 0
# when the loop starts, s is 0
while s < 5:
    s = s + 1
# when the loop ends, s is 5
# the final value, 5, will be printed only once
print(s)
```

break out of a loop

The keyword break stops execution of a loop.

```
s = 0
for i in range(1, 100):
    s = s + i
    if s==3:
        break
# s is 0 + 1 + 2 + 3 = 6
```

List comprehensions

- Transform a list

```
>>> [x**2 for x in range(4)]
[0, 1, 4, 9]
```

- Filter a list

```
>>> [x for x in range(4) if x%2==1]
[1, 3]
```

- Both things at once

```
>>> [x**2 for x in range(4) if x%2==1]
[1, 9]
```

Functions

In the **function definition**, the **body** of the function is indented. Statements in the function body are not executed when the function is defined.

```
def mysum(x,y):
    '''this text describes the purpose of the function

    It is called the docstring.'''
    s = x + y
    return s
```

In a **function call**, the body of the function is executed.

```
# the identifier z is bound to the integer 3
z = mysum(1,2)
```

Default values for optional arguments

If a function argument has a default value, it is optional:

```
def sum_of_powers(x, y, p=2):
    return x**p + y**p
>>> sum_of_powers(2, 2)
8
>>> sum_of_powers(2, 2, p=3)
16
```

python libraries

import modules

- Import a module


```
import numpy
my_array = numpy.zeros(10)
```
- Import a module using an alias


```
import numpy as np
x = np.pi/2
```
- Import specific functions from a module


```
>>> from numpy import sin, arcsin
>>> print( sin(arcsin(1)) )
1
```

Anonymous functions

The `lambda` keyword gives an alternative way to define functions:

```
filter_multiples_of_3 = lambda x: (x % 3 == 0)
```

Any function that can be defined with `lambda` can also be defined with `def`:

```
def filter_multiples_of_3(x):
    return x % 4 == 0
```

numpy

`numpy` provides **arrays**, which are mutable data structures, but with fixed size. They are very efficient, and are designed for numerical computation. Many famous libraries are built on top of `numpy`.

```
import numpy as np
```

- Build an array from a list


```
>>> xs = np.array([1,10,100])
```
- Add a number to all elements of the array


```
>>> xs + 1
array([1,11,101])
```
- Apply a function to all elements of the array


```
>>> np.log10(xs)
array([0,1,2])
```
- Add two arrays


```
>>> xs + np.log10(xs)
array([1,11,102])
```
- Fill a one dimensional array with numbers from 0 to n (n is not included)


```
>>> np.arange(3)
array([0, 1, 2])
```
- Fill a one dimensional array with n floating points equispaced from a to b (both a and b are included)


```
>>> np.linspace(1,2,5)
array([1, 1.25, 1.5, 1.75, 2])
```
- Fill a one dimensional array with n zeros


```
zs = np.zeros(n)
```
- Fill a two dimensional, $n \times m$ array (a.k.a. a matrix) with zeros.


```
A = np.zeros((3, 4))
```
- Fill a 5×5 array with the value 7.


```
B = 7*np.ones((5,5))
```
- $*$ is the element-wise product of arrays, $@$ is the matrix “dot product”.


```
B = 7*np.ones((5,5))
# 5x5 identity matrix
Id = np.eye(5)
# D is diagonal, with '7' in the diagonal
D = Id*B
# E = B
E = Id@B
```
- `reshape` changes the dimensions of the array, as long as it keeps the same number of elements.


```
>>> M = np.arange(12).reshape((3,4))
>>> M
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```
- Sum elements of a matrix along different axis


```
>>> M.sum(axis=0)
array([12, 15, 18, 21])
>>> M.sum(axis=1)
array([ 6, 22, 38])
```
- Keep only first row, second to last column

```
>>> M[0, 1:]
array([ 1,  2,  3])
```

- An array of booleans can work as a slice

```
>>> v = np.arange(10,20)
>>> v[ v%2 == 1 ]
array([11, 13, 15, 17, 19])
```

matplotlib

`matplotlib` can build many types of graphics that represent quantitative information. The submodule `pyplot` makes it easy to use.

Line plot of a function

```
import matplotlib.pyplot as plt
def myfun(x):
    return np.sin(x**2 + 1)
```

```
xmin, xmax = -3, 3 # plotting interval
N = 100 # number of subdivisions
xs = np.linspace(xmin, xmax, N)
ys = myfun(xs)
plt.plot(xs, ys, 'g') # 'g' for green
```

Combine plots, with labels and titles

```
fun1 = np.exp
fun2 = lambda xs: np.exp(xs+1)
```

```
xmin, xmax = -1,8
N = 100
xs = np.linspace(xmin, xmax, N)
ys1 = fun1(xs)
plt.plot(xs, ys1, 'g', label='exp(x)')
```

```
ys2 = fun2(xs)
plt.plot(xs, ys2, 'b-', label='exp(x+1)')
```

```
plt.title('Plot of two functions')
plt.xlabel('x')
plt.legend()
plt.show()
```

