



Kari BME ösztöndíj
2018/19. tanév tavaszi félév
Egyéb szakmai tevékenység igazolása

Név: Reizinger Patrik

Neptun kód: W5PDBR

Feladatléírás: Segédanyag készítése Python programnyelvhez

Tanszék: TMIT

Kelt: 2019.02.05

Reizinger Patrik

Python under the hood — tips and tricks from a C++-programmers' perspective 01



SmartLab AI [Follow](#)

Sep 5, 2018 · 6 min read

Author: Patrik Reizinger

How you ever felt the urge to quickly try out some novel ideas and turn them into code? After that, have you ever felt that allocating memory on the heap and black pointer magic is not exactly helping you to be creative?

Then you just stumbled upon Python, which seemed to provide a just too easy solution for that. Well, that can not be true that something is so handy with a vast number of packages for almost every scientific domain or everyday task.

Yeah, Python rocks in many ways, but for a C/C++-programmer it may need a small change of point of view, or shall we call it programming paradigm. As being in that shoes a year before, I have decided to write some posts about how Python works and give away my tips & tricks which can spare you a significant amount of puzzling looks above seemingly well-written but (also seemingly) erroneous code.

So this series will provide you some a **general introduction** and **tips & tricks** for Python, from a C/C++-programmers point of view, while her/his background is the only thing I assume you possess.

I intend to associate my posts with code examples, either with separate ones (e.g. as Gists on GitHub) or just as with a few lines inline. I might prefer the former solution due to better syntax highlighting capabilities on this blog. I may also suggest some article worth reading, if you have further interest in some topics.

The schedule is: firstly I will show you some rather significant and interesting nuances of the language, after that, might you be with me that time, dig deeper—which I highly recommend, because you should have to do something with deep learning if you read this blog (although, my content will be also useful for you if you don't). This post

will clarify for you, what exactly Python is and after that, we investigate through examples the object management system of Python.

We shall begin, shouldn't we?

The language of Python

Just for clarification, we will discuss the categorization of Python, namely: is it *compiled* or *interpreted*? Named after the British comedian group Monty Python, what should we have expected? The answer, honestly, is ***both***.

Yes, we can see the assembly of Python code (we need for that the `dis` package). But this bytecode is executed on a *Python virtual machine*.

Furthermore, Python is only a *language concept*, the implementation can be diverse: you most probably use CPython (the most popular one), but there exist such things as Jython and IronPython also.

You still with me? I hope this small piece of theoretical monologue does not frightened you to get to know Python a bit more thoroughly.

Object management

For in Python ***everything is an object***, they management should be crucial for efficient code. With everything, I really mean everything, e.g. Python has ***no native number type***, so integers and floats are also objects, which results in a significant overhead—one major reason why the most efficient numerical libraries (e.g. NumPy) are not written in Python, they just have an interface for this superb language.

```
integer = 5
floating_point = 5.0
string = "Five"
l = list()
d = dict()

print("integer is an object: ", isinstance(integer,
object)) # True
print("floating_point is an object: ",
isinstance(floating_point, object)) # True
print("string is an object: ", isinstance(string,
object)) # True
print("l is an object: ", isinstance(l, object)) # True
print("d is an object: ", isinstance(d, object)) # True
```

Since everything is an object, the fact seems feasible that Python is ***not typed***, i.e. you can assign a class instance to a variable, after that you change your mind and the variable will hold an integer. Alright, then?

If you are familiar with ***smart pointers*** from standard C++ or have ever used something like that from the also very fascinating Boost libraries, you surely have a grasp about what a ***reference counter*** is, if not, you can imagine that as a simple counter which counts how many references exist to a given variable.

```
x = 5
y = x
print("ID of x: ", id(x))
print("ID of y: ", id(y)) # id(x) = id(y)

###

a = 5
b = 5

print("ID of a: ", id(a))
print("ID of b: ", id(b)) # id(a) = id(b), because an
int is immutable, see further explanation below
```

To further enlighten my former, rather self-referent definition, we should add something to the math behind Python: if you assign a value to a variable, your variable is de facto a referent to that object (that means, it refers to the object). If you assign 2 variables to an object, the internal reference counter will have a value of 2, if one of them goes out of scope, it will be automatically decremented, showing only 1. When the counter hits 0, the object gets destructed automagically.

And here comes the black magic part of Python: if you copy a variable, you can forget in general about the commonly under “copy” referred C++ term, if you would like to have a C++-synonym, ***copy constructibility*** would suffice the most. Because Python by default makes ***shallow copies***, i.e. the internal counter will be incremented—it is quite similar to the C++ `shared_ptr` paradigm. Be sure you understand this paragraph, because it can be the cause for very hard to debug code, if you assume that with copying your objects are independent from each other (for that you need the ***deepcopy*** function from the ***copy package***, then you get really what you have expected, only it is a more resource-demanding operation).

Unfortunately, Python gets here more complicated, because it differentiates between “copying” mutable and immutable data types.

The KISS version: mutable means that modifying the object a referent (= a variable name) refers to modifies the original object, thus every referent of that specific object will register the change. In the case of immutable types, modification of the referred object breaks the reference, it creates a new object, thus only for the variable you modified will the change be visible.

If you are a bit confused, don't worry, I will show you some examples from which you will easily understand, but before that, I list some examples for *mutable* (*list*, *dict*, *tuple*) and *immutable* (*int*, *float*) data types.

```
# immutable references
x = 5
y = x
x = 6
print("y = ", y)

# mutable references
l = [1, 2, 3]
ll = l # shallow copy
l.append(4)
print("ll = ", ll)
```

Passing parameters

Well, Python does not have pointers as good old C/C++, but the before mentioned reference management system enables you to modify an object within a function and use the modified version in the outer scope.

But..., yes there is a but again! Because we should here also differentiate the mutable/immutable cases.

```
def mutable_func(x):
    x += 1
    print("x in inner scope = ", x)

def immutable_func(x):
    x.append(5)
    print("x in inner scope = ", x)

###
```

```
x = 4
print("x in outer scope before function call = ", x) # x
= 4
mutable_func(x) # x = 5
print("x in outer scope after function call = ", x) # x
= 4
```

```
###
```

```
x = [1,2,3,4]
print("x in outer scope before function call = ", x) # x
= [1, 2, 3, 4]
immutable_func(x) # x = [1, 2, 3, 4, 5]
print("x in outer scope after function call = ", x) # x
= [1, 2, 3, 4, 5]
```

For mutable types, the *modification within a function body is visible to the outer scope*, and *only for mutable ones*. *Immutable variables “are reset” to their original values as soon as the function returns*. What is the solution? There exist many, for example you can exploit the superb feature of Python to ***supporting multiple return values***.

Just a last remark: although Python support default arguments, you should use them very carefully for mutable types, because they retain their values between function calls, just see the example below and stay tuned for the next post!

```
def mutable_default_arg(x=[]):
    x.append(5)
    print(x)

mutable_default_arg() # [5]
mutable_default_arg() # [5, 5]
mutable_default_arg() # [5, 5, 5]
```

Complete code can be found here:

Object management

Post can be found here:

<https://medium.com/@SmartLabAI/b5f96895663>
(<https://medium.com/@SmartLabAI/b5f96895663>).

Everything is an object

In [1]:

```
integer = 5
floating_point = 5.0
string = "Five"
l = list()
d = dict()
```


Python under the hood — tips and tricks from a C++-programmers' perspective 02



SmartLab AI [Follow](#)

Oct 3, 2018 · 4 min read

Author: Patrik Reizinger

This is the second part of an article series. You can find the first one here: <https://medium.com/@SmartLabAI/python-under-the-hood-tips-and-tricks-from-a-c-programmers-perspective-01-b5f96895663>

Today, I will dig deeper into Python's inner world by examining the behavior of classes after our introductory journey into the world of Python objects (do not even try to skip the first part of the series, it contains some Easter Eggs and essential information for understanding this post).

For this series is intended for C++-programmers, I assume you are acquainted with the good old OOP paradigm. First, I will discuss the main differences of Python in comparison with C++, followed by pythonic ways of doing OOP, the end is dedicated to some really interesting concepts, so the post is definitely worth reading.

Classical differences

In C++-OOP you are taught to follow the principle of encapsulation, i.e. to make only the for the user needed variables/methods available with the *private/protected* specifiers. In Python, **everything is public** according to the C++-paradigm. Being everything accessible means, that you do **not need getters/setters**.

Although, there is a convention: if using an underscore at the beginning of the member method/variable (e.g. *_method*), it means that the intention of the programmer was, the usage should be restricted within the class.

This is only a notational difference, but it's worth mentioning: in Python, you write **self instead of this** to refer for the actual instance of the class—and the constructor is called `__init__`. It is to be emphasized

that `self` is just a convention, you can call it “kitten” if you’d like, but important is: the first argument of every class method refers to the actual object instance—the name does not really matter, but remember, it keeps your code more readable for others and for you.

You surely like heterogenous collections in C++, but there is nothing like that in Python. That is due to the fact that in Python everything is an object, and containers can hold items of different data types. So the referents are holding references directly to a specific object, i.e. no disambiguation is needed.

Pythonic OOP

Python has also `@abstractmethod` (for that you need a little bit of syntactic sugar, see the example below), `@staticmethod` (they are both used as decorators), as in C++, but some concepts are solved in other way. Well, these are the details we love (or actually, hate) Python for.

```
from abc import ABC, abstractmethod
# class with abstract method
class abstract_class(ABC):
    def __init__(self):
        self.x = 5

    @abstractmethod
    def abs_func(self, text):
        pass
x = abstract_class()

# raises TypeError: Can't instantiate abstract class
abstract_class with abstract methods abs_func

# class with static method
class static_class(object):
    static_var = 5 # so are static variables defined
    def __init__(self):
        self.x = 5

    @staticmethod
    def abs_func():
        print("static method")

x = static_class() # static method
x.static_var
```

In the case of inheritance, you should call, the *constructor of the base class*—nothing special until now. Python has a superb built-in, called ***super***, which is a joker for all base classes, i.e. if you refactor the base

class by renaming it, you should not change the code of the inherited class. Even better, if you change the base class (i.e. you inherit from other one), the inherited class still remains unchanged.

```
class base_class(object):
    def __init__(self):
        print("I am Groot")

class inherited_class(base_class):
    def __init__(self):
        super().__init__()
        print("I am inherited")

x = inherited_class()
# "I am Groot", then "I am inherited"
```

You may ask, does it work for multiple inheritance? The answer is of course yes, but for elaboration we need to discuss another concept first: the **Method Resolution Order (MRO)**. This is the piece of code deep under Python's hood which determines the order of hierarchy regarding inheritance. This concept is responsible for what you would call function override in C++. Because if a method of the instance is invoked, Python searches for one in the MRO—the search always terminates at the first hit (it begins with the actual class and move in the direction of the object class). Disambiguity is something neither programmers nor Python like, so in that case an exception will be raised.

```
class first(object):
    def func(self):
        print("First")

class second(first):
    def func(self):
        print("Second")

class third(second):
    def func(self):
        print("Third")

class fourth_inconsistent(second, third):
    # should MRO be [second, first, third] or [third,
    second, first]?
    #Python cannot decide it raises an exception
    def func(self):
        print("Fourth")

x = third()
x.func() # Third
```

It's time for mentioning the secret weapon of Python's classes: there are dynamically extendable. Yes, you read it correctly!

That means it is possible that object instances (not the class definition) can have new variables, even new methods, isn't it fascinating?

You may wonder: how is it possible? Well, you must pay the price for this, namely the overhead of dynamical class instance management, because it is possible that while extending a specific object it should be reallocated and that may not be a resource-friendly operation. These extensions are registered into the `__dict__` attribute of each instance.

In the case you already know that instance extension won't take place, you can force the *static allocation* of an object instance and your weapon of choice is the `__slots__` attribute. If an attribute is not in the list of `__slots__`, it cannot be created—this statement is only valid for member variables, member functions can be defined if desired, but only within the class body definition. It is rather a useless way of using `__slots__`, but you can include the `__dict__` attribute, then the original functionality will be retained, i.e. you can dynamically extend your class instances.

```
def f():
    print("extension")
class extendable(object):
    pass

x = extendable()
x.f = 5
x.func = f
x.func() # extension
x.__dict__
# {'f': 5, 'func': <function __main__.f>}

####

class not_extendable(object):
    __slots__ = []

    def f(self):
        self.x = 5
        print("defined f")

x = not_extendable()
x.a = f # AttributeError: 'not_extendable' object has no attribute 'a'
x.x = f # AttributeError: 'not_extendable' object has no attribute 'x'
x.__dict__ = f # AttributeError: 'not_extendable' object has no attribute '__dict__'
```

An extended code example is available under:

Pythonic OOP

Post can be found here soon:

<https://medium.com/@SmartLabAI/b52675c7c0af>
(<https://medium.com/@SmartLabAI/b52675c7c0af>)

Abstract classes

In [1]:

```
from abc import ABC, abstractmethod

# class with abstract method
class abstract_class(ABC):
    def init (self):
```


Python under the hood — tips and tricks from a C++-programmers' perspective 03

SmartLab AI [Follow](#)

Feb 4 · 4 min read

Author: Patrik Reizinger

This is the third part of an article series. You can find the first two here:

1. <https://medium.com/@SmartLabAI/python-under-the-hood-tips-and-tricks-from-a-c-programmers-perspective-01-b5f96895663>

2. <https://medium.com/@SmartLabAI/python-under-the-hood-tips-and-tricks-from-a-c-programmers-perspective-02-b52675c7c0af>

You may have wondered while reading my previous posts, why the title of this series consists the phrase tips & tricks. Today it's time to reveal that part, so I will show you some really exciting features I use whenever I can to make my code more human readable and concise.

Disclaimer: after reading this post, you may find yourself an urge to transform your code into nice Pythonic one-liners. What you should remember: I have warned you.

Python proms (*, **)

The star operators, doesn't matter if * or **, are a powerful (and really concise) way for manipulating *lists/tuples/sets (one star)* or *dicts (two stars)*. There are two ways of usage where they will be handy. First, you can unpack containers with the usage of these operators—the concept may seem similar to the dereference operator in C++, but as you recall, Python is not the language for black pointer magic. I.e., if you want to merge two lists, you can do it the following way:

```
# list -the same for sets (use {}) and tuples (use ())
```

```

a = [1, 2, 3]
b = [3, 4, 5]
c = [*a, *b]

print("a = ", a) # [1, 2, 3]
print("b = ", b) # [3, 4, 5]
print("c = ", c) # [1, 2, 3, 3, 4, 5]

```

Beware, the *merged containers* will *remain conform* with the *restrictions* of the used *container types*, e.g. if you merge two sets, the result won't contain duplicates of a given element.

On the other hand, you can use these operators to pass so called non-keyword and keyword arguments with `*` and `**`, respectively. This gives you the power to create functions with a *variable number of arguments*—do you recall that e.g. the `printf` function in C can take any number of arguments (the name for it is called variadic templates, a different concept, but the same result).

If you include in the function definition—I use the conventional names here, but you may pick something different, which I do not advise—****args*** or *****kwargs***, you are actually done, the (keyword) arguments are available for you in the function body.

```

# non-keyword arguments

t = (1, 2, 3)

def tt(*args):

    for arg in args:
        print(arg+1)
        tt(*t) # 2, 3, 4

#####

# keyword arguments

d = {'a' : 1, 'b' : 2}

def dd(**kwargs):
    for key, value in kwargs.items():
        print("Key: ", key, " value: ", value)

dd(**d) # Prints: Key:  a  value:  1

```

```
#           Key:  b  value:  2
```

Ternary operators

Although it is functionally same as in C/C++, but it is worth knowing that Python possesses this feature too. Beware, the syntax is a bit different, basically it is a **one-liner if-else clause**, see the following example.

```
r = 5 if True else 4 # 5
```

List comparisons

The tricks mentioned above were cool, weren't they? And you have not seen the best one, which allows you to **filter lists** as you wish and all of this practically as a one-liner. Although this technique is referred to as list comparison, you can use it also for **sets**.

```
l = list(range(10))
l_even = [x for x in l if x%2 == 0]
s_even_sq = {x**2 for x in l if x%2 == 0}

print("l = ", l) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print("l_even = ", l_even) # [0, 2, 4, 6, 8]
print("s_even_sq = ", s_even_sq) # {0, 64, 4, 36, 16}
```

To be or not to be

At the end comes a delicacy of Pythonic objects, which is definitely good to know if you don't want to get nervous about seemingly perfect code.

You surely remember the part where I explained the difference between *immutable* and *mutable* objects, but a quick recap never hurts, so *immutable* means that the attempt to modify an object results in the allocation of a new one, reasonably *mutable* means that the instance can be modified.

There is a difference even in the case of comparison, for which Python has two ways, the operator `==` as in e.g. C++ and the **is** keyword.

Well, they are slightly different—if you know PHP, it is similar to `==` vs `===`.

`==` checks for *equality*, while `is` does it for *identity*. If you have an *immutable* object, there is *no difference*, because only *one object with a given attribute* exists (e.g. the number 5, the string “Python”), obviously several references to it may exist too.

In the mutable case `==` works as expected with some background from C/C++, e.g. different list with the same elements are equal, but are not always identical, only if both references were created of the same object.

```
a = 1
b = 1
c = a
print("a == b :", a == b) # True
print("a == c :", a == c) # True
print("a is b :", a is b) # True
print("a is c :", a is c) # True

#####

x = [1, 2, 3, 4, 5]
y = [1, 2, 3, 4, 5]
z = x
print("x == y :", x == y) # True
print("x == z :", x == z) # True
print("x is y :", x is y, "<--") # False
print("x is z :", x is z) # True
```

Code is to be found here:

<https://gist.github.com/rpatrik96/5411ef4c86b07c054164309d6f1381ef>

