Buggy Python Code: The 10 Most Common Mistakes That Python Developers Make

View all articles

by **Martin Chikilian** - Python Software Engineer @ Toptal

#Python

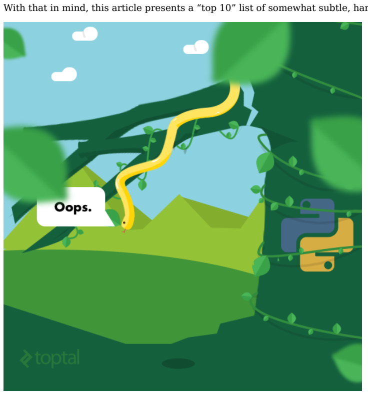
About Python

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components or services. Python supports modules and packages, thereby encouraging program modularity and code reuse.

About this article

Python's simple, easy-to-learn syntax can mislead Python developers – especially those who are newer to the language – into missing some of its subtleties and underestimating the power of the diverse Python language.

With that in mind, this article presents a "top 10" list of somewhat subtle, harder-to-catch mistakes that can bite even some more advanced Python developers in the rear.



(Note: This article is intended for a more advanced audience than Common Mistakes of Python Programmers, which is geared more toward those who are newer to the language.)

Common Mistake #1: Misusing expressions as defaults for function arguments

Python allows you to specify that a function argument is *optional* by providing a *default value* for it. While this is a great feature of the language, it can lead to some confusion when the default value is *mutable*. For example, consider this Python function definition:

```
>>> def foo(bar=[]):        # bar is optional and defaults to [] if not specified
...    bar.append("baz")    # but this line could be problematic, as we'll see...
...    return bar
```

A common mistake is to think that the optional argument will be set to the specified default expression *each time* the function is called without supplying a value for the optional argument. In the above code, for example, one might expect that calling `foo()` repeatedly (i.e., without specifying a `bar` argument) would always return `'baz'`, since the assumption would be that *each time* `foo()` is called (without a `bar` argument specified) `bar` is set to `[]` (i.e., a new empty list).

But let's look at what actually happens when you do this:

```
>>> foo()
["baz"]
>>> foo()
["baz", "baz"]
>>> foo()
["baz", "baz", "baz"]
```

Huh? Why did it keep appending the default value of `"baz"` to an *existing* list each time `foo()` was called, rather than creating a *new* list each time?

The more advanced Python programming answer is that *the default value for a function argument is only evaluated once, at the time that the function is defined*. Thus, the `bar` argument is initialized to its default (i.e., an empty list) only when `foo()` is first defined, but then calls to `foo()` (i.e., without a `bar` argument specified) will continue to use the same list to which `bar` was originally initialized.

FYI, a common workaround for this is as follows:

```
>>> def foo(bar=None):
...    if bar is None:          # or if not bar:
...        bar = []
...    bar.append("baz")
...    return bar
...
>>> foo()
["baz"]
>>> foo()
["baz"]
>>> foo()
["baz"]
```

Common Mistake #2: Using class variables incorrectly

Consider the following example:

```
>>> class A(object):
...     x = 1
...
>>> class B(A):
...     pass
...
>>> class C(A):
...     pass
...
>>> print A.x, B.x, C.x
1 1 1
```

Makes sense.

```
>>> B.x = 2
>>> print A.x, B.x, C.x
1 2 1
```

Yup, again as expected.

```
>>> A.x = 3
>>> print A.x, B.x, C.x
3 2 3
```

What the *$%#!&*?? We only changed `A.x`. Why did `C.x` change too?

In Python, class variables are internally handled as dictionaries and follow what is often referred to as Method Resolution Order (MRO). So in the above code, since the attribute `x` is not found in class `C`, it will be looked up in its base classes (only `A` in the above example, although Python supports multiple inheritance). In other words, `C` doesn't have its own `x` property, independent of `A`. Thus, references to `C.x` are in fact references to `A.x`. This causes a Python problem unless it's handled properly. Learn more aout class attributes in Python.

Common Mistake #3: Specifying parameters incorrectly for an exception block

Suppose you have the following code:

```
>>> try:
...     l = ["a", "b"]
...     int(l[2])
... except ValueError, IndexError:  # To catch both exceptions, right?
...     pass
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
IndexError: list index out of range
```

The problem here is that the `except` statement does *not* take a list of exceptions specified in this manner. Rather, In Python 2.x, the syntax `except Exception, e` is used to bind the exception to the *optional* second parameter specified (in this case `e`), in order to make it available for further inspection. As a result, in the above code, the `IndexError` exception is *not* being caught by the `except` statement; rather, the exception instead ends up being bound to a parameter named `IndexError`.

The proper way to catch multiple exceptions in an `except` statement is to specify the first parameter as a tuple containing all exceptions to be caught. Also, for maximum portability, use the `as` keyword, since that syntax is supported by both Python 2 and Python 3:

```
>>> try:
...     l = ["a", "b"]
...     int(l[2])
... except (ValueError, IndexError) as e:
...     pass
...
>>>
```

Common Mistake #4: Misunderstanding Python scope rules

Python scope resolution is based on what is known as the LEGB rule, which is shorthand for **L**ocal, **E**nclosing, **G**lobal, **B**uilt-in. Seems straightforward enough, right? Well, actually, there are some subtleties to the way this works in Python, which brings us to the common more advanced Python programming problem below. Consider the following:

```
>>> x = 10
>>> def foo():
...     x += 1
...     print x
...
>>> foo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in foo
UnboundLocalError: local variable 'x' referenced before assignment
```

What's the problem?

The above error occurs because, when you make an *assignment* to a variable in a scope, *that variable is automatically considered by Python to be local to that scope* and shadows any similarly named variable in any outer scope.

Many are thereby surprised to get an `UnboundLocalError` in previously working code when it is modified by adding an assignment statement somewhere in the body of a function. (You can read more about this here.)

It is particularly common for this to trip up developers when using lists. Consider the following example:

```
>>> lst = [1, 2, 3]
>>> def foo1():
...     lst.append(5)   # This works ok...
...
>>> foo1()
>>> lst
[1, 2, 3, 5]

>>> lst = [1, 2, 3]
>>> def foo2():
...     lst += [5]      # ... but this bombs!
...
>>> foo2()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in foo
UnboundLocalError: local variable 'lst' referenced before assignment
```

Huh? Why did `foo2` bomb while `foo1` ran fine?

The answer is the same as in the prior example problem, but is admittedly more subtle. `foo1` is not making an *assignment* to `lst`, whereas `foo2` is. Remembering that `lst += [5]` is really just shorthand for `lst = lst + [5]`, we see that we are attempting to *assign* a value to `lst` (therefore presumed by Python to be in the local scope). However, the value we are looking to assign to `lst` is based on `lst` itself (again, now presumed to be in the local scope), which has not yet been defined. Boom.

Common Mistake #5: Modifying a list while iterating over it

The problem with the following code should be fairly obvious:

```
>>> odd = lambda x : bool(x % 2)
>>> numbers = [n for n in range(10)]
>>> for i in range(len(numbers)):
...     if odd(numbers[i]):
...         del numbers[i]  # BAD: Deleting item from a list while iterating over it
...
Traceback (most recent call last):
          File "<stdin>", line 2, in <module>
IndexError: list index out of range
```

Deleting an item from a list or array while iterating over it is a Python problem that is well known to any experienced software developer. But while the example above may be fairly obvious, even advanced developers can be unintentionally bitten by this in code that is much more complex.

Fortunately, Python incorporates a number of elegant programming paradigms which, when used properly, can result in significantly simplified and streamlined code. A side benefit of this is that simpler code is less likely to be bitten by the accidental-deletion-of-a-list-item-while-iterating-over-it bug. One such paradigm is that of list comprehensions. Moreover, list comprehensions are particularly useful for avoiding this specific problem, as shown by this alternate implementation of the above code which works perfectly:

```
>>> odd = lambda x : bool(x % 2)
>>> numbers = [n for n in range(10)]
>>> numbers[:] = [n for n in numbers if not odd(n)]  # ahh, the beauty of it all
>>> numbers
[0, 2, 4, 6, 8]
```


Common Mistake #6: Confusing how Python binds variables in closures

Considering the following example:

```
>>> def create_multipliers():
...     return [lambda x : i * x for i in range(5)]
>>> for multiplier in create_multipliers():
...     print multiplier(2)
...
```

You might expect the following output:

```
0
2
4
6
8
```

But you actually get:

```
8
8
8
8
8
```

Surprise!

This happens due to Python's *late binding* behavior which says that the values of variables used in closures are looked up at the time the inner function is called. So in the above code, whenever any of the returned functions are called, the value of `i` is looked up *in the surrounding scope at the time it is called* (and by then, the loop has completed, so `i` has already been assigned its final value of 4).

The solution to this common Python problem is a bit of a hack:

```
>>> def create_multipliers():
...     return [lambda x, i=i : i * x for i in range(5)]
...
>>> for multiplier in create_multipliers():
...     print multiplier(2)
...
0
2
4
6
8
```

Voilà! We are taking advantage of default arguments here to generate anonymous functions in order to achieve the desired behavior. Some would call this elegant. Some would call it subtle. Some hate it. But if you're a Python developer, it's important to understand in any case.

Common Mistake #7: Creating circular module dependencies

Let's say you have two files, `a.py` and `b.py`, each of which imports the other, as follows:

In `a.py`:

```
import b

def f():
    return b.x

print f()
```

And in `b.py`:

```
import a

x = 1

def g():
    print a.f()
```

First, let's try importing `a.py`:

```
>>> import a
1
```

Worked just fine. Perhaps that surprises you. After all, we do have a circular import here which presumably should be a problem, shouldn't it?

The answer is that the mere *presence* of a circular import is not in and of itself a problem in Python. If a module has already been imported, Python is smart enough not to try to re-import it. However, depending on the point at which each module is attempting to access functions or variables defined in the other, you may indeed run into problems.

So returning to our example, when we imported `a.py`, it had no problem importing `b.py`, since `b.py` does not require anything from `a.py` to be defined *at the time it is imported*. The only reference in `b.py` to `a` is the call to `a.f()`. But that call is in `g()` and nothing in `a.py` or `b.py` invokes `g()`. So life is good.

But what happens if we attempt to import `b.py` (without having previously imported `a.py`, that is):

```
>>> import b
Traceback (most recent call last):
         File "<stdin>", line 1, in <module>
         File "b.py", line 1, in <module>
    import a
         File "a.py", line 6, in <module>
        print f()
         File "a.py", line 4, in f
        return b.x
AttributeError: 'module' object has no attribute 'x'
```

Uh-oh. That's not good! The problem here is that, in the process of importing `b.py`, it attempts to import `a.py`, which in turn calls `f()`, which attempts to access `b.x`. But `b.x` has not yet been defined. Hence the `AttributeError` exception.

At least one solution to this is quite trivial. Simply modify `b.py` to import `a.py` *within* `g()`:

```
x = 1

def g():
    import a    # This will be evaluated only when g() is called
    print a.f()
```

No when we import it, everything is fine:

```
>>> import b
>>> b.g()
1       # Printed a first time since module 'a' calls 'print f()' at the end
1       # Printed a second time, this one is our call to 'g'
```

Common Mistake #8: Name clashing with Python Standard Library modules

One of the beauties of Python is the wealth of library modules that it comes with "out of the box". But as a result, if you're not consciously avoiding it, it's not that difficult to run into a name clash between the name of one of your modules and a module with the same name in the standard library that ships with Python (for example, you might have a module named `email.py` in your code, which would be in conflict with the standard library module of the same name).

This can lead to gnarly problems, such as importing another library which in turns tries to import the Python Standard Library version of a module but, since you have a module with the same name, the other package mistakenly imports your version instead of the one within the Python Standard Library. This is where bad Python errors happen.

Care should therefore be exercised to avoid using the same names as those in the Python Standard Library modules. It's way easier for you to change the name of a module within your package than it is to file a [Python Enhancement Proposal (PEP)] to request a name change upstream and to try and get that approved.

Common Mistake #9: Failing to address differences between Python 2 and Python 3

Consider the following file `foo.py`:

```
import sys

def bar(i):
    if i == 1:
        raise KeyError(1)
    if i == 2:
        raise ValueError(2)

def bad():
    e = None
    try:
        bar(int(sys.argv[1]))
    except KeyError as e:
        print('key error')
    except ValueError as e:
        print('value error')
    print(e)

bad()
```

On Python 2, this runs fine:

```
$ python foo.py 1
key error
1
$ python foo.py 2
value error
2
```

But now let's give it a whirl on Python 3:

```
$ python3 foo.py 1
key error
Traceback (most recent call last):
  File "foo.py", line 19, in <module>
    bad()
  File "foo.py", line 17, in bad
    print(e)
UnboundLocalError: local variable 'e' referenced before assignment
```

What has just happened here? The "problem" is that, in Python 3, the exception object is not accessible beyond the scope of the `except` block. (The reason for this is that, otherwise, it would keep a reference cycle with the stack frame in memory until the garbage collector runs and purges the references from memory. More technical detail about this is available here).

One way to avoid this issue is to maintain a reference to the exception object *outside* the scope of the `except` block so that it remains accessible. Here's a version of the previous example that uses this technique, thereby yielding code that is both Python 2 and Python 3 friendly:

```
import sys

def bar(i):
    if i == 1:
        raise KeyError(1)
    if i == 2:
        raise ValueError(2)

def good():
    exception = None
    try:
        bar(int(sys.argv[1]))
    except KeyError as e:
        exception = e
        print('key error')
    except ValueError as e:
        exception = e
        print('value error')
    print(exception)

good()
```

Running this on Py3k:

```
$ python3 foo.py 1
key error
1
$ python3 foo.py 2
value error
2
```

Yippee!

(Incidentally, our Python Hiring Guide discusses a number of other important differences to be aware of when migrating code from Python 2 to Python 3.)

Common Mistake #10: Misusing the `__del__` method

Let's say you had this in a file called `mod.py`:

```
import foo

class Bar(object):
        ...
    def __del__(self):
        foo.cleanup(self.myhandle)
```

And you then tried to do this from `another_mod.py`:

```
import mod
mybar = mod.Bar()
```

You'd get an ugly `AttributeError` exception.

Why? Because, as reported here, when the interpreter shuts down, the module's global variables are all set to `None`. As a result, in the above example, at the point that `__del__` is invoked, the name `foo` has already been set to `None`.

A solution to this somewhat more advanced Python programming problem would be to use `atexit.register()` instead. That way, when your program is finished executing (when exiting normally, that is), your registered handlers are kicked off *before* the interpreter is shut down.

With that understanding, a fix for the above `mod.py` code might then look something like this:

```
import foo
import atexit

def cleanup(handle):
    foo.cleanup(handle)


class Bar(object):
    def __init__(self):
        ...
        atexit.register(cleanup, self.myhandle)
```

This implementation provides a clean and reliable way of calling any needed cleanup functionality upon normal program termination. Obviously, it's up to `foo.cleanup` to decide what to do with the object bound to the name `self.myhandle`, but you get the idea.

Wrap-up

Python is a powerful and flexible language with many mechanisms and paradigms that can greatly improve productivity. As with any software tool or language, though, having a limited understanding or appreciation of its capabilities can sometimes be more of an impediment than a benefit, leaving one in the proverbial state of "knowing enough to be dangerous".

Familiarizing oneself with the key nuances of Python, such as (but by no means limited to) the moderately advanced programming problems raised in this article, will help optimize use of the language while avoiding some of its more common errors.

You might also want to check out our Insider's Guide to Python Interviewing for suggestions on interview questions that can help identify Python experts.

We hope you've found the pointers in this article helpful and welcome your feedback.

About the author

View full profile »
Hire the Author
Martin Chikilian, Argentina
member since October 3, 2011
PythonGitHubMySQL
Martin has worked as a professional Python developer since 2007, although his career in IT began in 2001. He is a full-stack engineer, having administered operating systems and networks for so many years. His recent interests are web-development and Python (namely with Django). [click to continue...]

64 Comments Toptal Login

Recommend 12 Share Sort by Best

Join the discussion…

Andres Moreno • 2 years ago
for common mistake #6 you can replace the list comprehension with a generator (substitute the square brackets for parentheses) and the code will work fine....

In [6]: def create_multipliers():

...: return (lambda x: x*i for i in range(5))

...:

In [7]: for multiplies in create_multipliers():

...: print multiplies(2)

...:

0

2

4

6

8

11 • Reply • Share ›

Martin Chikilian → Andres Moreno • 2 years ago
Very nice one!
1 • Reply • Share ›

Guest • 2 years ago
For Common Mistake #3, you should be using the `as` syntax for capturing exceptions, as it's both Python2.6+ & Python3+ compatible:

https://docs.python.org/3/howt...
10 • Reply • Share ›

H. Singer Mod → Guest • 2 years ago
Good point! We updated this accordingly. Thanks for the input.

[Toptal blog editor]
1 • Reply • Share ›

Adam Hitchcock Disqus Engineering • 2 years ago
I found the `numbers[:]` an odd way of doing that assignment. So I looked it up. It looks like it keeps the same object.

```
>>> odd = lambda x : bool(x % 2)
>>> numbers = [n for n in range(10)]
>>> id(numbers)
4424446032
>>> numbers = [n for n in numbers if not odd(n)] # ahh, the beauty of it all
>>> id(numbers)
4424505176
>>> numbers
[0, 2, 4, 6, 8]
>>>
>>>
>>> odd = lambda x : bool(x % 2)
>>> numbers = [n for n in range(10)]
>>> id(numbers)
4424446032
>>> numbers[:] = [n for n in numbers if not odd(n)] # ahh, the beauty of it all
```

see more

7 • Reply • Share ›

Laurent → Adam Hitchcock • 2 years ago
One reason where I would see a use of [:] is if you have other names pointing to the array, and you want them to see the change.
1 • Reply • Share ›

Martin Chikilian → Adam Hitchcock • 2 years ago
As you correctly pointed out the form without the empty slice `numbers = ...` creates a new list and binds it to a new `numbers` name whereas using the `numbers[:] = ...` version modifies the already existent object. Code in this example wasn't written taking performance into consideration and the use of the empty slice notation is only to reflect that the same list as before has been modified ('[n for n in range(10)]')
1 • Reply • Share ›

Eli Collins • 2 years ago
For mistake #4, might want to point out that Python 3 added the `nonlocal` keyword, which indicates that a variable should be shared with the enclosing scope rather than shadow it.

PEP 3104 -- http://legacy.python.org/dev/p...
6 • Reply • Share ›

not me • 2 years ago
Common Mistake #-1: Being a programmer
8 • Reply • Share ›

Adam Hitchcock Disqus Engineering → not me • 2 years ago
You mean mistake #0.
27 • Reply • Share ›

Pooya Eghbali • 2 years ago
You can replace:
if bar is None: bar = []
with:
bar = bar or []
4 • Reply • Share ›

Tetsuya Morimoto → Pooya Eghbali • 2 years ago
No. The boolean value is able to change with a special method if bar is a custom object.
- py2: __nonzero__
- py3: __bool__
• Reply • Share ›

Abhishek Anurag → Tetsuya Morimoto • 2 years ago
Good point.
Also, I'm not sure if going by the suggestion "#or not bar" works correctly in all cases, I tried that with a list in the code below and it does not work as is explained.

def inorder(self, inorder_opt=None):
if inorder_opt is None: #if not bar doesn't work correctly here, get's reinitialized again in recursive calls
inorder_opt = []

if (self.left != None):
self.left.inorder(inorder_opt)

if (self.key != None):
inorder_opt.append(self.key)

if (self.right != None):
self.right.inorder(inorder_opt)

return inorder_opt
• Reply • Share ›

Kim Scheibel • 2 years ago
Re. #1: If you're mutating arguments that have been passed by reference, and you're surprised that this has sideeffects, then you have problems more generally than just when handling defaulted arguments. For example, what happens when the client actually does pass in a list (which in Python is always a reference) and your function blithely changes it? I know this is a minority view, but I don't think that the solution is to do the "if arg is None" dance (notwithstanding the Google style guide, etc). The solution is for the implementor of any function to understand that changing an argument that was passed in by reference, generally, has sideeffects, and that this should only be done very deliberately, and then only when the function is named to make it obvious that it's going to mess with its arguments.
4 • Reply • Share ›

Jürgen Erhard → Kim Scheibel • 2 years ago
Yeah, with Python, one of the first things you really have to understand, nay, to grok, is "mutable vs immutable". And all kinds of "weird behavior" then become clear.
• Reply • Share ›

Vaquero de Oro • 2 years ago
This post should be renamed "Top 10 Mistakes Made in Python's Design" (which most often confuse sensible people), IMO.
3 • Reply • Share ›

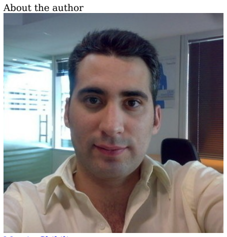
Mario Campos • 2 years ago
I noticed something strange about common mistake #5. You used

```
numbers = [n for n in range(10)]
```

Trending articles
Meet Ecto, The No-Compromise Database Wrapper For Concurrent Elixir Apps3 days agoSocial Network APIs: The Internet's Portal to the Real World6 days agoWrite Tests That Matter: Tackle The Most Complex Code First12 days agoHow to Build a Multitenant Application: A Hibernate Tutorial20 days agoA New Way of Using Email for Support Apps: An AWS Tutorial23 days agoBoost Your Productivity With Clever Travel Hardware25 days agoHunting Down Memory Issues In Ruby: A Definitive Guideabout 1 month agoIntroduction to Kotlin: Android Programming For Humansabout 1 month ago
Relevant technologies

- Back-end
- Software
- Python

About the author



Martin Chikilian
Python Developer
Martin has worked as a professional Python developer since 2007, although his career in IT began in 2001. He is a full-stack engineer, having administered operating systems and networks for so many years. His recent interests are web-development and Python (namely with Django).

Home › Blog › Buggy Python Code: The 10 Most Common Mistakes That Python Developers Make

-
- Top 3%
- Why
- Clients
- Partners
- Community
- Blog
- About Us
- Hire a developer
- Apply as a Developer
- Login
-
  - Questions?
  - Contact Us
  - Call us:
  - +1.888.604.3188
  -

[Hire a developer](https://www.toptal.com)