CS 16 Python coding conventions

Coding conventions make your code easier to read and debug. Thus, for both our and your benefit, we would like you to use the conventions outlined in this document for your Python code[1].

Startup

1. Import one thing at a time:

```
import imaplib
import stack
from queue import enqueue
```

Spacing

1. Use four spaces instead of tabs. You can set this in Atom by going to `Edit` > `Preferences` > `Tab Length` (under the `Settings` tab). This is particularly important because not only is improperly indented code difficult to read, but python is whitespace sensitive and your code may not work if you don't follow this guideline.
2. Use two blank lines before any class definition or top-level function
3. Use one blank line between any two methods in a class
4. Put blanks around arithmetic operators, but not before commas, or near brackets or parens:

   `(3 + 5 * a[4], 33)` but not
   `( 3 + 5*a [ 4 ] , 33)`

Naming

1. Class names start with a capital letter: `Stack`, `Queue`, `Hashtable` or `HashTable` (the "CapWords" convention)
2. Module names are brief and lowercase: `stack`, `graphalgs`, ...
3. Exceptions are classes, so also use CapWords; they end with the word Error (`"BadDataError"`)
4. Functions are lowercase with underscore separators: `mst_helper`
5. Constants: all caps with underscores: `MAX_OVERFLOW`
6. For private data in classes, start with an underscore: `_my_data`

[1]Actually, we insist on it. If you don't follow the conventions, you'll lose credit on the program.

Comments

1. Write a docstring for every class, module, function, and method. Use triple quotes. End with the triple-quote on a line by itself.

   ```
   def gcd(x, y) :
       """gcd: int * int -> int
       Purpose: Compute the gcd of the integers x and y
       Example: gcd(12, 8) -> 4; gcd(0, -2) -> 2
       """
   ```

2. The documentation for a function begins with the *signature*, i.e., the name, a colon, the argument types, an arrow, and the return type. That's followed by a description of what the function consumes and produces, a description of the purpose, and illustrative examples, especially cases in which the reader might have doubts about the correct output. The types in the signature may be annotated with brief descriptions:

   ```
   def foo(name, age) :
       """foo: str * int [age in years] -> str [birthday greeting]
       Purpose: Generate a birthday greeting by name and age.
       Example: foo("Fred", 21) -> "Happy 21 birthday, Fred"
       """
   ```

3. In Python, variables can hold items of an arbitrary type. While this can make for natural coding, it is also makes it easy to give methods unexpected input. For this reason it is important to be explicit in your method signatures as to what your function produces and consumes. If a method is supposed to receive an object of arbitrary type use `any` (Note: this is not the same as `Object` because this does not include things like `int` and `float`), or if it consumes a specific type try to use standard names for things (`bool` rather than `Boolean`, for instance). Also, if a method does not produce anything, it is designated with a ".".

   ```
   def checkedAdd(name, isCat, myStack) :
       """push: any * bool * stack-> .
       Purpose: pushes any and bool onto the input stack in that order
       Example: checkedAdd("Alice", False, myStack) -> the string "Alice"
       and the bool False are pushed onto myStack in that order
       """
   ```

Runtime Expectations

If we specify a runtime, you must meet that runtime. If we do NOT specify a runtime, you should figure out the most efficient way possible to do the problem. In this case, "most efficient" means big-O. However, you should also consider the constant factor–within reason. This means that you don't need to stress about tweaking your code for the tiniest changes to make it a bit more efficient, especially if it will make your code less clear or concise, but you also shouldn't do obviously inefficient things when there is a much more efficient way that is just as easy to implement as is the less efficient way. For example, if we ask you to write a function that takes as input an integer i and outputs one plus that integer, your function should return `i + 1` as opposed to something unnecessarily more complex (and with a bigger constant factor), such as `i + sqrt(0 + 7 - 7 + 1^2)`. Both of those would return the correct answer, and both would be O(1), but clearly the second has an unnecessarily large constant factor. Note that we will not necessarily tell you what the best big-O runtime is. Sometimes, it's part of the problem to figure that out yourself.

Testing Expectations

1. If you test your code, you are much more likely to fix bugs in it and hand in 100-percent correct code.

2. If you don't hand in (good) test cases, we may dock points– even if your code works perfectly! What qualifies as a set of "good" test cases, you ask? A good rule of thumb is that test cases should cover both typical and edge-case functionality of your code. This means, among other things, that your test cases should "cover" all of your code (for which we have provided the handy "coverage" module). However, be aware that if you only write test cases according to coverage, you might make a mistake– for example, you may not have accounted for a certain edge case when writing your code, in which case coverage will NOT tell you that you messed up. Also note that there is no "magic number" of test cases that you should hand in. We want your test cases to demonstrate that your code is 100-percent correct, and depending on the complexity of the problem you are asked to solve and/or how much functionality each one of your test cases covers, we can't assign a number to that.

Python's Built-in Utilities

Exercise common sense when using built-in Python methods, data structures, and classes. In general, the rule of thumb says "if using this built-in means that I don't write all of the code that was the point of this problem," then you shouldn't use it. If, on the other hand, using that built-in function does not oversimplify the problem, then feel free to use it. For example, Python has a built-in sorting method that uses an algorithm called "Timsort." If we asked you to implement Timsort in Python, it would NOT be acceptable to use the built-in Python sorting function. Also, keep in mind that it can be temping to use Python built-ins in such a way that the code actually becomes more complicated or slower than it would be otherwise, and try not to fall into this trap. (Yes, you could lose points.)