Common Pitfalls All Versions

Improvements requested:

Examples

List multiplication and common references

Consider the case of creating a nested list structure by multiplying:

```
li = [[]] * 3
print(li)
# Out: [[], [], []]
```

At first glance we would think we have a list of containing 3 different nested lists. Let's try to append `1` to the first one:

```
li[0].append(1)
print(li)
# Out: [[1], [1], [1]]
```

`1` got appended to all of the lists in `li`.

The reason is that `[[]] * 3` doesn't create a `list` of 3 different `list`s. Rather, it creates a `list` holding 3 references to the same `list` object. As such, when we append to `li[0]` the change is visible in all sub-elements of `li`. This is equivalent of:

```
li = []
element = []
li.append(element)
li.append(element)
li.append(element)
print(li)
# Out: [[], [], []]
element.append(1)
print(li)
# Out: [[1], [1], [1]]
```

This can be further corroborated if we print the memory addresses of the contained `list` by using `id`:

```
li = [[]] * 3
print([id(inner_list) for inner_list in li])
# Out: [6830760, 6830760, 6830760]
```

The solution is to create the inner lists with a loop:

```
li = [[] for _ in range(3)]
```

Instead of creating a single `list` and then making 3 references to it, we now create `3` different distinct lists. This, again, can be verified by using the `id` function:

```
print([id(inner_list) for inner_list in li])
# Out: [6331048, 6331528, 6331488]
```

You can also do this. It causes a new empty list to be created in each `append` call.

```
>>> li = []
>>> li.append([])
>>> li.append([])
>>> li.append([])
>>> for k in li: print(id(k))
...
4315469256
4315564552
4315564808
>>>
```

Mutable default argument

```
def foo(li=[]):
    li.append(1)
    print(li)
```

```
foo([2])
# Out: [2, 1]
foo([3])
# Out: [3, 1]
```

This code behaves as expected, but what if we don't pass an argument?

```
foo()
# Out: [1] As expected...

foo()
# Out: [1, 1]  Not as expected...
```

This is because default arguments of functions and methods are evaluated at **definition** time rather than run time. So we only ever have a single instance of the `li` list.

The way to get around it is to use only immutable types for default arguments:

```
def foo(li=None):
    if not li:
        li = []
    li.append(1)
    print(li)

foo()
# Out: [1]

foo()
# Out: [1]
```

While an improvement and although `if not li` correctly evaluates to `False`, many other objects do as well, such as zero-length sequences. The following example arguments can cause unintended results:

```
x = []
foo(li=x)
# Out: [1]

foo(li="")
# Out: [1]

foo(li=0)
# Out: [1]
```

The idiomatic approach is to directly check the argument against the `None` object:

```
def foo(li=None):
    if li is None:
        li = []
    li.append(1)
    print(li)

foo()
# Out: [1]
```

Changing the sequence you are iterating over

A `for` loop iterates over a sequence, so **altering this sequence inside the loop could lead to unexpected results** (especially when adding or removing elements):

```
alist = [0, 1, 2]
for index, value in enumerate(alist):
    alist.pop(index)
print(alist)
# Out: [1]
```

Note: `list.pop()` is being used to remove elements from the list.

The second element was not deleted because the iteration goes through the indices in order. The above loop iterates twice, with the following results:

```
# Iteration #1
index = 0
alist = [0, 1, 2]
alist.pop(0) # removes '0'

# Iteration #2
index = 1
alist = [1, 2]
alist.pop(1) # removes '2'
```

```
# loop terminates, but alist is not empty:
alist = [1]
```

This problem arises because the indices are changing while iterating in the direction of increasing index. To avoid this problem, you can **iterate through the loop backwards** :

```
alist = [1,2,3,4,5,6,7]
for index, item in reversed(list(enumerate(alist))):
    # delete all even items
    if item % 2 == 0:
        alist.pop(index)
print(alist)
# Out: [1, 3, 5, 7]
```

By iterating through the loop starting at the end, as items are removed (or added), it does not affect the indices of items earlier in the list. So this example will properly remove all items that are even from `alist` .

A similar problem arises when **inserting or appending elements to a list that you are iterating over** , which can result in an infinite loop:

```
alist = [0, 1, 2]
for index, value in enumerate(alist):
    # break to avoid infinite loop:
    if index == 20:
        break
    alist.insert(index, 'a')
print(alist)
# Out (abbreviated): ['a', 'a', ..., 'a', 'a',  0,   1,   2]
```

Without the `break` condition the loop would insert `'a'` as long as the computer does not run out of memory and the program is allowed to continue. In a situation like this, it is usually preferred to create a new list, and add items to the new list as you loop through the original list.

When using a `for` loop, **you cannot modify the list elements with the placeholder variable** :

```
alist = [1,2,3,4]
for item in alist:
    if item % 2 == 0:
        item = 'even'
print(alist)
# Out: [1,2,3,4]
```

In the above example, **changing `item` doesn't actually change anything in the original list** . You need to use the list index (`alist[2]`), and `enumerate()` works well for this:

```
alist = [1,2,3,4]
for index, item in enumerate(alist):
    if item % 2 == 0:
        alist[index] = 'even'
print(alist)
# Out: [1, 'even', 3, 'even']
```

A **`while` loop** might be a better choice in some cases:

If you are going to **delete all the items** in the list:

```
zlist = [0, 1, 2]
while zlist:
    print(zlist[0])
    zlist.pop(0)
print('After: zlist =', zlist)

# Out: 0
#      1
#      2
# After: zlist = []
```

Although simply resetting `zlist` will accomplish the same result;

```
zlist = []
```

The above example can also be combined with `len()` to stop after a certain point, or to delete all but `x` items in the list:

```
zlist = [0, 1, 2]
x = 1
while len(zlist) > x:
    print(zlist[0])
    zlist.pop(0)
print('After: zlist =', zlist)
```

```
# Out: 0
#      1
# After: zlist = [2]
```

Or to **loop through a list while deleting elements that meet a certain condition** (in this case deleting all even elements):

```
zlist = [1,2,3,4,5]
i = 0
while i < len(zlist):
    if zlist[i] % 2 == 0:
        zlist.pop(i)
    else:
        i += 1
print(zlist)
# Out: [1, 3, 5]
```

Notice that you don't increment `i` after deleting an element. By deleting the element at `zlist[i]`, the index of the next item has decreased by one, so by checking `zlist[i]` with the same value for `i` on the next iteration, you will be correctly checking the next item in the list.

A contrary way to think about removing unwanted items from a list, is to **add wanted items to a new list**. The following example is an alternative to the latter `while` loop example:

```
zlist = [1,2,3,4,5]

z_temp = []
for item in zlist:
    if item % 2 != 0:
        z_temp.append(item)
zlist = z_temp
print(zlist)
# Out: [1, 3, 5]
```

Here we are funneling desired results into a new list. We can then optionally reassign the temporary list to the original variable.

With this trend of thinking, you can invoke one of Python's most elegant and powerful features, **list comprehensions**, which eliminates temporary lists and diverges from the previously discussed in-place list/index mutation ideology.

```
zlist = [1,2,3,4,5]
[item for item in zlist if item % 2 != 0]
# Out: [1, 3, 5]
```

Integer and String identity

Python uses internal caching for a range of integers to reduce unnecessary overhead from their repeated creation.

In effect, this can lead to confusing behavior when comparing integer identities:

```
>>> -8 is (-7 - 1)
False
>>> -3 is (-2 - 1)
True
```

and, using another example:

```
>>> (255 + 1) is (255 + 1)
True
>>> (256 + 1) is (256 + 1)
False
```

Wait what?

We can see that the identity operation `is` yields `True` for some integers (`-3`, `256`) but no for others (`-8`, `257`).

To be more specific, integers in the range `[-5, 256]` are internally cached during interpreter startup and are only created once. As such, they are **identical** and comparing their identities with `is` yields `True`; integers outside this range are (usually) created on-the-fly and their identities compare to `False`.

This is a common pitfall since this is a common range for testing, but often enough, the code fails in the later staging process (or worse - production) with no apparent reason after working perfectly in development.

The solution is to **always compare values using the equality** (`==`) operator and **not** the identity (`is`) operator.

Python also keeps references to commonly used strings and can result in similarly confusing behavior when comparing identities (i.e. using `is`) of strings.

```
>>> 'python' is 'py' + 'thon'
True
```

The string `'python'` is commonly used, so Python has one object that all references to the string `'python'` use.

For uncommon strings, comparing identity fails even when the strings are equal.

```
>>> 'this is not a common string' is 'this is not' + ' a common string'
False
>>> 'this is not a common string' == 'this is not' + ' a common string'
True
```

So, just like the rule for Integers, **always compare string values using the equality** (`==`) operator and **not** the identity (`is`) operator.

Dictionaries are unordered

You might expect a python dictionary to be sorted by keys like, for example, a C++ `std::map` , but this is not the case:

```
dict = {'first': 1, 'second': 2, 'third': 3}
print(dict)
# Out: {'first': 1, 'second': 2, 'third': 3}

print([e for e in dict])
# Out: ['second', 'third', 'first']
```

Python doesn't have any built-in class to do that.

However,if sorting is not a must, and you just want your dictionary to remember the order of insertion of it's key,value pairs, you could use `collections.OrderedDict` instead:

```
from collections import OrderedDict

oDict = OrderedDict([('first', 1), ('second', 2), ('third', 3)])

print([e for e in oDict])
# Out: ['first', 'second', 'third']
```

Also keep in mind that initializing an `OrderedDict` with a standard dictionary won't give you the order you want either.

The implementation of dictionaries was changed in python 3.6 , and the insertion order of keys in a dictionary is now preserved in some implementations. As a side effect, this also preserves the order of keyword arguments passed to a function:

```
>>> def func(**kw): print(kw.keys())
...
>>> func(a=1, b=2, c=3, d=4, e=5)
dict_keys(['a', 'b', 'c', 'd', 'e']) # expected order
```

As mentioned in the "What's New" section of the Python 3.6 documentation :

> The order-preserving aspect of this new implementation is considered an implementation detail and should not be relied upon (this may change in the future, but it is desired to have this new dict implementation in the language for a few releases before changing the language spec to mandate order-preserving semantics for all current and future Python implementations; this also helps preserve backwards-compatibility with older versions of the language where random iteration order is still in effect, e.g. Python 3.5)

Variable leaking in list comprehensions and for loops

Consider the following list comprehension

Python 2.x ≤2.7

```
i = 0
a = [i for i in range(3)]
print(i) # Outputs 2
```

This occurs only in Python 2 due to the fact that the list comprehension “leaks” the loop control variable into the surrounding scope (source). This behavior can lead to hard-to-find bugs and ***it has been fixed in Python 3*** .

Python 3.x ≥3.0

```
i = 0
a = [i for i in range(3)]
print(i) # Outputs 0
```

Similarly, for loops have no private scope for their iteration variable

```
i = 0
for i in range(3):
    pass
print(i) # Outputs 2
```

This type of behavior occurs both in Python 2 and Python 3.

To avoid issues with leaking variables, use new variables in list comprehensions and for loops as appropriate.

Chaining of or operator

When testing for any of several equality comparisons:

```
if a == 3 or b == 3 or c == 3:
```

it is tempting to abbreviate this to

```
if a or b or c == 3: # Wrong
```

This is wrong; the `or` operator has lower precedence than `==` , so the expression will be evaluated as `if (a) or (b) or (c == 3):` . The correct way is explicitly checking all the conditions:

```
if a == 3 or b == 3 or c == 3:  # Right Way
```

Alternately, the built-in `any()` function may be used in place of chained `or` operators:

```
if any([a == 3, b == 3, c == 3]): # Right
```

Or, to make it more efficient:

```
if any(x == 3 for x in (a, b, c)): # Right
```

Or, to make it shorter:

```
if 3 in (a, b, c): # Right
```

Here, we use the `in` operator to test if the value is present in a tuple containing the values we want to compare against.

Similarly, it is incorrect to write

```
if a == 1 or 2 or 3:
```

which should be written as

```
if a in (1, 2, 3):
```

Accessing int literals' attributes

You might have heard that everything in Python is an object, even literals. This means, for example, `7` is an object as well, which means it has attributes. For example, one of these attributes is the `bit_length` . It returns the amount of bits needed to represent the value it is called upon.

```
x = 7
x.bit_length()
# Out: 3
```

Seeing the above code works, you might intuitively think that `7.bit_length()` would work as well, only to find out it raises a `SyntaxError` . Why? because the interpreter needs to differentiate between an attribute access and a floating number (for example `7.2` or `7.bit_length()`). It can't, and that's why an exception is raised.

There are a few ways to access an `int` literals' attributes:

```
# parenthesis
(7).bit_length()
# a space
7 .bit_length()
```

Using two dots (like this `7..bit_length()`) doesn't work in this case, because that creates a `float` literal and floats don't have the `bit_length()` method.

This problem doesn't exist when accessing `float` literals' attributes since the interperter is "smart" enough to know that a `float` literal can't contain two `.` , for example:

```
7.2.as_integer_ratio()
# Out: (8106479329266893, 1125899906842624)
```

sys.argv[0] is the name of the file being executed

The first element of `sys.argv[0]` is the name of the python file being executed. The remaining elements are the script arguments.

```
# script.py
import sys

print(sys.argv[0])
print(sys.argv)
```

```
$ python script.py
=> script.py
=> ['script.py']

$ python script.py fizz
=> script.py
=> ['script.py', 'fizz']

$ python script.py fizz buzz
=> script.py
=> ['script.py', 'fizz', 'buzz']
```

Global Interpreter Lock (GIL) and blocking threads

Plenty has been written about Python's GIL . It can sometimes cause confusion when dealing with multi-threaded (not to be confused with multiprocess) applications.

Here's an example:

```
import math
from threading import Thread

def calc_fact(num):
    math.factorial(num)

num = 600000
t = Thread(target=calc_fact, daemon=True, args=[num])
print("About to calculate: {}!".format(num))
t.start()
print("Calculating...")
t.join()
print("Calculated")
```

You would expect to see `Calculating...` printed out immediately after the thread is started, we wanted the calculation to happen in a new thread after all! But in actuality, you see it get printed after the calculation is complete. That is because the new thread relies on a C function (`math.factorial`) which will lock the GIL while it executes.

There are a couple ways around this. The first is to implement your factorial function in native Python. This will allow the main thread to grab control while you are inside your loop. The downside is that this solution will be **a lot** slower, since we're not using the C function anymore.

```
def calc_fact(num):
```

```
    """ A slow version of factorial in native Python """
    res = 1
    while num >= 1:
        res = res * num
        num -= 1
    return res
```

You can also `sleep` for a period of time before starting your execution. Note: this won't actually allow your program to interrupt the computation happening inside the C function, but it will allow your main thread to continue after the spawn, which is what you may expect.

```
def calc_fact(num):
    sleep(0.001)
    math.factorial(num)
```

Syntax

Parameters

Remarks