"Least Astonishment" and the Mutable Default Argument

Asked 12 years, 3 months ago Active 2 days ago Viewed 196k times

2990

957

Anyone tinkering with Python long enough has been bitten (or torn to pieces) by the following issue:

```
def foo(a=[]):
    a.append(5)
    return a
```

Python novices would expect this function to always return a list with only one element: `[5]`. The result is instead very different, and very astonishing (for a novice):

```
>>> foo()
[5]
>>> foo()
[5, 5]
>>> foo()
[5, 5, 5]
>>> foo()
[5, 5, 5, 5]
>>> foo()
```

A manager of mine once had his first encounter with this feature, and called it "a dramatic design flaw" of the language. I replied that the behavior had an underlying explanation, and it is indeed very puzzling and unexpected if you don't understand the internals. However, I was not able to answer (to myself) the following question: what is the reason for binding the default argument at function definition, and not at function execution? I doubt the experienced behavior has a practical use (who really used static variables in C, without breeding bugs?)

Edit:

Baczek made an interesting example. Together with most of your comments and Utaal's in particular, I elaborated further:

```
>>> def a():
...     print("a executed")
...     return []
...
>>>
>>> def b(x=a()):
...     x.append(5)
...     print(x)
...
a executed
>>> b()
[5]
>>> b()
[5, 5]
```

To me, it seems that the design decision was relative to where to put the scope of parameters: inside the function, or "together" with it?

Doing the binding inside the function would mean that `x` is effectively bound to the specified default when the function is called, not defined, something that would present a deep flaw: the `def` line would be "hybrid" in the sense that part of the binding (of the function object) would happen at definition, and part (assignment of default parameters) at function invocation time.

The actual behavior is more consistent: everything of that line gets evaluated when that line is executed, meaning at function definition.

python language-design default-parameters least-astonishment

Share Follow

edited Jul 30 at 23:36

Josh Correia
2,438 1 19 34

asked Jul 15 '09 at 18:00

Stefano Borini
129k 92 280 410

61 Complementary question - Good uses for mutable default arguments – Jonathan Livni Feb 6 '12 at 20:54

7 I have not doubt mutable arguments violate least astonishment principle for an average person, and I have seen beginners stepping there, then heroically replacing mailing lists with mailing tuples. Nevertheless mutable arguments are still in line with Python Zen (Pep 20) and falls into "obvious for Dutch" (understood/exploited by hard core python programmers) clause. The recommended workaround with doc string is the best, yet resistance to doc strings and any (written) docs is not so uncommon nowadays. Personally, I would prefer a decorator (say @fixed_defaults). – Serge Apr 6 '17 at 16:04

5 My argument when I come across this is: "Why do you need to create a function that returns a mutable that could optionally be a mutable you would pass to the function? Either it alters a mutable or creates a new one. Why do you need to do both with one function? And why should the interpreter be rewritten to allow you to do that without adding three lines to your code?" Because we are talking about rewriting the way the interpreter handles function definitions and evocations here. That's a lot to do for a barely necessary use case. – Alan Leuthard Jun 1 '17 at 21:22

21 "Python novices would expect this function to always return a list with only one element: `[5]` ." I'm a Python novice, and I wouldn't expect this, because obviously `foo([1])` will return `[1, 5]` , not `[5]` . What you meant to say is that a novice would expect the function *called with no parameter* will always return `[5]` . – symplectomorphic Jul 6 '17 at 16:08

5 This question asks ***"Why did this [the wrong way] get implemented so?"*** It doesn't ask ***"What's the right way?"***, which is covered by [*Why does using arg=None fix Python's mutable default argument issue?*]*(stackoverflow.com/questions/10676729/…). New users are almost always less interested in the former and much more in the latter, so that's sometimes a very useful link/dupe to cite. – smci Apr 21 '19 at 8:48

31 Answers

Active Oldest Votes

1776

Actually, this is not a design flaw, and it is not because of internals or performance.
It comes simply from the fact that functions in Python are first-class objects, and not only a piece of code.

As soon as you think of it this way, then it completely makes sense: a function is an object being evaluated on its definition; default parameters are kind of "member data" and therefore their state may change from one call to the other - exactly as in any other object.

In any case, Effbot has a very nice explanation of the reasons for this behavior in Default Parameter Values in Python.
I found it very clear, and I really suggest reading it for a better knowledge of how function objects work.

Share Follow

edited Jul 30 at 23:41

Josh Correia
2,438 1 19 34

answered Jul 17 '09 at 21:29

rob
34.4k 2 53 63

89 To anyone reading the above answer, I strongly recommend you take the time to read through the linked Effbot article. As well as all the other useful info, the part on how this language feature can be used for result caching/memoisation is very handy to know! – Cam Jackson Oct 14 '11 at 0:05

115 Even if it's a first-class object, one might still envision a design where the *code* for each default value is stored along with the object and re-evaluated each time the function is called. I'm not saying that would be better, just that functions being first-class objects does not fully preclude it. – gerrit Jan 11 '13 at 10:55

425 Sorry, but anything considered "The biggest WTF in Python" is **most definitely a design flaw**. This is a source of bugs for *everyone* at some point, because no one expects that behavior at first - which means it should not have been designed that way to begin with. I don't care what hoops they had to jump through, they **should** have designed Python so that default arguments are non-static.
– BlueRaja - Danny Pflughoeft Jun 7 '13 at 21:28

253 Whether or not it's a design flaw, your answer seems to imply that this behaviour is somehow necessary, natural and obvious given that functions are first-class objects, and that simply isn't the case. Python has closures. If you replace the default argument with an assignment on the first line of the function, it evaluates the expression each call (potentially using names declared in an enclosing scope). There is no reason at all that it wouldn't be possible or reasonable to have default arguments evaluated each time the function is called in exactly the same way. – Mark Amery Jan 8 '14 at 22:16

40 The design doesn't directly follow from `functions are objects`. In your paradigm, the proposal would be to implement functions' default values as properties rather than attributes. – bukzor May 3 '14 at 20:46

296

Suppose you have the following code

```
fruits = ("apples", "bananas", "loganberries")

def eat(food=fruits):
    ...
```

When I see the declaration of eat, the least astonishing thing is to think that if the first parameter is not given, that it will be equal to the tuple `("apples", "bananas", "loganberries")`

However, suppose later on in the code, I do something like

```
def some_random_function():
    global fruits
    fruits = ("blueberries", "mangos")
```

then if default parameters were bound at function execution rather than function declaration, I would be astonished (in a very bad way) to discover that fruits had been changed. This would be more astonishing IMO than discovering that your `foo` function above was mutating the list.

The real problem lies with mutable variables, and all languages have this problem to some extent. Here's a question: suppose in Java I have the following code:

```
StringBuffer s = new StringBuffer("Hello World!");
Map<StringBuffer,Integer> counts = new HashMap<StringBuffer,Integer>();
counts.put(s, 5);
s.append("!!!!");
System.out.println( counts.get(s) );  // does this work?
```

Now, does my map use the value of the `StringBuffer` key when it was placed into the map, or does it store the key by reference? Either way, someone is astonished; either the person who tried to get the object out of the `Map` using a value identical to the one they put it in with, or the person who can't seem to retrieve their object even though the key they're using is literally the same object that was used to put it into the map (this is actually why Python doesn't allow its mutable built-in data types to be used as dictionary keys).

Your example is a good one of a case where Python newcomers will be surprised and bitten. But I'd argue that if we "fixed" this, then that would only create a different situation where they'd be bitten instead, and that one would be even less intuitive. Moreover, this is always the case when dealing with mutable variables; you always run into cases where someone could intuitively expect one or the opposite behavior depending on what code they're writing.

I personally like Python's current approach: default function arguments are evaluated when the function is defined and that object is always the default. I suppose they could special-case using an empty list, but that kind of special casing would cause even more astonishment, not to mention be backwards incompatible.

Share Follow

edited Jul 30 at 23:44

Josh Correia
2,438 1 19 34

answered Jul 15 '09 at 18:11

Eli Courtwright
170k 63 204 255

47 I think it's a matter of debate. You are acting on a global variable. Any evaluation performed anywhere in your code involving your global variable will now (correctly) refer to ("blueberries", "mangos"). the default parameter could just be like any other case. – Stefano Borini Jul 15 '09 at 18:16

64 Actually, I don't think I agree with your first example. I'm not sure I like the idea of modifying an initializer like that in the first place, but if I did, I'd expect it to behave exactly as you describe — changing the default value to `("blueberries", "mangos")`. – Ben Blank Jul 15 '09 at 18:26

14 The default parameter *is* like any other case. What is unexpected is that the parameter is a global variable, and not a local one. Which in turn is because the code is executed at function definition, not call. Once you get that, and that the same goes for classes, it's perfectly clear. – Lennart Regebro Jul 15 '09 at 18:59

24 I find the example misleading rather than brilliant. If `some_random_function()` appends to `fruits` instead of assigning to it, the behaviour of `eat()` *will* change. So much for the current wonderful design. If you use a default argument that's referenced elsewhere and then modify the reference from outside the function, you are asking for trouble. The real WTF is when people define a fresh default argument (a list literal or a call to a constructor), and *still* get bit. – alexis Oct 9 '14 at 15:37

21 You just explicitly declared `global` and reassigned the tuple - there is absolutely nothing surprising if `eat` works differently after that. – user3467349 Jan 26 '15 at 16:07

270

The relevant part of the documentation:

> **Default parameter values are evaluated from left to right when the function definition is executed.** This means that the expression is evaluated once, when the function is defined, and that the same “pre-computed” value is used for each call. This is especially important to understand when a default parameter is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default value is in effect modified. This is generally not what was intended. A way around this is to use `None` as the default, and explicitly test for it in the body of the function, e.g.:

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

Share Follow

edited Jan 30 '20 at 18:00
Boris
8,210 7 71 69

answered Jul 10 '12 at 14:50
glglgl
83.2k 11 132 205

227 The phrases "this is not generally what was intended" and "a way around this is" smell like they're documenting a design flaw. – bukzor May 3 '14 at 20:53

6 @Matthew: I'm well aware, but it's not worth the pitfall. You'll generally see style guides and linters unconditionally flag mutable default values as wrong for this reason. The explicit way to do the same thing is to stuff an attribute onto the function (`function.data = []`) or better yet, make an object. – bukzor Jun 19 '14 at 3:59

13 @bukzor: Pitfalls need to be noted and documented, which is why this question is good and has received so many upvotes. At the same time, pitfalls don't necessarily need to be removed. How many Python

beginners have passed a list to a function that modified it, and were shocked to see the changes show up in the original variable? Yet mutable object types are wonderful, when you understand how to use them. I guess it just boils down to opinion on this particular pitfall. – Matthew Jun 19 '14 at 17:54

37 The phrase "this is not generally what was intended" means "not what the programmer actually wanted to happen," not "not what Python is supposed to do." – holdenweb Dec 19 '14 at 11:48

12 @holdenweb Wow, I'm mega-late to the party. Given the context, bukzor is completely right: they're documenting behavior/consequence that was not "intended" when they they decided the language should exec the function's definition. Since it's an unintended consequence of their design choice, it's a design flaw. If it were not a design flaw, there'd be no need to even offer "a way around this". – code_dredd Oct 3 '17 at 7:35

128

I know nothing about the Python interpreter inner workings (and I'm not an expert in compilers and interpreters either) so don't blame me if I propose anything unsensible or impossible.

Provided that python objects **are mutable** I think that this should be taken into account when designing the default arguments stuff. When you instantiate a list:

```
a = []
```

you expect to get a **new** list referenced by `a`.

Why should the `a=[]` in

```
def x(a=[]):
```

instantiate a new list on function definition and not on invocation? It's just like you're asking "if the user doesn't provide the argument then *instantiate* a new list and use it as if it was produced by the caller". I think this is ambiguous instead:

```
def x(a=datetime.datetime.now()):
```

user, do you want `a` to default to the datetime corresponding to when you're defining or executing `x`? In this case, as in the previous one, I'll keep the same behaviour as if the default argument "assignment" was the first instruction of the function (`datetime.now()` called on function invocation). On the other hand, if the user wanted the definition-time mapping he could write:

```
b = datetime.datetime.now()
def x(a=b):
```

I know, I know: that's a closure. Alternatively Python might provide a keyword to force definition-time binding:

```
def x(static a=b):
```

Share Follow

edited May 9 '19 at 9:15

Georgy
7,414 7 52 63

answered Jul 15 '09 at 23:21

Utaal
8,174 4 26 37

13 You could do: def x(a=None): And then, if a is None, set a=datetime.datetime.now() – Anon Jul 16 '09 at 0:18

27 Thank you for this. I couldn't really put my finger on why this irks me to no end. You have done it beautifully with a minimum of fuzz and confusion. As someone comming from systems programming in C++ and sometimes naively "translating" language features, this false friend kicked me in the in the soft of the head big time, just like class attributes. I understand why things are this way, but I cannot help but dislike it, no matter what positive might come of it. At least it is so contrary to my experience, that I'll probably (hopefully) never forget it... – AndreasT Apr 22 '11 at 9:33

6 @Andreas once you use Python for long enough, you begin to see how logical it is for Python to interpret things as class attributes the way it does - it is only because of the particular quirks and limitations of languages like C++ (and Java, and C#...) that it makes any sense for contents of the `class {}` block to be interpreted as belonging to the *instances* :) But when classes are first-class objects, obviously the natural thing is for their contents (in memory) to reflect their contents (in code). – Karl Knechtel Jul 22 '11 at 19:55

7 Normative structure is no quirk or limitation in my book. I know it can be clumsy and ugly, but you can call it a "definition" of something. The dynamic languages seem a bit like anarchists to me: Sure everybody is free, but you need structure to get someone to empty the trash and pave the road. Guess I'm old... :) – AndreasT Jul 26 '11 at 8:54

7 The function **definition** is executed at module load time. The function **body** is executed at function call time. The default argument is part of the function definition, not of the function body. (It gets more complicated for nested functions.) – Lutz Prechelt Mar 30 '15 at 11:28

88

Well, the reason is quite simply that bindings are done when code is executed, and the function definition is executed, well... when the functions is defined.

Compare this:

```
class BananaBunch:
    bananas = []

    def addBanana(self, banana):
        self.bananas.append(banana)
```

This code suffers from the exact same unexpected happenstance. bananas is a class attribute, and hence, when you add things to it, it's added to all instances of that class. The reason is exactly the same.

It's just "How It Works", and making it work differently in the function case would probably be complicated, and in the class case likely impossible, or at least slow down object instantiation a

lot, as you would have to keep the class code around and execute it when objects are created.

Yes, it is unexpected. But once the penny drops, it fits in perfectly with how Python works in general. In fact, it's a good teaching aid, and once you understand why this happens, you'll grok python much better.

That said it should feature prominently in any good Python tutorial. Because as you mention, everyone runs into this problem sooner or later.

Share Follow

edited Dec 19 '14 at 22:53

answered Jul 15 '09 at 18:54

Lennart Regebro
153k 40 212 244

How do you define a class attribute that is different for each instance of a class? – Kieveli Jul 15 '09 at 19:04

20 If it's different for each instance it's not a class attribute. Class attributes are attributes on the CLASS. Hence the name. Hence they are the same for all instances. – Lennart Regebro Jul 15 '09 at 19:17

2 How do you define an attribute in a class that is different for each instance of a class? (Re-defined for those who could not determine that a person not familiar with Python's naming convenctions might be asking about normal member variables of a class). – Kieveli Jul 16 '09 at 14:03

2 @Kievieli: You ARE talking about normal member variables of a class. :-) You define instance attributes by saying self.attribute = value in any method. For example __init__(). – Lennart Regebro Jul 16 '09 at 14:14

@Kieveli: Two answers: you can't, because any thing you define at a class level will be a class attribute, and any instance that accesses that attribute will access the same class attribute; you can, /sort of/, by using `property`s -- which are actually class level functions that act like normal attributes but save the attribute in the instance instead of the class (by using `self.attribute = value` as Lennart said). – Ethan Furman Jan 7 '12 at 4:45

Why don't you introspect?

77

I'm *really* surprised no one has performed the insightful introspection offered by Python (`2` and `3` apply) on callables.

Given a simple little function `func` defined as:

```
>>> def func(a = []):
...     a.append(5)
```

When Python encounters it, the first thing it will do is compile it in order to create a `code` object for this function. While this compilation step is done, *Python **evaluates** and then **stores** the default arguments (an empty list `[]` here) in the function object itself*. As the top answer mentioned: the list `a` can now be considered a *member* of the function `func`.

So, let's do some introspection, a before and after to examine how the list gets expanded **inside** the function object. I'm using `Python 3.x` for this, for Python 2 the same applies (use `__defaults__` or `func_defaults` in Python 2; yes, two names for the same thing).

Function Before Execution:

```
>>> def func(a = []):
...     a.append(5)
...
```

After Python executes this definition it will take any default parameters specified (`a = []` here) and cram them in the `__defaults__` attribute for the function object (relevant section: Callables):

```
>>> func.__defaults__
([],)
```

O.k, so an empty list as the single entry in `__defaults__` , just as expected.

Function After Execution:

Let's now execute this function:

```
>>> func()
```

Now, let's see those `__defaults__` again:

```
>>> func.__defaults__
([5],)
```

Astonished? The value inside the object changes! Consecutive calls to the function will now simply append to that embedded `list` object:

```
>>> func(); func(); func()
>>> func.__defaults__
([5, 5, 5, 5],)
```

So, there you have it, the reason why this *'flaw'* happens, is because default arguments are part of the function object. There's nothing weird going on here, it's all just a bit surprising.

The common solution to combat this is to use `None` as the default and then initialize in the function body:

```
def func(a = None):
    # or: a = [] if a is None else a
    if a is None:
        a = []
```

Since the function body is executed anew each time, you always get a fresh new empty list if no argument was passed for `a`.

To further verify that the list in `__defaults__` is the same as that used in the function `func` you can just change your function to return the `id` of the list `a` used inside the function body. Then, compare it to the list in `__defaults__` (position `[0]` in `__defaults__`) and you'll see how these are indeed refering to the same list instance:

```
>>> def func(a = []):
...     a.append(5)
...     return id(a)
>>>
>>> id(func.__defaults__[0]) == func()
True
```

All with the power of introspection!

* To verify that Python evaluates the default arguments during compilation of the function, try executing the following:

```
def bar(a=input('Did you just see me without calling the function?')):
    pass  # use raw_input in Py2
```

as you'll notice, `input()` is called before the process of building the function and binding it to the name `bar` is made.

Share Follow

edited Oct 11 '18 at 16:33

Toby Speight
24.3k 47 58 89

answered Dec 9 '15 at 7:13

Dimitris Fasarakis Hilliard
127k 27 232 229

1 Is `id(...)` needed for that last verification, or would the `is` operator answer the same question? – das-g Mar 9 '16 at 8:09

1 @das-g `is` would do just fine, I just used `id(val)` because I think it might be more intuitive. – Dimitris Fasarakis Hilliard Mar 9 '16 at 8:20

Using `None` as the default severely limits the usefulness of the `__defaults__` introspection, so I don't think that works well as a defense of having `__defaults__` work the way it does. Lazy-evaluation would do more to keep function defaults useful from both sides. – Brilliand Oct 18 '19 at 5:32

64

I used to think that creating the objects at runtime would be the better approach. I'm less certain now, since you do lose some useful features, though it may be worth it regardless simply to prevent newbie confusion. The disadvantages of doing so are:

1. Performance

```
def foo(arg=something_expensive_to_compute()):
    ...
```

If call-time evaluation is used, then the expensive function is called every time your function is used without an argument. You'd either pay an expensive price on each call, or need to manually cache the value externally, polluting your namespace and adding verbosity.

2. Forcing bound parameters

A useful trick is to bind parameters of a lambda to the *current* binding of a variable when the lambda is created. For example:

```
funcs = [ lambda i=i: i for i in range(10)]
```

This returns a list of functions that return 0,1,2,3... respectively. If the behaviour is changed, they will instead bind `i` to the *call-time* value of i, so you would get a list of functions that all returned `9`.

The only way to implement this otherwise would be to create a further closure with the i bound, ie:

```
def make_func(i): return lambda: i
funcs = [make_func(i) for i in range(10)]
```

3. Introspection

Consider the code:

```
def foo(a='test', b=100, c=[]):
   print a,b,c
```

We can get information about the arguments and defaults using the `inspect` module, which

```
>>> inspect.getargspec(foo)
(['a', 'b', 'c'], None, None, ('test', 100, []))
```

This information is very useful for things like document generation, metaprogramming, decorators etc.

Now, suppose the behaviour of defaults could be changed so that this is the equivalent of:

```
_undefined = object()  # sentinel value

def foo(a=_undefined, b=_undefined, c=_undefined)
    if a is _undefined: a='test'
```

```
    if b is _undefined: b=100
    if c is _undefined: c=[]
```

However, we've lost the ability to introspect, and see what the default arguments *are*. Because the objects haven't been constructed, we can't ever get hold of them without actually calling the function. The best we could do is to store off the source code and return that as a string.

Share Follow

edited Jul 16 '09 at 19:13

answered Jul 16 '09 at 10:05

Brian
110k 28 105 109

2 you could achieve introspection also if for each there was a function to create the default argument instead of a value. the inspect module will just call that function. – yairchu Jul 16 '09 at 10:24

@SilentGhost: I'm talking about if the behaviour was changed to recreate it - creating it once is the current behaviour, and why the mutable default problem exists. – Brian Jul 16 '09 at 10:59

2 @yairchu: That assumes the construction is safe to so (ie has no side effects). Introspecting the args shouldn't *do* anything, but evaluating arbitrary code could well end up having an effect. – Brian Jul 16 '09 at 11:02

2 A different language design often just means writing things differently. Your first example could easily be written as: _expensive = expensive(); def foo(arg=_expensive), if you specifically *don't* want it reevaluated. – Glenn Maynard Jul 16 '09 at 18:23

@Glenn - that's what I was referring to with "cache the variable externally" - it is a bit more verbose, and you end up with extra variables in your namespace though. – Brian Jul 16 '09 at 19:04

5 points in defense of Python

63

1. **Simplicity**: The behavior is simple in the following sense: Most people fall into this trap only once, not several times.
2. **Consistency**: Python *always* passes objects, not names. The default parameter is, obviously, part of the function heading (not the function body). It therefore ought to be evaluated at module load time (and only at module load time, unless nested), not at function call time.
3. **Usefulness**: As Frederik Lundh points out in his explanation of "Default Parameter Values in Python", the current behavior can be quite useful for advanced programming. (Use sparingly.)
4. **Sufficient documentation**: In the most basic Python documentation, the tutorial, the issue is loudly announced as an **"Important warning"** in the *first* subsection of Section "More on Defining Functions". The warning even uses boldface, which is rarely applied outside of headings. RTFM: Read the fine manual.
5. **Meta-learning**: Falling into the trap is actually a very helpful moment (at least if you are a reflective learner), because you will subsequently better understand the point "Consistency"

above and that will teach you a great deal about Python.

Share Follow

edited Jan 21 at 12:41

jakubde
23 1 4

answered Mar 30 '15 at 11:18

Lutz Prechelt
30.7k 7 53 82

21 It took me a year to find this behavior is messing up my code on production, ended up removing a complete feature until I bumped into this design flaw by chance. I'm using Django. Since the staging environment did not have many requests, this bug never had any impact on QA. When we went live and received many simultaneous requests - some utility functions started overwriting each other's parameters! Making security holes, bugs and what not. – oriadam Sep 5 '15 at 13:09

12 @oriadam, no offense, but I wonder how you learned Python without running into this before. I am just learning Python now and this possible pitfall is mentioned in the official Python tutorial right alongside the first mention of default arguments. (As mentioned in point 4 of this answer.) I suppose the moral is—rather unsympathetically—to read the **official docs** of the language you use to create production software. – Wildcard Aug 30 '16 at 2:26

Also, it would be surprising (to me) if a function of unknown complexity was called in addition to the function call I am making. – Vatine Sep 2 '16 at 13:26

2 @oriadam, your company needs code review and actual expert coders in the language they write in by the time they have development, staging and production environments. Newbie bugs and bad code habits should not make it to production code – Robin De Schepper Jan 4 at 22:03

55

This behavior is easy explained by:

1. function (class etc.) declaration is executed only once, creating all default value objects
2. everything is passed by reference

So:

```
def x(a=0, b=[], c=[], d=0):
    a = a + 1
    b = b + [1]
    c.append(1)
    print a, b, c
```

1. `a` doesn't change - every assignment call creates new int object - new object is printed
2. `b` doesn't change - new array is build from default value and printed
3. `c` changes - operation is performed on same object - and it is printed

Share Follow

edited Oct 24 '17 at 6:34

Dimitris Fasarakis Hilliard
127k 27 232 229

answered Jul 15 '09 at 19:15

ymv
2,047 13 20

(Actually, **add** is a bad example, but integers being immutable still is my main point.) – Anon Jul 15 '09 at 23:54

Realized it to my chagrin after checking to see that, with b set to [], b.__add__([1]) returns [1] but also leaves b still [] even though lists are mutable. My bad. – Anon Jul 16 '09 at 0:03

@ANon: there is `__iadd__` , but it doesn't work with int. Of course. :-) – Veky May 8 '14 at 13:16

37

1) The so-called problem of "Mutable Default Argument" is in general a special example demonstrating that:
"All functions with this problem **suffer also from similar side effect problem on the actual parameter**,"
That is against the rules of functional programming, usually undesiderable and should be fixed both together.

Example:

```
def foo(a=[]):                 # the same problematic function
    a.append(5)
    return a

>>> somevar = [1, 2]           # an example without a default parameter
>>> foo(somevar)
[1, 2, 5]
>>> somevar
[1, 2, 5]                      # usually expected [1, 2]
```

Solution: a **copy**
An absolutely safe solution is to **`copy`** or **`deepcopy`** the input object first and then to do whatever with the copy.

```
def foo(a=[]):
    a = a[:]     # a copy
    a.append(5)
    return a     # or everything safe by one line: "return a + [5]"
```

Many builtin mutable types have a copy method like `some_dict.copy()` or `some_set.copy()` or can be copied easy like `somelist[:]` or `list(some_list)`. Every object can be also copied by `copy.copy(any_object)` or more thorough by `copy.deepcopy()` (the latter useful if the mutable object is composed from mutable objects). Some objects are fundamentally based on side effects like "file" object and can not be meaningfully reproduced by copy. copying

Example problem for a similar SO question

```
class Test(object):            # the original problematic class
  def __init__(self, var1=[]):
    self._var1 = var1

somevar = [1, 2]               # an example without a default parameter
```

```
t1 = Test(somevar)
t2 = Test(somevar)
t1._var1.append([1])
print somevar                   # [1, 2, [1]] but usually expected [1, 2]
print t2._var1                  # [1, 2, [1]] but usually expected [1, 2]
```

It shouldn't be neither saved in any *public* attribute of an instance returned by this function. (Assuming that *private* attributes of instance should not be modified from outside of this class or subclasses by convention. i.e. `_var1` is a private attribute)

Conclusion:
Input parameters objects shouldn't be modified in place (mutated) nor they should not be binded into an object returned by the function. (If we prefere programming without side effects which is strongly recommended. see Wiki about "side effect" (The first two paragraphs are relevent in this context.) .)

2)
Only if the side effect on the actual parameter is required but unwanted on the default parameter then the useful solution is `def ...(var1=None): if var1 is None: var1 = []` More..

3) In some cases is the mutable behavior of default parameters useful.

Share Follow

edited May 23 '17 at 11:47
Community Bot
1 1

answered Nov 22 '12 at 18:09

hynekcer
13.1k 4 56 88

6 I hope you're aware that Python is *not* a functional programming language. – Veky May 8 '14 at 13:18

7 Yes, Python is a multi-paragigm language with some functional features. ("Don't make every problem look like a nail just because you have a hammer.") Many of them are in Python best practicies. Python has an interesting HOWTO Functional Programming Other features are closures and currying, not mentioned here. – hynekcer May 8 '14 at 15:54

1 I'd also add, at this late stage, that Python's assignment semantics have been designed explicitly to avoid data copying where necessary, so the creation of copies (and especially of deep copies) will affect both run-time and memory usage adversely. They should therefore be used only when necessary, but newcomers often have difficulty understanding when that is. – holdenweb Jan 16 '18 at 14:27

1 @holdenweb I agree. A temporary copy is the most usual way and sometimes the only possible way how to protect the original mutable data from an extraneous function that modifies them potentially. Fortunately a function that unreasonably modifies data is considered a bug and therefore uncommon. – hynekcer Jan 17 '18 at 21:41

I agree with this answer. And I don't understand why the `def f( a = None )` construct is recommended when you really mean something else. Copying is ok, because you shouldn't mutate arguments. And when you do `if a is None: a = [1, 2, 3]` , you do copy the list anyway. – koddo Feb 16 '18 at 16:15

What you're asking is why this:

36

```
def func(a=[], b = 2):
    pass
```

isn't internally equivalent to this:

```
def func(a=None, b = None):
    a_default = lambda: []
    b_default = lambda: 2
    def actual_func(a=None, b=None):
        if a is None: a = a_default()
        if b is None: b = b_default()
    return actual_func
func = func()
```

except for the case of explicitly calling func(None, None), which we'll ignore.

In other words, instead of evaluating default parameters, why not store each of them, and evaluate them when the function is called?

One answer is probably right there--it would effectively turn every function with default parameters into a closure. Even if it's all hidden away in the interpreter and not a full-blown closure, the data's got to be stored somewhere. It'd be slower and use more memory.

Share Follow

answered Jul 15 '09 at 20:18

Glenn Maynard
51.9k 9 111 128

9 It wouldn't need to be a closure - a better way to think of it would simply to make the bytecode creating defaults the first line of code - after all you're compiling the body at that point anyway - there's no real difference between code in the arguments and code in the body. – Brian Jul 16 '09 at 9:39

11 True, but it would still slow Python down, and it would actually be quite surprising, unless you do the same for class definitions, which would make it stupidly slow as you would have to re-run the whole class definition each time you instantiate a class. As mentioned, the fix would be more surprising than the problem. – Lennart Regebro Jul 16 '09 at 11:49

Agreed with Lennart. As Guido is fond of saying, for every language feature or standard library, there's *someone* out there using it. – Jason Baker Jul 16 '09 at 13:21

8 Changing it now would be insanity--we're just exploring why it is the way it is. If it did late default evaluation to begin with, it wouldn't necessarily be surprising. It's definitely true that such a core a difference of parsing would have sweeping, and probably many obscure, effects on the language as a whole. – Glenn Maynard Jul 16 '09 at 18:10

30

This actually has nothing to do with default values, other than that it often comes up as an unexpected behaviour when you write functions with mutable default values.

```
>>> def foo(a):
    a.append(5)
```

```
    print a

>>> a  = [5]
>>> foo(a)
[5, 5]
>>> foo(a)
[5, 5, 5]
>>> foo(a)
[5, 5, 5, 5]
>>> foo(a)
[5, 5, 5, 5, 5]
```

No default values in sight in this code, but you get exactly the same problem.

The problem is that `foo` is *modifying* a mutable variable passed in from the caller, when the caller doesn't expect this. Code like this would be fine if the function was called something like `append_5`; then the caller would be calling the function in order to modify the value they pass in, and the behaviour would be expected. But such a function would be very unlikely to take a default argument, and probably wouldn't return the list (since the caller already has a reference to that list; the one it just passed in).

Your original `foo`, with a default argument, shouldn't be modifying `a` whether it was explicitly passed in or got the default value. Your code should leave mutable arguments alone unless it is clear from the context/name/documentation that the arguments are supposed to be modified. Using mutable values passed in as arguments as local temporaries is an extremely bad idea, whether we're in Python or not and whether there are default arguments involved or not.

If you need to destructively manipulate a local temporary in the course of computing something, and you need to start your manipulation from an argument value, you need to make a copy.

Share Follow

answered May 23 '11 at 4:24

Ben

59.2k 19 118 157

10 Although related, I think this is distinct behaviour (as we expect `append` to change `a` "in-place"). That a **default mutable is not re-instantiated on each call** is the "unexpected" bit... at least for me. :) – Andy Hayden Aug 24 '12 at 12:27

2 @AndyHayden if the function is *expected* to modify the argument, why would it make sense to have a default? – Mark Ransom Oct 17 '17 at 13:31

1 @AndyHayden I left my own answer here with an expansion of that sentiment. Let me know what you think. I might add your example of `cache={}` into it for completeness. – Mark Ransom Oct 17 '17 at 18:02

1 @AndyHayden The point of my answer is that if you are ever astonished by accidentally mutating the default value of an argument, then you have another bug, which is that your code can accidentally mutate a caller's value when the default *wasn't* used. And note that using `None` and assigning the real default if the arg is `None` *does not resolve that problem* (I consider it an anti pattern for that reason). If you fix the other bug by avoiding mutating argument values whether or not they have defaults then you'll never notice or care about this "astonishing" behavior. – Ben Oct 17 '17 at 21:44

1 @AndyHayden And if you're putting `self.foo = foo.copy()` in there anyway, what harm is it if the default value for `foo` is `[]` ? It's the **copy** that protects you from mutable argument woes, not setting the default to `None` when you really want a default of `[]` . Sure you *could* write `if foo is None: self.foo = []; else: self.foo = foo.copy()` , but why when you could replace 4 lines with a single line (which is one of the 4 you need anyway), *and* have the real value of the default argument be clearer in the function definition? – Ben Oct 18 '17 at 0:51

27

Already busy topic, but from what I read here, the following helped me realizing how it's working internally:

```
def bar(a=[]):
     print id(a)
     a = a + [1]
     print id(a)
     return a

>>> bar()
4484370232
4484524224
[1]
>>> bar()
4484370232
4484524152
[1]
>>> bar()
4484370232 # Never change, this is 'class property' of the function
4484523720 # Always a new object
[1]
>>> id(bar.func_defaults[0])
4484370232
```

Share Follow

answered Mar 26 '15 at 23:14

Stéphane

1,857 20 27

3 actually this might be a bit confusing for newcomers as `a = a + [1]` overloads `a` ... consider changing it to `b = a + [1] ; print id(b)` and add a line `a.append(2)` . That will make it more obvious that `+` on two lists always creates a new list (assigned to `b`), while a modified `a` can still have the same `id(a)` . – Jörn Hees Apr 8 '17 at 13:47

26

It's a performance optimization. As a result of this functionality, which of these two function calls do you think is faster?

```
def print_tuple(some_tuple=(1,2,3)):
    print some_tuple

print_tuple()        #1
print_tuple((1,2,3)) #2
```

I'll give you a hint. Here's the disassembly (see http://docs.python.org/library/dis.html):

1

```
 0 LOAD_GLOBAL              0 (print_tuple)
 3 CALL_FUNCTION            0
 6 POP_TOP
 7 LOAD_CONST               0 (None)
10 RETURN_VALUE
```

2

```
 0 LOAD_GLOBAL              0 (print_tuple)
 3 LOAD_CONST               4 ((1, 2, 3))
 6 CALL_FUNCTION            1
 9 POP_TOP
10 LOAD_CONST               0 (None)
13 RETURN_VALUE
```

> I doubt the experienced behavior has a practical use (who really used static variables in C, without breeding bugs ?)

As you can see, there *is* a performance benefit when using immutable default arguments. This can make a difference if it's a frequently called function or the default argument takes a long time to construct. Also, bear in mind that Python isn't C. In C you have constants that are pretty much free. In Python you don't have this benefit.

Share Follow

edited Apr 2 '13 at 21:52

answered Jul 15 '09 at 23:18

Jason Baker

176k 125 356 503

25

Python: The Mutable Default Argument

Default arguments get evaluated at the time the function is compiled into a function object. When used by the function, multiple times by that function, they are and remain the same object.

When they are mutable, when mutated (for example, by adding an element to it) they remain mutated on consecutive calls.

They stay mutated because they are the same object each time.

Equivalent code:

Since the list is bound to the function when the function object is compiled and instantiated, this:

```
def foo(mutable_default_argument=[]): # make a list the default argument
    """function that uses a list"""
```

is almost exactly equivalent to this:

```
_a_list = [] # create a list in the globals

def foo(mutable_default_argument=_a_list): # make it the default argument
    """function that uses a list"""

del _a_list # remove globals name binding
```

Demonstration

Here's a demonstration - you can verify that they are the same object each time they are referenced by

- seeing that the list is created before the function has finished compiling to a function object,
- observing that the id is the same each time the list is referenced,
- observing that the list stays changed when the function that uses it is called a second time,
- observing the order in which the output is printed from the source (which I conveniently numbered for you):

example.py

```
print('1. Global scope being evaluated')

def create_list():
    '''noisily create a list for usage as a kwarg'''
    l = []
    print('3. list being created and returned, id: ' + str(id(l)))
    return l

print('2. example_function about to be compiled to an object')

def example_function(default_kwarg1=create_list()):
    print('appending "a" in default default_kwarg1')
    default_kwarg1.append("a")
    print('list with id: ' + str(id(default_kwarg1)) +
          ' - is now: ' + repr(default_kwarg1))

print('4. example_function compiled: ' + repr(example_function))


if __name__ == '__main__':
    print('5. calling example_function twice!:')
    example_function()
    example_function()
```

and running it with `python example.py`:

```
1. Global scope being evaluated
2. example_function about to be compiled to an object
```

```
3. list being created and returned, id: 140502758808032
4. example_function compiled: <function example_function at 0x7fc9590905f0>
5. calling example_function twice!:
appending "a" in default default_kwarg1
list with id: 140502758808032 - is now: ['a']
appending "a" in default default_kwarg1
list with id: 140502758808032 - is now: ['a', 'a']
```

Does this violate the principle of "Least Astonishment"?

This order of execution is frequently confusing to new users of Python. If you understand the Python execution model, then it becomes quite expected.

The usual instruction to new Python users:

But this is why the usual instruction to new users is to create their default arguments like this instead:

```
def example_function_2(default_kwarg=None):
    if default_kwarg is None:
        default_kwarg = []
```

This uses the None singleton as a sentinel object to tell the function whether or not we've gotten an argument other than the default. If we get no argument, then we actually want to use a new empty list, `[]`, as the default.

As the tutorial section on control flow says:

> If you don't want the default to be shared between subsequent calls, you can write the function like this instead:
>
> ```
> def f(a, L=None):
>     if L is None:
>         L = []
>     L.append(a)
>     return L
> ```

Share Follow

edited Dec 23 '17 at 21:18

answered May 1 '16 at 16:20

Aaron Hall ♦

312k 77 380 316

25

The shortest answer would probably be "definition is execution", therefore the whole argument makes no strict sense. As a more contrived example, you may cite this:

```
def a(): return []
```

```
def b(x=a()):
    print x
```

Hopefully it's enough to show that not executing the default argument expressions at the execution time of the `def` statement isn't easy or doesn't make sense, or both.

I agree it's a gotcha when you try to use default constructors, though.

Share Follow

edited May 20 '18 at 23:22

Ashraf.Shk786
580 1 9 23

answered Jul 16 '09 at 12:19

Baczek
1,009 1 12 21

22

This behavior is not surprising if you take the following into consideration:

1. The behavior of read-only class attributes upon assignment attempts, and that
2. Functions are objects (explained well in the accepted answer).

The role of **(2)** has been covered extensively in this thread. **(1)** is likely the astonishment causing factor, as this behavior is not "intuitive" when coming from other languages.

(1) is described in the Python tutorial on classes. In an attempt to assign a value to a read-only class attribute:

> ...all variables found outside of the innermost scope are read-only (***an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged***).

Look back to the original example and consider the above points:

```
def foo(a=[]):
    a.append(5)
    return a
```

Here `foo` is an object and `a` is an attribute of `foo` (available at `foo.func_defs[0]`). Since `a` is a list, `a` is mutable and is thus a read-write attribute of `foo`. It is initialized to the empty list as specified by the signature when the function is instantiated, and is available for reading and writing as long as the function object exists.

Calling `foo` without overriding a default uses that default's value from `foo.func_defs`. In this case, `foo.func_defs[0]` is used for `a` within function object's code scope. Changes to `a` change `foo.func_defs[0]`, which is part of the `foo` object and persists between execution of the code in `foo`.

Now, compare this to the example from the documentation on emulating the default argument behavior of other languages, such that the function signature defaults are used every time the

function is executed:

```
def foo(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

Taking **(1)** and **(2)** into account, one can see why this accomplishes the desired behavior:

- When the `foo` function object is instantiated, `foo.func_defs[0]` is set to `None`, an immutable object.
- When the function is executed with defaults (with no parameter specified for `L` in the function call), `foo.func_defs[0]` (`None`) is available in the local scope as `L`.
- Upon `L = []`, the assignment cannot succeed at `foo.func_defs[0]`, because that attribute is read-only.
- Per **(1)**, ***a new local variable also named `L` is created in the local scope*** and used for the remainder of the function call. `foo.func_defs[0]` thus remains unchanged for future invocations of `foo`.

Share Follow

edited Aug 27 '20 at 20:44

HoldOffHunger
12.9k 8 63 109

answered Apr 24 '12 at 19:43

Dmitry Minkovsky
31.6k 24 100 141

A simple workaround using None

21

```
>>> def bar(b, data=None):
...     data = data or []
...     data.append(b)
...     return data
...
>>> bar(3)
[3]
>>> bar(3)
[3]
>>> bar(3)
[3]
>>> bar(3, [34])
[34, 3]
>>> bar(3, [34])
[34, 3]
```

Share Follow

answered Feb 28 '13 at 11:10

hugo24
1,033 13 21

I am going to demonstrate an alternative structure to pass a default list value to a function (it

works equally well with dictionaries).

19

As others have extensively commented, the list parameter is bound to the function when it is defined as opposed to when it is executed. Because lists and dictionaries are mutable, any alteration to this parameter will affect other calls to this function. As a result, subsequent calls to the function will receive this shared list which may have been altered by any other calls to the function. Worse yet, two parameters are using this function's shared parameter at the same time oblivious to the changes made by the other.

Wrong Method (probably...):

```
def foo(list_arg=[5]):
    return list_arg

a = foo()
a.append(6)
>>> a
[5, 6]

b = foo()
b.append(7)
# The value of 6 appended to variable 'a' is now part of the list held by 'b'.
>>> b
[5, 6, 7]

# Although 'a' is expecting to receive 6 (the last element it appended to the
list),
# it actually receives the last element appended to the shared list.
# It thus receives the value 7 previously appended by 'b'.
>>> a.pop()
7
```

You can verify that they are one and the same object by using `id`:

```
>>> id(a)
5347866528

>>> id(b)
5347866528
```

Per Brett Slatkin's "Effective Python: 59 Specific Ways to Write Better Python", *Item 20: Use `None` and Docstrings to specify dynamic default arguments* (p. 48)

> The convention for achieving the desired result in Python is to provide a default value of `None` and to document the actual behaviour in the docstring.

This implementation ensures that each call to the function either receives the default list or else the list passed to the function.

Preferred Method:

```
def foo(list_arg=None):
   """
   :param list_arg:  A list of input values.
                     If none provided, used a list with a default value of 5.
   """
   if not list_arg:
       list_arg = [5]
   return list_arg

a = foo()
a.append(6)
>>> a
[5, 6]

b = foo()
b.append(7)
>>> b
[5, 7]

c = foo([10])
c.append(11)
>>> c
[10, 11]
```

There may be legitimate use cases for the 'Wrong Method' whereby the programmer intended the default list parameter to be shared, but this is more likely the exception than the rule.

Share Follow

edited Sep 12 '15 at 20:41

answered Sep 12 '15 at 6:00

Alexander

91.6k 24 174 178

17

The solutions here are:

1. Use `None` as your default value (or a nonce `object`), and switch on that to create your values at runtime; or
2. Use a `lambda` as your default parameter, and call it within a try block to get the default value (this is the sort of thing that lambda abstraction is for).

The second option is nice because users of the function can pass in a callable, which may be already existing (such as a `type`)

Share Follow

edited Jun 30 '13 at 16:20

answered Mar 20 '12 at 17:22

Marcin

45.5k 17 114 193

16

You can get round this by replacing the object (and therefore the tie with the scope):

```
def foo(a=[]):
    a = list(a)
```

```
    a.append(5)
    return a
```

Ugly, but it works.

Share Follow

answered Jan 15 '13 at 11:02

joedborg
15.6k 30 76 114

3 This is a nice solution in cases where you're using automatic documentation generation software to document the types of arguments expected by the function. Putting a=None and then setting a to [] if a is None doesn't help a reader understand at a glance what is expected. – Michael Scott Cuthbert Jan 20 '13 at 6:55

Cool idea: rebinding that name guarantees it can never be modified. I really like that. – holdenweb Jan 16 '18 at 14:29

This is exactly the way to do it. Python doesn't make a copy of the parameter, so it's up to you to make the copy explicitly. Once you have a copy, it's yours to modify as you please without any unexpected side effects. – Mark Ransom May 25 '18 at 16:56

16

When we do this:

```
def foo(a=[]):
    ...
```

... we assign the argument `a` to an *unnamed* list, if the caller does not pass the value of a.

To make things simpler for this discussion, let's temporarily give the unnamed list a name. How about `pavlo` ?

```
def foo(a=pavlo):
   ...
```

At any time, if the caller doesn't tell us what `a` is, we reuse `pavlo` .

If `pavlo` is mutable (modifiable), and `foo` ends up modifying it, an effect we notice the next time `foo` is called without specifying `a` .

So this is what you see (Remember, `pavlo` is initialized to []):

```
>>> foo()
[5]
```

Now, `pavlo` is [5].

Calling `foo()` again modifies `pavlo` again:

```
>>> foo()
[5, 5]
```

Specifying `a` when calling `foo()` ensures `pavlo` is not touched.

```
>>> ivan = [1, 2, 3, 4]
>>> foo(a=ivan)
[1, 2, 3, 4, 5]
>>> ivan
[1, 2, 3, 4, 5]
```

So, `pavlo` is still `[5, 5]`.

```
>>> foo()
[5, 5, 5]
```

Share Follow

answered Sep 11 '14 at 22:05

Saish

491 3 11

16

I sometimes exploit this behavior as an alternative to the following pattern:

```
singleton = None

def use_singleton():
    global singleton

    if singleton is None:
        singleton = _make_singleton()

    return singleton.use_me()
```

If `singleton` is only used by `use_singleton`, I like the following pattern as a replacement:

```
# _make_singleton() is called only once when the def is executed
def use_singleton(singleton=_make_singleton()):
    return singleton.use_me()
```

I've used this for instantiating client classes that access external resources, and also for creating dicts or lists for memoization.

Since I don't think this pattern is well known, I do put a short comment in to guard against future misunderstandings.

Share Follow

edited Feb 6 '15 at 12:56

answered Feb 5 '15 at 21:44


bgreen-litl
394 3 7

2 I prefer to add a decorator for memoization, and put the memoization cache onto the function object itself. – Stefano Borini Feb 6 '15 at 7:34

This example doesn't replace the more complex pattern you show, because you call `_make_singleton` at def time in the default argument example, but at call time in the global example. A true substitution would use some sort of mutable box for the default argument value, but the addition of the argument makes an opportunity to pass alternate values. – Yann Vernier Nov 19 '17 at 8:29

15

It may be true that:

1. Someone is using every language/library feature, and
2. Switching the behavior here would be ill-advised, but

it is entirely consistent to hold to both of the features above and still make another point:

3. It is a confusing feature and it is unfortunate in Python.

The other answers, or at least some of them either make points 1 and 2 but not 3, or make point 3 and downplay points 1 and 2. **But all three are true.**

It may be true that switching horses in midstream here would be asking for significant breakage, and that there could be more problems created by changing Python to intuitively handle Stefano's opening snippet. And it may be true that someone who knew Python internals well could explain a minefield of consequences. *However,*

The existing behavior is not Pythonic, and Python is successful because very little about the language violates the principle of least astonishment anywhere *near* this badly. It is a real problem, whether or not it would be wise to uproot it. It is a design flaw. If you understand the language much better by trying to trace out the behavior, I can say that C++ does all of this and more; you learn a lot by navigating, for instance, subtle pointer errors. But this is not Pythonic: people who care about Python enough to persevere in the face of this behavior are people who are drawn to the language because Python has far fewer surprises than other language. Dabblers and the curious become Pythonistas when they are astonished at how little time it takes to get something working--not because of a design fl--I mean, hidden logic puzzle--that cuts against the intuitions of programmers who are drawn to Python because it **Just Works**.

Share Follow

answered Jul 16 '09 at 19:17


Christos Hayward
5,437 15 53 105

7 -1 Although a defensible perspective, this not an answer, **and** I disagree with it. Too many special exceptions beget their own corner cases. – Marcin Jul 7 '12 at 19:24

5

So then, it is "amazingly ignorant" to say that in Python it would make more sense for a default argument of [] to remain [] every time the function is called? – Christos Hayward Dec 27 '12 at 22:09

4 And it is ignorant to consider as an unfortunate idiom setting a default argument to None, and then in the body of the body of the function setting if argument == None: argument = []? Is it ignorant to consider this idiom unfortunate as often people want what a naive newcomer would expect, that if you assign f(argument = []), argument will automatically default to a value of []? – Christos Hayward Dec 27 '12 at 22:11

4 But in Python, part of the spirit of the language is that you don't have to take too many deep dives; array.sort() works, and works regardless of how little you understand about sorting, big-O, and constants. The beauty of Python in the array sorting mechanism, to give one of innumerable examples, is that you are not required to take a deep dive into internals. And to say it differently, the beauty of Python is that one is not ordinarily required to take a deep dive into implementation to get something that Just Works. And there is a workaround (...if argument == None: argument = []), FAIL. – Christos Hayward Dec 27 '12 at 22:41

3 As a standalone, the statement `x=[]` means "create an empty list object, and bind the name 'x' to it." So, in `def f(x=[])`, an empty list is also created. It doesn't always get bound to x, so instead it gets bound to the default surrogate. Later when f() is called, the default is hauled out and bound to x. Since it was the empty list itself that was squirreled away, that same list is the only thing available to bind to x, whether anything has been stuck inside it or not. How could it be otherwise? – Jerry B Oct 5 '13 at 6:18

9

This "bug" gave me a lot of overtime work hours! But I'm beginning to see a potential use of it (but I would have liked it to be at the execution time, still)

I'm gonna give you what I see as a useful example.

```
def example(errors=[]):
    # statements
    # Something went wrong
    mistake = True
    if mistake:
        tryToFixIt(errors)
        # Didn't work.. let's try again
        tryToFixItAnotherway(errors)
        # This time it worked
    return errors

def tryToFixIt(err):
    err.append('Attempt to fix it')

def tryToFixItAnotherway(err):
    err.append('Attempt to fix it by another way')

def main():
    for item in range(2):
        errors = example()
    print '\n'.join(errors)

main()
```

prints the following

```
Attempt to fix it
Attempt to fix it by another way
```

```
Attempt to fix it
Attempt to fix it by another way
```

Share Follow

edited Jul 23 '13 at 10:07

answered Jul 22 '13 at 7:35

Norfeldt

5,526 15 76 121

1 Your example doesn't seem very realistic. Why would you pass `errors` as a parameter rather than starting from scratch every time? – Mark Ransom May 25 at 23:39

9

This is not a design flaw. Anyone who trips over this is doing something wrong.

There are 3 cases I see where you might run into this problem:

1. You intend to modify the argument as a side effect of the function. In this case it *never makes sense* to have a default argument. The only exception is when you're abusing the argument list to have function attributes, e.g. `cache={}` , and you wouldn't be expected to call the function with an actual argument at all.
2. You intend to leave the argument unmodified, but you accidentally *did* modify it. That's a bug, fix it.
3. You intend to modify the argument for use inside the function, but didn't expect the modification to be viewable outside of the function. In that case you need to make a *copy* of the argument, whether it was the default or not! Python is not a call-by-value language so it doesn't make the copy for you, you need to be explicit about it.

The example in the question could fall into category 1 or 3. It's odd that it both modifies the passed list and returns it; you should pick one or the other.

Share Follow

edited Oct 17 '17 at 18:04

answered Oct 17 '17 at 17:38

Mark Ransom

277k 39 358 591

"Doing something wrong" is the diagnosis. That said, I think there are times were =None pattern is useful, but generally you don't want to modify if passed a mutable in that case (2). The `cache={}` pattern is really an interview-only solution, in real code you probably want `@lru_cache` ! – Andy Hayden Oct 17 '17 at 18:19

1 Totally disagree, its absolutely a design flaw in many cases and not the programmer doing something wong – aCuria May 25 at 10:44

@aCuria so you have a case 4 that's different from the 3 I presented? I'd love to hear about it, please tell me more. Python's behavior may not make sense in this circumstance, but it's very useful in other places and to change it would be a disaster. – Mark Ransom May 25 at 13:04

I have never run into the problem of the OP even though it is so highly upvoted, because having a default argument be mutable is weird design to me. – qwr May 25 at 19:56

@MarkRansom If we take it as given that side effects are OK, there's nothing wrong with modifying a default argument as part of a side-effect-ful function. Let's say you have a function that does *something* to a list and returns the list. We want to ensure that the function always returns a list. Then having an empty (or non-empty) list as a default makes perfect sense. The language is violating a large proportion of new Python programmers' expectations. Why are they wrong and the language right? Would you be making the opposite argument if the language had the opposite behavior? – Clement Cherlin Jun 3 at 14:05

8

Just change the function to be:

```
def notastonishinganymore(a = []):
    '''The name is just a joke :)'''
    a = a[:]
    a.append(5)
    return a
```

Share Follow

edited Sep 6 '18 at 21:29

Prune

74.2k 14 51 74

answered May 25 '15 at 23:04

ytpillai

3,258 1 25 43

8

Every other answer explains why this is actually a nice and desired behavior, or why you shouldn't be needing this anyway. Mine is for those stubborn ones who want to exercise their right to bend the language to their will, not the other way around.

We will "fix" this behavior with a decorator that will copy the default value instead of reusing the same instance for each positional argument left at its default value.

```
import inspect
from copy import copy

def sanify(function):
    def wrapper(*a, **kw):
        # store the default values
        defaults = inspect.getargspec(function).defaults # for python2
        # construct a new argument list
        new_args = []
        for i, arg in enumerate(defaults):
            # allow passing positional arguments
            if i in range(len(a)):
                new_args.append(a[i])
            else:
                # copy the value
                new_args.append(copy(arg))
        return function(*new_args, **kw)
    return wrapper
```

Now let's redefine our function using this decorator:

```
@sanify
def foo(a=[]):
    a.append(5)
```

```
    return a

foo() # '[5]'
foo() # '[5]' -- as desired
```

This is particularly neat for functions that take multiple arguments. Compare:

```
# the 'correct' approach
def bar(a=None, b=None, c=None):
    if a is None:
        a = []
    if b is None:
        b = []
    if c is None:
        c = []
    # finally do the actual work
```

with

```
# the nasty decorator hack
@sanify
def bar(a=[], b=[], c=[]):
    # wow, works right out of the box!
```

It's important to note that the above solution breaks if you try to use keyword args, like so:

```
foo(a=[4])
```

The decorator could be adjusted to allow for that, but we leave this as an exercise for the reader ;)

Share Follow

answered Jan 3 '19 at 7:38

Przemek D

584 5 24

7

TLDR: Define-time defaults are consistent and strictly more expressive.

Defining a function affects two scopes: the defining scope *containing* the function, and the execution scope *contained by* the function. While it is pretty clear how blocks map to scopes, the question is where `def <name>(<args=defaults>):` belongs to:

```
...                          # defining scope
def name(parameter=default):  # ???
    ...                      # execution scope
```

The `def name` part **must** evaluate in the defining scope - we want `name` to be available there, after all. Evaluating the function only inside itself would make it inaccessible.

Since `parameter` is a constant name, we can "evaluate" it at the same time as `def name`. This also has the advantage it produces the function with a known signature as `name(parameter=...):`, instead of a bare `name(...):`.

Now, when to evaluate `default`?

Consistency already says "at definition": everything else of `def <name>(<args=defaults>):` is best evaluated at definition as well. Delaying parts of it would be the astonishing choice.

The two choices are not equivalent, either: If `default` is evaluated at definition time, it *can still* affect execution time. If `default` is evaluated at execution time, it *cannot* affect definition time. Choosing "at definition" allows expressing both cases, while choosing "at execution" can express only one:

```
def name(parameter=defined):  # set default at definition time
    ...

def name(parameter=default):     # delay default until execution time
    parameter = default if parameter is None else parameter
    ...
```

Share Follow

edited Aug 8 '19 at 7:39

answered Dec 15 '18 at 12:09

MisterMiyagi
31.7k 5 74 89

1 "Consistency already says "at definition": everything else of `def <name>(<args=defaults>):` is best evaluated at definition as well." I don't think the conclusion follows from the premise. Just because two things are on the same line doesn't mean they should be evaluated in the same scope. `default` is a different thing than the rest of the line: it's an expression. Evaluating an expression is a very different process from defining a function. – LarsH Sep 23 '19 at 14:38

@LarsH Function definitions are *are* evaluated in Python. Whether that is from a statement (`def`) or expression (`lambda`) does not change that creating a function means evaluation -- especially of its signature. And defaults are part of a function's signature. That does not mean defaults *have* to be evaluated immediately -- type hints may not, for example. But it certainly suggests they should unless there is a good reason not to. – MisterMiyagi Sep 23 '19 at 15:22

1 OK, creating a function means evaluation in some sense, but obviously not in the sense that every expression within it is evaluated at the time of definition. Most aren't. It's not clear to me in what sense the signature is especially "evaluated" at definition time any more than the function body is "evaluated" (parsed into a suitable representation); whereas expressions in the function body are clearly not evaluated in the full sense. From this point of view, consistency would say that expressions in the signature shouldn't be "fully" evaluated either. – LarsH Sep 24 '19 at 15:09

1 I don't mean that you're wrong, only that your conclusion doesn't follow from consistency alone. – LarsH Sep 24 '19 at 15:09

@LarsH Defaults are neither part of the body, nor am I claiming that consistency is the only criteria. Can you make a suggestion how to clarify the answer? – MisterMiyagi Sep 24 '19 at 15:15

6

I think the answer to this question lies in how python pass data to parameter (pass by value or by reference), not mutability or how python handle the "def" statement.

A brief introduction. First, there are two type of data types in python, one is simple elementary data type, like numbers, and another data type is objects. Second, when passing data to parameters, python pass elementary data type by value, i.e., make a local copy of the value to a local variable, but pass object by reference, i.e., pointers to the object.

Admitting the above two points, let's explain what happened to the python code. It's only because of passing by reference for objects, but has nothing to do with mutable/immutable, or arguably the fact that "def" statement is executed only once when it is defined.

[] is an object, so python pass the reference of [] to `a`, i.e., `a` is only a pointer to [] which lies in memory as an object. There is only one copy of [] with, however, many references to it. For the first foo(), the list [] is changed to 1 by append method. But Note that there is only one copy of the list object and this object now becomes 1. When running the second foo(), what effbot webpage says (items is not evaluated any more) is wrong. `a` is evaluated to be the list object, although now the content of the object is 1. This is the effect of passing by reference! The result of foo(3) can be easily derived in the same way.

To further validate my answer, let's take a look at two additional codes.

====== No. 2 ========

```
def foo(x, items=None):
    if items is None:
        items = []
    items.append(x)
    return items

foo(1)  #return [1]
foo(2)  #return [2]
foo(3)  #return [3]
```

`[]` is an object, so is `None` (the former is mutable while the latter is immutable. But the mutability has nothing to do with the question). None is somewhere in the space but we know it's there and there is only one copy of None there. So every time foo is invoked, items is evaluated (as opposed to some answer that it is only evaluated once) to be None, to be clear, the reference (or the address) of None. Then in the foo, item is changed to [], i.e., points to another object which has a different address.

====== No. 3 =======

```
def foo(x, items=[]):
    items.append(x)
    return items

foo(1)    # returns [1]
foo(2,[]) # returns [2]
foo(3)    # returns [1,3]
```

The invocation of foo(1) make items point to a list object [] with an address, say, 11111111. the content of the list is changed to 1 in the foo function in the sequel, but the address is not changed, still 11111111. Then foo(2,[]) is coming. Although the [] in foo(2,[]) has the same content as the default parameter [] when calling foo(1), their address are different! Since we provide the parameter explicitly, `items` has to take the address of this new `[]`, say 2222222, and return it after making some change. Now foo(3) is executed. since only `x` is provided, items has to take its default value again. What's the default value? It is set when defining the foo function: the list object located in 11111111. So the items is evaluated to be the address 11111111 having an element 1. The list located at 2222222 also contains one element 2, but it is not pointed by items any more. Consequently, An append of 3 will make `items` [1,3].

From the above explanations, we can see that the effbot webpage recommended in the accepted answer failed to give a relevant answer to this question. What is more, I think a point in the effbot webpage is wrong. I think the code regarding the UI.Button is correct:

```
for i in range(10):
    def callback():
        print "clicked button", i
    UI.Button("button %s" % i, callback)
```

Each button can hold a distinct callback function which will display different value of `i`. I can provide an example to show this:

```
x=[]
for i in range(10):
    def callback():
        print(i)
    x.append(callback)
```

If we execute `x[7]()` we'll get 7 as expected, and `x[9]()` will gives 9, another value of `i`.

Share Follow

answered Aug 22 '13 at 5:58

user2384994

1,609 4 23 28

5 Your last point is wrong. Try it and you'll see that `x[7]()` is `9`. – Duncan Oct 2 '13 at 13:29

3 "python pass elementary data type by value, i.e., make a local copy of the value to a local variable" is completely incorrect. I am astonished that someone can obviously know Python very well, yet have such horrible misunderstanding of fundamentals. :-(– Veky Nov 19 '14 at 9:07

1 2 Next

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.