MIT 6.100L's Style Guide

Table of Contents:

Integers and Floats back top

Be careful when deciding whether to represent data as integers or floats, and be sure that you consider all possible behaviors in computation.

For a detailed explanation of how Python represents floats, read Section 3.4 (pp. 29-31) of your textbook. In short, floats are represented as factors of powers of two, and many decimal numbers cannot be exactly represented in this manner. The most common problem you'll find is trying to check for equality with a floating point number that you are changing. Say that you have a while loop that increments a float by 0.1, and you want this while loop to continue until a control variable equals 0.3. You might find that setting up your loop in the following manner produces an infinite loop:

```
variable = 0.0
while variable != 0.3:
    variable += 0.1
```

That's because `0.1+0.1+0.1` is not equal to `0.3`. `0.1` cannot be precisely represented as a floating point number, so `0.1+0.1+0.1 = 0.30000000000000004`. When you are trying to take simple counts of elements, always use integers to avoid this type of problem

One big difference between Python 2 and Python 3 is that in Python 3, division is by default float division, meaning that dividing `3/2` will give you `1.5`, which returns a float when you divide two integers. If instead you wanted to do integer division, you need to use the integer division operator `//` (two slashes). This will divide the two numbers and then floor (or round towards negative infinity) the result. So, `3//2` will give you `1`, but `-3//2` will give you `-2`. Doing integer division with a float as one of the operands will cast the result to a float, so `3.0//2` will yield `1.0`.

for vs. while Loops back top

Choosing the right loop type makes your code more legible and can also help prevent bugs. Everything that can be written with a for loop can be written with a while loop, but while loops can solve some problems that for loops don't address easily. You should usually write for loops when possible.

In general, use for loops when you know the number of iterations you need to do - e.g., 500 trials, one operation per character in a string, or an action on every element in a list. If you can describe the problem you're trying to solve in terms of each or every element in an iterable object, aim for a for loop. Using a for loop when possible will decrease the risk of writing an infinite loop and will generally prevent you from running into errors with incrementing counter variables.

Example (print 'hello' 500 times):

```
for i in range(500):
    print('hello')
```

Example (add 1 to every element in a list):

```
my_list = [5, 2, 7, -4, 0]
for i in range(len(my_list)):
    my_list[i] += 1
```

If you're instead iterating for a certain condition to be satisfied, you want to use a *while* loop. While loops are useful when you can define the number of iterations in terms of a boolean variable. If you are waiting for a user to enter an input correctly or are waiting for a randomly generated value to exceed a certain amount, you'll want to use a while loop. Problems that can be described using "*until*" should use while loops.

Example (loop until the user enters a positive number):

```
num = float(input('Enter a positive number: '))
while num <= 0.0:
    num = float(input('Enter a POSITIVE number: '))
```

Example (loop until the randomly generated number is greater than 0.5):

```
import random
num = random.random()
while num <= 0.5:
    num = random.random()
```

To improve the average-case performance of your code, you can sometimes exit out of loops as soon as you find your answer; you'll find many loops that are used to find True/False answers follow this pattern. For example, say you want to check whether any value in a list is great than 5:

```
my_list = [1,2,3,4,5,6,7,8]
greater_than_five = False
for elem in my_list:
    if elem > 5:
        greater_than_five = True
        break
```

Checking Boolean Conditions with if/else back top

Often, people have a tendency to be overly verbose. Observe the following example:

```
if my_function() == True: # my_function returns True or False
    return True
else:
    return False
```

When Python evaluates my_function(), the code is reduced to the following (let's pretend it returned True):

```
if True == True: # my_function returns True or False
    return True
else:
    return False
```

This seems repetitive, doesn't it? We know that True is equal to True, and False is not. So, instead of keeping that == True around, we could just have the function call inside the if statement:

```
if my_function(): # my_function returns True or False
    return True
else:
    return False
```

There is an important point to note here. Since my_function() is going to be a boolean, and we're effectively returning the value of that boolean, there's no reason not to just return the boolean itself:

```
return my_function() # my_function returns True or False
```

This is nice and concise, but what if we want to return True if my_function returns False, and False when it returns True? There's a Python keyword for that! So, imagine our code starts as:

```
if my_function() == True: # my_function returns True or False
    return False
else:
    return True
```

We can use `not` to write this as:

```
return not my_function() # my_function returns True or False
```

Code Documentation back top

Docstrings
When writing new classes and functions, it is important to document your intent by using docstrings. For instance, in pset 4, since there were a lot of new classes to implement, adding a docstring explaining the purpose of each class is a good idea

Even something as simple as:

```
class CiphertextMessage(Message):
    """
    Subclass of Message that represents a shifted Message
    """
```

Including a docstring means the specification you've written can be accessed by those who try to create an instance of your class. For example, if you change your CiphertextMessage class definition to the above, run the file, then type the following at the interpreter:

```
>>> CiphertextMessage.__doc__
```

You will see your docstring pop up.

Comments
In addition to docstrings, it is important to indicate what specific parts of a function do. It is expected that you comment your code for non trivial areas. This is helpful in 6.00 because you are able to quickly glance at your code and remember why/what you were doing. In addition, it is helpful when a TA/LA is helping you debug.

Here is an example from pset 3, where naming variables of the Robot class is trivial, however, multiplying by 360 is not.

```
self.room = room
self.speed = speed
self.capacity = capacity
self.position = room.get_random_position()
self.direction = random.random()*360 # 360 degrees in a circle
```

Beyond 6.00 it is useful when other people will inevitably read/utilize your code to know what is happening.

Changing Collections while Iterating over them back top

We've mentioned that it's poor practice to modify a collection while iterating over it. This is because the behavior resulting from modification during iteration is ambiguous. The for statement maintains an internal index, which is incremented for each loop iteration. If you modify the list you're looping over, the indices will get out of sync, and you may end up skipping over items or processing the same item multiple times.

Let's look at a couple of examples:

```
elems = ['a', 'b', 'c']
for e in elems:
    print(e)
    elems.remove(e)
```

This prints:

```
a
c
```

Meanwhile, if we look at what elems now contains, we see the following:

```
>>> elems
['b']
```

Why does this happen? Let's look at this code rewritten using a while loop.

```
elems = ['a', 'b', 'c']
i = 0
while i < len(elems):
    e = elems[i]
    print(e)
    elems.remove(e)
    i += 1
```

This code has the same result. Now it's clear what's happening: when you remove the `'a'` from elems, the remaining elements slide down the list. The list is now `['b', 'c']`. Now `'b'` is at index 0, and `'c'` is at index 1. Since the next iteration is going to look at index 1 (which is the `'c'` element), the `'b'` gets skipped entirely! This is not what the programmer intended at all!

Let's take a look at another example. Instead of removing from the list, we are now adding elements to it. What happens in the following piece of code?

```
for e in elems:
    print(e)
    elems.append(e)
```

We might expect the list elems to be `['a', 'b', 'c', 'a', 'b', 'c']` at the end of execution. However, instead, we get an infinite loop as we keep adding new items to the `elems` list while iterating. Oops!

To work around this kind of issue, you can loop over a copy of the list. For instance, in the following code snippets, we wish to retain all the elements of the list the meet some condition.

```
elems_copy = elems[:]
for item in elems_copy:
    if not condition:
        elems.remove(item)
```

`elems` will contain the desired items.

Alternatively, you can create a new list, and append to it:

```
elems_copy = []
for item in elems:
    if condition:
        elems_copy.append(object)
```

`elems_copy` will now contain the desired items.

Note that the same rule applies to the set and dict types; however, mutating a set or dictionary while iterating over it will actually raise a RuntimeError -- in this way, Python explicitly prevents this.

Directly Accessing Instance Variables back top

This can be a problem because it breaks the interface your class provides to the Abstract Data Type. Here is an example of this, as a line of code you might write for something like decode_story in pset 4:

```
shifts = message.shifts # evil
```

This is bad because it means that the instance variable for the text in `PlaintextMessage` had to be stored as `self.shifts` for this to work. However, the programmer writing `PlaintextMessage` (yes, that's you, too) might not have stored it as `self.shifts`, because maybe he/she/you preferred `self.s`, or `self.my_message_shifts`, or maybe the 1st element of the list `self.message_attributes`, or anything else. In fact, the only promise made about `PlaintextMessage` is that there is a constructor that takes in some elements of a Message (e.g. text, shifts), and there is a getter method for some of these properties.

So, the safer way to do this? Use getter methods!

```
shifts = message.get_shifts() # much better!
```

Calling a Superclass Constructor from a Subclass back top

For a subclass that extends the functionality of a superclass (e.g. Message and PlaintextMessage in pset 4), always reuse the functionality of the superclass instead of rewriting it. This goes back to the concept that we should avoid repeated code.

Say we have the following constructor for the Message class:

```
class Message():
    def __init__(self, text):
        self.message_text = text
        self.valid_words = load_words(WORDLIST_FILENAME)
…
```

When we define the PlaintextMessage class, we could just repeat the two lines of code contained in the constructor as in the below code snippet:

```
# Method 1
class PlaintextMessage(Message):
    def __init__(self, text, shifts):
        self.message_text = text
        self.valid_words = load_words(WORDLIST_FILENAME)
        self.shifts = shifts
        self.encryption_dict = {}
…
```

Alternatively, we could make use of inheritance and call the superclass constructor on the first two parameters since this would have the same effect.

```
# Method 2
class Plaintext(Message):
    def __init__(self, text, shifts):
        # Always call the superclass constructor as the first line of
        # the subclass constructor.
        Message.__init__(self, text) # or super().__init__(text)
        self.shifts = shifts
…
```

Method 2 is superior to Method 1 in that we prevent ourselves from repeating the code that initializes the variables `self.message_text` and `self.valid_words`. Why is this so important?

Say for some reason, while implementing Method 1, we decided to call the list of valid words in the PlaintextMessage class `self.list_of_valid_words` instead of `self.valid_words`. In Python, the superclass constructor is not automatically called when constructing a subclass instance. Now, in all of the superclass methods that reference `self.valid_words`, we will get an error because it only exists as `self.list_of_valid_words`.

Another reason to use Method 2 is that the superclass constructor may do some other, more complicated initialization and we want to ensure that it is executed.

Calling Class Methods back top

When you call class methods on objects, `obj.method_name(args)` is preferable over `class_name.method_name(obj, args)`.

`obj.method_name(args)` is preferred because we do not make any assumptions about which class `method_name` belongs to. The Python interpreter will look for a method called `method_name` from within `obj`'s class first. Only if it doesn't find an implementation of `method_name` in that class will it look in parent classes.

Pitfalls of Storing custom Objects in Data Structures back top

Some people tried to store Position objects in their data structure of clean tiles, writing code such as the following:

```
def isTileCleaned(self, m, n):
    newPosition = Position(m, n)
    return newPosition in self.cleanTileList
```

This code will always return False, even if the tile at (m, n) is clean. The reason why this is problematic gets into the internals of how objects are compared for equality in Python.

How does Python compare objects for equality? Well, for primitives like integers, it's pretty simple -- just see if the two numbers have the same value. Similarly, for strings, check if the characters at each position in the string are the same.

Whenever we create a new class from scratch in Python, what is the default way to check for equality?

The answer is that for objects, the default way Python checks if they are equal is to check the location where that object is stored in memory. Because this code creates a new Position object, `newPosition`, it makes sense that this particular instance being created is not stored in the same location as any other Position object in the list of clean tiles! Therefore, of course it won't be found when Python checks to see if it's in the list.

There are a couple of ways to avoid this issue. Our recommended way for the purposes of this course involves changing what is being stored in the data structure. Representing a set of x, y coordinates as a tuple would make testing for equality much simpler.

There are other, better ways to solve this problem that are more complicated. If you'd like to get more information, please come to office hours and ask a TA or LA. :)

Which Data Structure should I use? back top

In 6.0002 problem set 3, we ask you to store the state of cleanliness for w * h tiles in a rectangular room. We saw different solutions to this, and we want to discuss the pros and cons of different approaches.

List
Most people chose to store a list of tuples `(x, y)` for every clean tile. While this works, there are a few things that make this implementation difficult to work with.

The first is that whenever a tile is "cleaned", we must iterate through the entire list (an O(len(cleanTileList)) operation) to see if the tile is not yet considered "clean" before adding the tile's coordinates to the clean tile list. The necessity of this check can lead to bugs and reduces efficiency.

Another issue with this representation is that by including only those tiles that are clean in the list, we are storing a boolean value for each tile implicitly (i.e., if tile is present, it is clean). Here, since what we are doing is storing both the tile's coordinates and something about the tile, expressing it explicitly would be clearer (see dictionary-based solution). For instance, what if we changed the possible states of the room to be one of `"clean"`, `"dirty"`, and `"just a little messy"`? This representation would not be flexible enough to accommodate that.

Set

Another solution involved storing coordinates as a set of tuples in the set if the tile was clean, e.g., `set((x, y),...)`

This solution is superior to the solution using lists, since adding a tuple to the set that already exists will never yield a duplicate, so this is less likely to run into bugs. Additionally, set operations like lookup and removal are O(1), so it is very efficient. However, it has the same problem as the list representation in that we are implicitly storing a boolean, and we should try to make that more explicit.

List of lists

Some people used a list of lists to implement a sort of matrix, with a boolean representing whether or not the room was clean. The list of lists would be indexed twice, corresponding to the x and y coordinates of the tile, to see if that tile is clean. e.g.,

```
[[True, True, True, True, True, True, True, True],
[True, True, True, True, True, True, True, True],
...
[True, True, True, True, True, True, True, True]]
```

This solution avoids the problem of implicit boolean storage, but it is less efficient in that updating an entry requires indexing two lists. It can also be confusing -- knowing which dimension to index first can be tricky.

Dictionary

A more natural way to represent this problem is using a dictionary, where the key is a tuple `(x,y)` of the tile's position, and the value is a boolean that is True if the tile is clean.

This is more flexible in that if we were asked to accommodate the states `"clean"`, `"dirty"`, and `"just a little messy"`, we could switch the value stored in the dictionary to be, say, an integer in the set `{0, 1, 2}`, or even the strings themselves. Updating the cleanliness status of a tile would be a constant time operation (O(1)), and for every tile, we are storing its coordinates and cleanliness status in the same dictionary entry, which is clearer than the other representations.

Takeaway

In this course, we are trying to teach you to use the data structure most appropriate for the problem at hand. For future problems, think about how best to store your data before just picking the data structure that you happen to be most familiar with. :)