A Python tutorial

Step-by-step introduction to Python for the C/C++/Java programmer

Devert Alexandre

Hello, world !

First things first

```
print 'Hello, world!'
>>> Hello, world !
```

Hello, world !

String can be double-quoted too

```
print "Hello, world !"
>>> Hello, world !
```

Print function

The print function can take any number of argument of any type

```
print 'You are', 23, 'years old'
You are 23 years old
```

Basic types

But what are the basic types ?

- strings
- integer value
- floating point value
- list
- tuples
- map
- objects
- functions
- more ...

Fun with numbers

Let's start with numbers

```
2 + 3 * 5
>>> 17
```

Fun with numbers

Note that division with integer yield an integer

```
(2 + 3 * 5) / 2
>>> 8
```

If you want a float, divide with floats

```
(2 + 3 * 5) / 2.
>>> 7.5
```

Fun with numbers

A new thing for you C/C++/Java/C# people : the ** operator

```
3 ** 2
>>> 9

4 ** -2
>>> 0.0625
```

$$0.0625 = \frac{1}{4^2}$$

Fun with numbers

Some built-in functions are nice to now

```
max(3, 4, 5, 7)
>>> 7

min(3, 4, 5, 7)
>>> 3

sum((3, 4, 5, 7))
>>> 19
```

Why the double parenthesis for *sum* ? We will see that later.

Fun with numbers

The *math* module brings a lot of useful math function

```
import math
math.cos(.5 * math.pi)
```

Note that the *math* module have to be imported to be used

Fun with strings

Let's play with strings now

```
'you' + 'me'
>>> 'youme'

3 * 'ha'
>>> 'hahaha'
```

Fun with strings

A powerful way to build strings : the % operator

```
'You␣are␣%d␣years␣old' % (2012 - 1981)
>>> 'You␣are␣31␣years␣old'

'%s␣is␣%d␣years␣old' % ('Alex', 2012 - 1981)
>>> 'Alex␣is␣31␣years␣old'
```

There is a whole mini-language for string formatting. More about it in the Python documentation

Fun with strings

Just so you know : both are empty strings

```
''
""
```

Fun with strings

Python comes with a high quality hash function implementation.

```
hash('weed')
>>> -3668931459657339441
```

Actually, many other things are hashable. More on this later.

Fun with strings

Python comes with a couple of basic string processing function

```
'mechanic robotic'.split()
>>> ['mechanic', 'robotic']

'low'.upper()
>>> 'LOW'

'UP'.lower()
>>> 'up'

'lololol'.count('lo')
>>> 3
```

Any string is an object with the *string* type. All we do here, is calling methods of the *string* class. More on objects later.

Conditionals

Before moving towards more exciting things, the old classics

```
if name == 'alex':
    print 'ho, not him !'
```

Note the ':' symbol, and the indentation. Forget them, and Python will angry, very angry at you ⇒ syntax error, no less

Conditionals

Conditions are especially readable

```
if not hungry:
  print 'I am not hungry'
```

not is to express the negation of a condition

Conditionals

```
if have_beer and have_pizza:
  print 'I'm fine'
```

and is just a logical and

Conditionals

```
if have_beer or have_pizza:
  print 'Why not BOTH ?'
```

or is just a logical or

Conditionals

Alternative syntax to a conditional, with the classic *else*

```
if name == 'alex':
    print 'ho, not him !'
else:
    print 'feels good man'
```

Conditionals

Multiple conditions can be chained together

```
if a == 3:
  print 'lol'
elif a == 2 and b == 42:
  print 'bloop'
elif a == 5 or b == 69:
  print 'burp'
else:
  print 'noooooo'
```

This also the equivalent to *switch* . . . *case*

Boolean expressions

Conditions are expressed with *boolean* expressions

```
1 == 1
>>> True

1 == 2
>>> False

True and False
>>> True

True or not False
>>> True
```

Booleans are a type like any other in Python

Loops

Everybody loves a loop

```
while thirsty:
  print 'give␣me␣a␣beer␣!'
```

Again ':' and indentation are important in Python

Loops

One can stop or shortcut a loop with a *break* or a *continue*

```
while thirsty:
  if have_beer:
    print 'thanks !'
    break

  print 'give me a beer !'
```

Works exactly like in C or Java

Lists

Lists are an essential part of the language.

```
[1, 2, 3, 'lol', 5.6, True]
```

A list can contain anything, mixing different types

Lists

List can be used as arrays

```
['alex', 'xiang', 'omar'][0]
>>> 'alex'

['alex', 'xiang', 'omar'][1]
>>> 'xiang'

['alex', 'xiang', 'omar'][2]
>>> 'omar'
```

Lists

Using negative index gives the elements in reverse order

```
['alex', 'xiang', 'omar'][-1]
>>> 'omar'

['alex', 'xiang', 'omar'][-2]
>>> 'xiang'

['alex', 'xiang', 'omar'][-3]
>>> 'alex'
```

Lists

The * and + operators works like for strings

```
[1, 2, 'wei'] + [7, 'tom', 9]
>>> [1, 2, 'wei', 7, 'tom', 9]

[3, 'san'] * 3
>>> [3, 'san', 3, 'san', 3, 'san']
```

Lists

Just so you know : an empty list

```
[]
```

Lists

A powerful concept : *slices*, a way to describe sub-lists

```
[0, 1, 2, 3, 4, 5, 6][:2]
>>> [0, 1]

[0, 1, 2, 3, 4, 5, 6][2:]
>>> [2, 3, 4, 5, 6]

[0, 1, 2, 3, 4, 5, 6][2:5]
>>> [2, 3, 4]

[0, 1, 2, 3, 4, 5, 6][1:-1]
>>> [1, 2, 3, 4, 5]
```

Lists

List can be used as . . . lists !

```
a = [0, 1, 2, 3, 4, 5, 6]
a.pop()
>>> 6

a
>>> [0, 1, 2, 3, 4, 5]
```

Lists

List can be used as . . . lists !

```
a = [0, 1, 2, 3, 4, 5, 6]
a.pop(0)
>>> 0

a
>>> [1, 2, 3, 4, 5, 6]
```

Lists

List can be used as . . . lists !

```
a = [0, 1, 2, 3, 4, 5, 6]
a.pop(2)
>>> 2

a
>>> [0, 1, 3, 4, 5, 6]
```

Lists

List can be used as . . . lists !

```
a = [0, 1, 2, 3, 4, 5, 6]
a.append(9)
a
>>> [0, 1, 2, 3, 4, 5, 6, 9]
```

Lists

List can be used as . . . lists !

```
a = [0, 1, 2, 3, 4, 5, 6]
a.insert(0, 9)
a
>>> [9, 0, 1, 2, 3, 4, 5, 6]
```

Lists

List can be used as . . . lists !

```
a = [0, 1, 2, 3, 4, 5, 6]
a.insert(3, 9)
a
>>> [0, 1, 2, 9, 3, 4, 5, 6]
```

Lists

List can be used as . . . lists !

```
a = [0, 1, 2, 3, 4, 5, 6]
a.insert(-2, 9)
a
>>> [0, 1, 2, 3, 4, 9, 5, 6]
```

Lists

Using *del* and the slice notation, parts of a list can be deleted

```
a = [0, 1, 2, 3, 4, 5, 6]
del a[2]
a
>>> [0, 1, 3, 4, 5, 6]
```

Lists

Using *del* and the slice notation, parts of a list can be deleted

```
a = [0, 1, 2, 3, 4, 5, 6]
del a[2:5]
a
>>> [0, 1, 5, 6]
```

Lists

Using *del* and the slice notation, parts of a list can be deleted

```
a = [0, 1, 2, 3, 4, 5, 6]
del a[2:]
a
>>> [0, 1]
```

Lists

Using *del* and the slice notation, parts of a list can be deleted

```
a = [0, 1, 2, 3, 4, 5, 6]
del a[:5]
a
>>> [5, 6]
```

Lists

Using *del* and the slice notation, parts of a list can be deleted

```
a = [0, 1, 2, 3, 4, 5, 6]
del a[:]
a
>>> []
```

Lists

Finding an item in a list

```
[1, 3, 5, 9, 6].index(5)
>>> 2

[1, 2, 3, 4, 'potatoe'].index('potatoe')
>>> 4
```

Lists

Counting occurences of an item in a list

```
[1, 3, 5, 9, 6, 9].count(9)
>>> 2
```

Lists

Checking if an item is in a list

```
9 in [1, 3, 5, 9, 6]
>>> True

42 in [1, 3, 5, 9, 6]
>>> False

15 not in [1, 3, 5, 9, 6]
>>> True
```

Lists

Checking if the elements of two list are equals

```
[1, 2, 3] == [1, 1 + 1, 5 - 2]
>>> True

[1, 2, 3] == [1, 2, 5]
>>> False

[1, 2, 3] == [1, 2, 3, 4]
>>> False
```

Lists

Lists can be compared if its element can be compared : *lexicographic order*

```
[1, 2, 3] < [1, 2, 4]
>>> True

[1, 2, 3] < [3, 2, 1]
>>> True

[1, 2, 3, 4] < [1, 3, 1]
>>> True
```

Tuples

Tuples are *immutable* lists : lists that can not be modified

```
(1, 2, 3, 4, 'potatoe')
```

Tuples

Python syntax gotcha : a tuple of one element

```
('potatoe', )
```

And an empty tuple

```
()
```

Tuples

You can convert a list to a tuple and vice-versa

```
tuple([1, 2, 3, 4, 'potatoe'])
>>> (1, 2, 3, 4, 'potatoe')

list((1, 2, 3, 4, 'potatoe'))
>>> [1, 2, 3, 4, 'potatoe']
```

Tuples

Because tuples are *immutable*, they can be hashed

```
hash((1, 2, 3, 4, 'potatoe'))
>>> 8048177300002279206
```

Thus, the numerous technics based on hashing can work on tuples.

Tuples

As far you do read-only operations, tuples behave like a list

```
(1, 2, 3, 4, 'potatoe')[0]
>>> 1

(1, 2, 3, 4, 'potatoe')[1:4]
>>> (2, 3, 4)

(1, 2, 3, 4, 'potatoe')[-1]
>>> 'potatoe'

(1, 2, 3) + (4, 5)
>>> (1, 2, 3, 4, 5)

(1, 2) * 3
>>> (1, 2, 1, 2, 1, 2)
```

Tuples

Why tuples ?

- Can not be changed by accident
- Thread-safe
- Hashable
- Might be more efficient

Tuples

Tuples can be also created implicitly

```
a = 4, 5, 6
a
>>> (4, 5, 6)
```

Tuples

Tuples (explicitly or implicitly created) can be used to affect several values at once

```
a, b, c = ('alex', 3, 5)
a
>>> 'alex'

b
>>> 3

c
>>> 5
```

Tuples

Tuples (explicitly or implicitly created) can be used to affect several values at once

```
a, b, c = 'alex', 3, 5
a
>>> 'alex'

b
>>> 3

c
>>> 5
```

Tuples

Tuples (explicitly or implicitly created) can be used to affect several values at once

```
a = [1, 2, 3]
a[:] = 4, 5, 6
a
>>> [4, 5, 6]
```

Tuples

Tuples (explicitly or implicitly created) can be used to affect several values at once

```
a = [1, 2, 3]
a[:] = (4, 5, 6)
a
>>> [4, 5, 6]
```

Tuples

A nice side effect of this : shuffling values is a very natural

```
a, b = 'first', 'last'
a, b = b, a
a
>>> 'last'
b
>>> 'first'
```

Tuples

A nice side effect of this : shuffling values is a very natural

```
a, b, c = 'first', 'middle', 'last'
a, b, c = b, c, a

a
>>> 'middle'

c
>>> 'last'

a
>>> 'first'
```

Sequences

strings, list and tuples are sequences. Some functions works for *all* kind of sequences

```
len('fruit')
>>> 5

len(['alex', 'xiang', 42])
>>> 3

len(('alex', 'xiang', 42))
>>> 3
```

Sequences

More examples of functons working for all kind of sequences.

```
min([1, 3, 5, 9, 6])
>>> 1

min(['tom', 'alex', 'emilio', 'aaron'])
>>> 'aaron'
```

Sequences

More examples of functons working for all kind of sequences.

```
max([1, 3, 5, 9, 6])
>>> 9

max(['tom', 'alex', 'emilio', 'aaron'])
>>> 'tom'
```

Sequences

More examples of functons working for all kind of sequences.

```
sum([1, 3, 5, 9, 6])
>>> 24
```

Sequences

More examples of functons working for all kind of sequences.

```
count(([1, 3, 5, 9, 6, 9], 9)
>>> 2
```

Functions

Functions are defined as follow

```
def my_function(x, y):
  return x ** 2 + 2 * y + 1
```

Again, notice the importance of : and indentation in Python

Functions

Functions can return more than one parameter

```
def my_function(x):
  return x ** 2, x

my_function(3)
<<< (9, 6)
```

Remember, what we saw with implicity defined tuples . . .

Functions

You can use * to give the elements of a list as parameters to a function

```
def my_function(a, b):
  return a + b

a = (2, 3)
my_function(*a)
<<< 5

my_function(2, 3)
<<< 5
```

Functions

Functions *always* return a value.

```
def print_something():
    print 'HO HAI GUYS'

print_something()
>>> None
```

None is what returns a function that return nothing !

For loops

So far, to count from 0 to 9 in Python, we would do

```
i = 0
while < 10:
  print i
  i += 1
```

Not very convenient . . .

For loops

Counting from 0 to 9 in Python, the idiomatic way

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

For loops

Range is a function generating a list

```
range(10)
>>> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

For loops

Range is a function generating a list

```
range(5, 10)
>>> [5, 6, 7, 8, 9]
```

For loops

Range is a function generating a list

```
range(10, 5, -1)
>>> [10, 9, 8, 7, 6]
```

For loops

for in is a way to iterate over a sequence, any sequence

```
for i in [4, 42, 7, 'potato']:
  print i
```

For loops

If the items of a sequence are composite, we can retrieve the components separately

```
a = ((1, 2), (3, 4), (5, 6), (7, 8))
for i, j in a:
    print i, j
```

For loops

If the items of a sequence are composite, we can retrieve the components separately

```
a = (((1, 2), 3), ((4, 5), 6))
for (i, j), k in a:
    print i, j
```

Generators

Generators are functions returning a list of values one by one. Instead of using *return* to return one value, they use *yield* to deliver a value and continue

```
def odd_range(n):
  for i in range(n):
    yield 2 * i + 1

[odd_range(5)]
>>> [1, 3, 5, 7, 9]
```

Generators

Generators are a *very* powerful way to define sequences

- items produced only when needed $\Rightarrow$ lazy evaluation
- allows to manipulate huge lists while using little memory
- functional programming style
- leads to *very* compact implementations of algorithms

Generators

A compact way to write generators : *list comprehension*

```
[2 * x + 1 for x in range(5)]
>>> [1, 3, 5, 7, 9]
```

Sequence processing in one line and easy to read !

Generators

list comprehension support conditionals

```
[x for x in range(10) if x % 2 == 1]
>>> [1, 3, 5, 7, 9]
```

Note that despite the compactness, it's easy to read

Generators

list comprehension support conditionals

```
[x for x in range(10) if x % 2 == 1]
>>> [1, 3, 5, 7, 9]
```

Note that despite the compactness, it's easy to read

Generators

Conditionals with *list comprehension* supports *else* too

```
[x if x % 2 == 1 else -x for x in range(10)]
>>> [0, 1, -2, 3, -4, 5, -6, 7, -8, 9]
```

Generators

Functions working on sequences work on generators too

```
sum(2 * i + 1 for i in range(50))
>>> 2500
```

A good example of why Python code can be so compact

Generators

Python gives a few useful generators.

```
list(enumerate(('alpha', 'beta', 'gamma')))
>>> [(0, 'alpha'), (1, 'beta'), (2, 'gamma')]
```

enumerate produces a list of tuples *(index, item)*

Generators

Example with *enumerate* : a simple string checksum algorithm

```
my_string = 'fabulous'
sum((2 ** i) * ord(c) for i, c in enumerate(my_string))
>>> 29112
```

Remember, strings are sequences too !

Generators

```
a = ['apple', 'orange', 'banana']
b = ['rice', 'corn', 'potatoe']
zip(a, b)
>>> ('apple', 'rice'), ('orange', 'corn'),
('banana', 'potatoe')
```

zip produces a list of tuples *(item1, item2)* from the items of two lists.

Objects

Objects in Python looks like this

```
class Book:
  def __init__(self, title, author):
    self.title = title
    self.author = author

my_book = Book("Life of Wei", "Wei")
```

The class name is *Book*

Objects

Objects in Python looks like this

```
class Book:
  def __init__(self, title, author):
    self.title = title
    self.author = author

my_book = Book("Life of Wei", "Wei")
```

__init__ is the constructor of *Book*

Objects

Objects in Python looks like this

```
class Book:
  def __init__(self, title, author):
    self.title = title
    self.author = author

my_book = Book("Life of Wei", "Wei")
```

2 member variables : *title*, *authors*

Objects

Objects in Python looks like this

```
class Book:
  def __init__(self, title, author):
    self.title = title
    self.author = author

my_book = Book("Life of Wei", "Wei")
```

my_book is an instance of *Book*

Objects

Objects in Python looks like this

```
class Book:
  def __init__(self, title, author):
    self.title = title
    self.author = author

my_book = Book("Life of Wei", "Wei")
```

self is the instance of the object

Objects

Python object model, not like C++ or Java object model

- Only one constructor, with special name : *__init__*
- *self*, the object instance, is explicitly passed as first parameter of any method of an object
- All members of an object are public. No private, no protected.
- Member variables are created by the constructor. Actually, you can create them any time you like. It's just good practice to do it in constructor.