# 6.00.2x Style Guide

## Table of Contents:

## 1. Checking boolean conditions with if/else

Often, people have a tendency to be over--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(), this becomes 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 `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*. Here's how this looks:

```
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 is 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
```

## 2. Docstrings

When writing new classes and functions, it is important to document your intent by using docstrings. For instance, in Problem Set 2, since there were a lot of new classes to implement, adding a docstring explaining the purpose of each class is a good idea. Including a docstring means the specification you've written can be accessed by those who try to create an instance of your class.

## 3. Changing collections while iterating over them

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. This means that if you modify the list you're looping over, the indexes 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)
```

Now, `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.

## 4. Pitfalls of storing custom objects in data structures

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.

## 5. Which data structure should I use?

In Problem Set 2, we asked 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 not ever 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}.

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. :)