New York University
School of Continuing and Professional Studies
Division of Programs in Information Technology

Advanced Python Code Quality

Your study of Code Quality should begin with the Introduction to Python Code Quality document from the Introduction to Python course, and continue through this document. Please familiarize yourself with the points contained in each and send me any questions.

Your homework solutions as well as your comments on other students' solutions must take into consideration the points made in both documents.

(Please note that there are many occasions in which these strictures are not appropriate -- specifically when you're writing short programs, programs that may never be modified again, or you're under some time pressure to develop a working script. In these instances you should consider it perfectly acceptable to ignore one or more of these points. For this class, however, they are always required.)

PART I (to be reviewed for Session 1 of Advanced Python)

1. Overall structure for programs. Here is a tip calculator, with individual steps divided into functions. Also note the use of constants, the **usage()** function, the **main()** function, and the **if __name__ == '__main__':** gate.

```
#!/usr/bin/env python3

"""
    tip_calculator.py -- calculate tip for a restaurant bill
    Author:  David Blaikie dbb212@nyu.edu
    Last modified:  9/19/2017
"""

import sys              # part of Python distribution (installed with Python)
import pandas as pd     # installed "3rd party" module
import myownmod as mm   # "local" module (part of local codebase)

MSG1 = 'A {}% tip (${}) was added to the bill, for a total of ${}.'
MSG2 = 'With {} in your party, each person must pay ${}.'

USAGE_STRING = """Error:  {}

Usage:  tip_calculator.py   [total amount] [# in party] [tip percentage]"""


def usage(msg):
    exit(USAGE_STRING.format(msg))


def validate_normalize_input():
    """ verify command-line input """
    if not len(sys.argv) == 4:
        usage('please enter all required arguments')
```

```
    try:
        bill_amt = float(sys.argv[1])
        party_size = int(sys.argv[2])
        tip_pct = float(sys.argv[3])
    except ValueError:
        usage('arguments must be numbers')

    return bill_amt, party_size, tip_pct


def perform_calculations(bill_amt, party_size, tip_pct):
    """ calculate tip amount, total bill and person's share """

    tip_amt = bill_amt * tip_pct * .01
    total_bill = bill_amt + tip_amt
    person_share = total_bill / party_size

    return tip_amt, total_bill, person_share


def report_results(pct, tip_amt, total_bill, size, person_share):
    """ print results in formatted strings """

    print(MSG1.format(pct, tip_amt, total_bill))
    print(MSG2.format(size, person_share))


def main():
    """ the main event """

    bill, size, pct = validate_normalize_input()
    tip_amt, total_bill, person_share = perform_calculations(bill, size,
                                                             pct)

    report_results(pct, tip_amt, total_bill, size, person_share)


if __name__ == '__main__':            # 'main body' code
    main()
```

a. string comment at very top: this "floating" triple-quoted string must be the first statement in the script. This string can be multi-line or single line. There are a number of styles that can be used in this string, at your discretion.

b. **import** statements, one module per line, in the following order:
   i. modules from Python distribution (**sys, random, datetime**, etc.)
   ii. modules from installed library imports (**pandas**, **flask**, etc.)
   iii. project-specific modules (modules that are part of the current project)

c. constant values: these are global values that do not change during program execution

d. function definitions (**def** statements): these usually make up the bulk of your code

e. **main()** function: near the bottom, this function can be seen as the "controller" for the others

f. **if __name__ == '__main__':** at the bottom, this **if** statement can contain several statements, but oftentimes only contains a single call to **main()**

2. Code should be structured around user-defined functions. Short functions, or coding "units", divide our program's actions into discrete steps that we can test, call selectively or repeatedly, and turn on and off as needed.

   When you find yourself performing an action that comprises 3 or more lines, consider a function. (However, there is no rigid standard -- this should be done at your discretion.)

   a. Use functions to organize your code (and your thinking). Few people can hold more than several lines of logic in mind at one time. Dividing code into functions helps focus our thinking on small areas of code at one time.

   b. Use functions to avoid repetition. Obviously placing code inside a function means we can execute the same code repeatedly without having to repeat the code in multiple places in our program.

   c. Use "versatile" functions (with optional "keyword" arguments) to avoid repetition. Sometimes our program might do the same thing multiple times, but with slight variations. Adding optional "keyword" arguments, e.g. **reverse=False** can help us avoid repetition as well.

   d. Create "pure" functions to help avoid variable name clashes and produce more reliable code. A "pure" function is one that has no "side effects", i.e. one that doesn't modify or read outside (global) variables (reading from *but not writing to* globals is also acceptable; *writing to* global variables inside functions is a big transgression). Pure functions work only with input that has come to it as arguments, and produce output and modified values only through its return values. Pure functions can also be easily tested in isolation. It's not always practical to code only pure functions, but they should be favored when possible.

   e. Commenting each function. Make sure to include a "floating" string (i.e., not assigned to a name) as the first statement inside each function that describes how the function works. (There are some conventions for describing each function's argument(s) and return value(s), but following this convention is not necessary.)

   f. Literal values defined inside functions. *Literal values* are values apart from variables, like numbers or quoted strings. Many of these values are configuration values -- for example, the number of header lines in a file (i.e., the number of lines to skip before reading a file's data). Configuration values should be defined as constants at the top of the script, or may even be defined in a separate file. Placing them in these places allows the maintainer of a codebase to easily change the value without having to dig through the code for it. They should not usually be defined inside a function.
   One exception that I personally make to this rule is to use literal error strings when raising exceptions or calling **exit()** with an error message.

3. Proper use of variables. As described in the discussion of functions above, think of your variables along these lines:

    a. local variables: variables defined and used inside functions, including arguments to functions
    b. globals: with a few exceptions, all globals should be constants. Constants are not expected to change during program execution; they are denoted with ALL_CAPS names.
    c. non-constant globals: these should be rare and *very clearly understood within the context of the program*. One good example is the **app** variable within Flask -- it is used throughout the program so it is defined as a global, although it may change as the program progresses. This is considered acceptable only because it is central to any Flask application and thus is well known by any Flask developer. Other non-constant globals should be similarly understood by the reader / user of the code.
    d. literal values: as noted above, literals are bare values (like strings and numbers) that appear in our code. They should almost always be defined as constants at the top of the script. One common exception is error messages; I personally prefer to define these in the place where they are being used.

4. Be concerned with potential errors in program input as well as function input. This can be applied to the program as a whole (did the user include valid arguments when running the program?) as well as any given function (does the function raise an appropriate exception when bad input is given? Is the error message helpful to the user?). Which errors to check for is at your discretion; some useful rules of thumb are:

    a. all user input to the program (i.e., command-line arguments) should be checked for errors, with descriptive error messages resulting from bad input
    b. arguments to module or library functions (that is, functions designed to be reused in multiple scripts) should usually be checked for errors (since they are often used by other programmers)
    c. arguments to internal functions (i.e., functions for this program only) should either be checked, or at least considered for exceptions that may be raised if they are called incorrectly

5. Prefer **try:** blocks to detect errors. "It's better to beg for forgiveness than ask for permission" -- the Pythonic approach to error handling (preferred, though not the only approach) is to let an error occur and trap it in a relevant **try/except** block, rather than run tests on values to see whether the error might occur.

    Exceptions also have the virtue of passing back through each function call that led to the exception, so it's possible to handle the exception at any point along the "call stack".

    [continued]

instead of this:

```
if len(sys.argv) != 3:
    exit('please enter 2 arguments')
if not sys.argv[1].isdigit() or not sys.argv[2].isdigit():
    exit('please enter 2 numbers')

arg1 = int(sys.argv[1])
arg2 = int(sys.argv[2])
```

try this:

```
try:
    arg1 = int(sys.argv[1])
    arg2 = int(sys.argv[2])
except (IndexError, ValueError):
    exit('please enter 2 numbers')
```

6. Prepare a **usage()** function that explains how the program should be executed. The function can be the "go to" for errors. It can accept an error message argument to be displayed the user.

```
def usage(msg):
    usage_msg = """error:  {}

usage:  get_done.py [num]

[num] must be a positive integer"""

    exit(usage_msg.format(msg))
```

(note that since triple-quoted strings include newlines and spaces, the above string wraps to the left margin so additional spaces are not included)

7. Reducing # of code lines without sacrificing clarity. One of Python's strengths is in its clarity, and most design choices tend to seek clarity above other considerations. (Speed and efficiency may also be primary goals depending on the situation.) Obviously if shortening your code makes it less readable it's probably not a worthwhile choice. But some code shortening can actually enhance readability.

   a. chaining method calls

   instead of this:

   ```
   x = ['hello', 'world!']
   y = ','.join(x)
   z = y.title()            # Hello, World!
   ```

   try this:

   ```
   z = ','.join(x).title()
   ```

b. nesting function and/or method calls

instead of this:

```
uinput = input('Please enter a number: ')
uint = int(uinput)
```

try this:

```
uint = int(input('Please enter a number: ')
```

…which you may want to wrap in a **try:** block:

```
try:
    uint = int(input('Please enter a number: ')
except ValueError:
    exit('need an int!')
```

c. subscripting a method/function call

instead of this:

```
fh = open('myfile.txt')
lines = fh.readlines()
first_two = lines[0:2]
```

try this:

```
first_two = open('myfile.txt').readlines()[0:2]
```

note that the latter syntax prevents the file from being closed during program execution (usually not a problem unless the program stays running for long periods of time or opens many files) Many coders, along with the general consensus of the community, suggest that you use the **with** context manager with files:

```
with open('myfile.txt') as fh:
    first_two = fh.readlines()[0:2]
```

d. using a dict instead of chained "ifs"

instead of this:

```
mylist = ['a', 'b', 'c']

if uinput == 'first':
    x = 0
elif uinput == 'second':
    x = 1
elif uinput == 'third':
    x = 2

print(mylist[x])
```

try this:

```
chooser = {
    'first':  0,
    'second': 1,
    'third':  2
}

print(mylist[chooser[uinput]])
```

e. using multi-target assignment

instead of this:

```
x = 5
y = 10
z = 20
```

try this:

```
(x, y, z) = 5, 10, 20
```

this works well with list unpacking:

```
line = 'this,that,other'
x, y, z = line.split(',')
```

this can also be used effectively in function return values as well:

```
def myfunc():
    return 1, 2, 3

a, b, c = myfunc()
```

8. No-nos / gotchas
   a. never use an index counter to step through a list or sequence: loop through directly
   b. never use an index counter to step through a specific portion of a list: use a slice
   c. never use a counter to count while looping: use **enumerate()**
   d. never use a mutable as a default function argument
   e. never use a bare **except:** statement
   f. never use **from modname import ***

[continued]

PART II (to be reivewed for Session 2 onwards in Advanced Python)

9. Use exceptions (raised by Python or generated by the **raise** statement) to signal errors occurring in a function; handle errors in the main body (Exceptions are introduced in Session 2). If something goes wrong in our program, we have a few options:
   a. if the error is detected by Python and an exception is raised, we can allow the exception to terminate the program, displaying the exception
   b. if the error is detected by Python and an exception is raised, we can trap the error in a **try:** block and do something in response -- call **exit()** with a message or handle it some other way
   c. if our program detects the error (let's say, by testing a value to see if it's valid), we can call **exit()** ourselves or handle it some other way
   d. if our program detects the error, we can **raise** a fresh exception and allow Python to terminate the program, displaying the exception
   e. we can even **raise** an exception in one part of the program and trap it in another, such is the flexibility of error handling with exceptions

   All of the above options are available to us, and are valid approaches.

   Since we generally prefer that the (non-programmer) user not see exceptions directly, the suggested approach is to have functions signal errors using exceptions (whether generated by Python or raised by the code using **raise**) and use the **exit(error_msg)** function to signal errors in the main body. This can be done by trapping all exceptions in the main body, and then using **exit()** to display the final error message:

```
""" get_done.py -- demo program """

import sys

DEBUG = False                   # constant can be set manually as desired

def do_something():
    try:
        arg = sys.argv[1]       # might raise an IndexError
        int_arg = int(arg)      # might raise a ValueError
    except ValueError:
        raise ValueError('please enter a positive integer')
    if not int_arg > 0:
        raise ValueError('please enter a positive integer')

def main():
    do_something()

if __name__ == '__main__':
    try:
        main()
    except Exception as obj:    # traps most exceptions
        if DEBUG:               # if DEBUG == True,
            raise               # re-raise same exception
        else:
            exit(obj.args[0])   # exit with error msg from exception

    exit(0)                     # exit without error
```

10. Use of power tools for simple operations

    a. list comprehension in place of simple loops
    b. if/else "ternary" expression
    c. **range()** for cycling through sequential integers
    d. **enumerate()** for counting items while looping