I shall not mutate my program, I shall use correct default values, and override them by passing appropriate command line parameters for testing

An important part of good software engineering practices is not hard-coding "magic numbers" into the problem which cannot be changed except through recompilation. By allowing these constants to be specified at startup time, with a configuration file or command-line arguments, the program can be considerably more flexible in its use patterns, development effort is saved if a parameter needs to be changed, and the code becomes easier to understand. At the same time, people who are using the program may not wish to specify their own values for every parameter, so the program should transparently provide default values which are used instead.

Fortunately, Python makes it fairly easy to implement this. The first thing to do is allow the program to take in arguments from the command line, so if we originally had a program like this:

```
# Constants
foo = 5
bar = 42

class Processing(Thread):
      # do stuff

thread = new Processing()
thread.start()
```

we could allow foo and bar to be specified at the CLI by changing it to

```
import sys

foo = sys.argv[1] # argv[0] is name of program, actual args start at 1
bar = sys.argv[2]
```

so further references to foo and bar will use the arguments specified on the command line (in Python, sys.argv is a list with one element for each command-line argument. Because Python has an operator to determine the length of the list, it is not necessary to have a separate variable given by the runtime, as in C).

This solution still doesn't allow for default values, though. There are a number of ways this could be implemented; which one is best would depend on the exact nature and requirements of the program. For example, in a program allowing dozens of arguments, simply hardcoding the variables to a given index is not practical. However, for a program with few, or one, variable to be specified in this way, a very simple and readable solution that can be implemented with the minimum of additional code would be to just check the number of arguments passed and assign accordingly, like this:

```
import sys

foo = sys.argv[1] if len(sys.argv) > 1 else FOO_DEFAULT
bar = sys.argv[2] if len(sys.argv) > 2 else BAR_DEFAULT
```

which uses Python's ternary operator to find if the user passed in arguments. For larger numbers of arguments, this approach doesn't scale. An alternative solution might be to have an array filled with default values, then loop over sys.argv and assign global variables to elements in that array until it runs out, then continue assigning them from the defaults array.

Another solution, better when "non-programmers" will be using the utility or where there can be large numbers of arguments of different types, which can be invoked independently of each other, is to use command-line switches, instead of simply passing the arguments one after the other, resulting in program invocations looking like "python server.py -t 90" to specify a timeout of 90 seconds. Python provides the "getopt" library for this purpose, example usage is demonstrated below:

```
import sys
import getopt

foo = FOO_DEFAULT
bar = BAR_DEFAULT

try:
      opts, args = getopt.getopt(sys.argv[1:], "f:b:", ["foo=", "bar="])
      for k, v in opts:
            if k == "-f" or k == "--foo":
                  foo = int(v)
            else if k == "-b" or k == "--bar":
                  bar = int(v)
except:
      print "Error, unknown flag"
```

This approach is much easier to scale up to as many variables as needed, does not require that they be passed in any particular order, and it makes it much easier for users of the program to understand what the parameters they are passing mean, since each switch can be documented (in the program's man page, for instance). With all of these methods in place for passing custom parameters values, there is no excuse for hardcoding them!