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 Kate by going to `Settings` > `Configure Kate` > `Indentation`. 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.

Underscores before and after function names

These rules also apply to instance variables:

1. Override a built-in function with double underscores before and after the function name:
   `def __str__(...)`
2. No underscores is equivalent to Java's public methods:
   `def foo_bar(...)`
3. One underscore before the function name is still a public method, but is the programmer's way to indicate that it should be considered private:
   `def _bananas(...)`
4. Double underscores only before a name "mangles" the method name so that it is private to that class. What the interpreter actually does is replace `__functionname` with `_classname__functionname` to prevent name conflicts, but really you could still access it if you wanted:
   `__yippee(...)`

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
    Consumes: two integers, x and y, not both zero
    Produces, an integer, the gcd of x and y
    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]
    Consumes: a string and an integer
    Produces: a string
    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-> .
    Consumes: an arbitrary type, a bool, and a stack
    Produces: nothing
    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
    """
```