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

Python Code Quality

Clarity and readability are important (easter egg: type **import this** at the Python prompt), as well as *extensibility*, meaning code that can be easily enhanced and extended.

These coding standards and recommendations are required for your work in the course. You'll be required both to use them in your own code and point them out to others when reading their code.

Style Standards

1. Variable names: all lowercase, with underscores between any words

   Please do not use **camelCase**, but instead **words_with_underscores**.

2. Use 4-space Indents, no tabs

   Atom takes care of this for us - inserting 4 spaces whenever we use the [Tab] key. If you are using a different editor, you must configure it to do the same. See me for details.

3. One space on either side of operators

```
var = aa + bb     # note one space on either side of = and +
```

   This might seem picky, but it's all for readability, which is extremely important in the professional world.

4. No space between function name and argument list (parentheses)

```
string_length = len('hello')
```

Naming and Clarity

5. Use descriptive variable names; never use the plural for a loop variable

   At the outset, we will be using variable names that have no signficance -- this is done to aid in identifying the Python "parts of speech". However starting in the 3rd session,we will require that variable names be descriptive. If a variable holds the user's first name, it should be called **first_name**, **user_fname**, **fname**, i.e. something descriptive. If you call first name **name1** and last name **name2** you're not being as clear. If you call first name

aa then the variable's identity becomes opaque to the reader (until he or she traces it back in your code).

In particular there is one particular misnomer that I'd like you to avoid:

```
fh = open('datafile.txt')

for lines in fh:          # please use singular ('line')
    print(lines)
```

We use the **for** construct to loop through a file line-by-line, but the *control variable* **lines** is supposed to be only one element at a time. Therefore it's misleading to call it **lines** when it's really just one line.

6. Do Not Use the Built-In Names for Your Variables

I'd prefer that Python prohibit the use of these names, but it does not. **list, dict, file, len, float, int, str** and other builtin functions can be overwritten if we assign to these labels.

```
>>> len('hello')
5
>>> len = 0.00001
>>> len('hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
```

7. Use the string **format()** method to combine values with strings.

Some folks prefer to stick to string concatenation or use commas when printing -- instead, use the string **format()** method to combine values (numbers or strings) with strings. (This is not a style convention, but I feel something you should be familiar with - outside of class you are free to use whatever you prefer.)

8. Comment your code only when needed for clarity, and please remove diagnostic code

Most code if clearly written can be read and understood easily, but some code is by nature a little harder to read (such as list comprehensions, lambdas, code involving multidimensional structures, etc.). In these cases it makes sense to add a comment to explain what's happening.

Some students like to comment each of their code steps. One method is to start with a comment outline and then write code to fulfill each comment. This is fine for draft work, but please remove unnecessary comments before submitting.

9. Put a String Comment at the top of the script with title description and author

   This is a great habit that you will thank yourself for acquiring, and it will help me in reviewing as well. No matter how short, every script should have an identifier that will make it instantly recognizable to you when you open the script later (otherwise you are forced to start reading it to get a sense of what it is supposed to be doing).

```
#!/usr/bin/env python

"""david_blaikie_1.2.py -- calculate a tip based on user input"""
```

   This "docstring" should be the first line of your code. Such strings can also be used for automatically generated documentation. Customarily we use a "triple-quoted string" for this purpose, although it can be a single- or double-quoted string as well.

Code Organization

10. Conceptualize the logical "steps" of your code.

    Consider that every program takes discrete steps; separate your code visually (with blank lines) so these steps can be quickly identified by the reader as well. Note: keep in mind that these are guidelines and don't need to be followed in every case; you'll begin to develop good judgment about what's readable, what's well organized, etc.

    a. Take input. Most programs begin by taking user input or reading command-line input, which tells the program how to function.

    b. Validate and/or "normalize" input. If the input is coming from the user, it has to be checked to see if it is valid. It may also need to be converted into a usable form -- for example, if the user is asked to input an integer, the string needs to be converted to an integer. If the user is asked to input a filename, the program must determine if the file exists and can be read.

    c. Process or compute the data. This is the main event - once everything has been normalized and is ready to go, then we go through the computations -- summing, aggregating, etc.

    d. Output the results. Finally, we inform the user of the results by printing the results of the computation.

    Again, these steps don't always apply -- but if you can avoid mixing them (for example, normalizing while you're in the middle of computation) then your code will be much clearer and more straightforward.

11. Use Blank Lines Sparingly, but Use Them to Separate Logical Steps

Think of your code as an essay with paragraphs. Note in the code above that each step is comprised of a few statements and these are single-spaced, and that between steps we have placed a blank line. This indicates to the reader what the steps are and how best to read the code. This sort of logical separation greatly aids readability.

12. "Flat is better than nested"

(Note that this will become important starting with the 2nd session.) Nested code creates logical dependencies, which means that one section of code is dependent on the other to be fully understood by the reader. If the dependencies are not needed, we have unnecessary logical complexity. Compare these two code snippets -- in the first below, notice that the computational **while** loop (bolded below) is logically dependent on the **if** and the other **while**, even though it doesn't need to be. Also note that the **if** and **else** are separated, requiring the reader to unite those two pieces in order to make sense of it.

```
counter = 0
while True:
    max_count = input('enter an int:  ')
    if max_count.isdigit():
        while counter < max_count:
            print(counter)
            counter = counter + 1
        print('done')
    else:
        print('sorry, bad input, try again')
```

In a flatter (i.e., less indented) version of the same code below, each step (take and validate user input; convert input to int(); perform the computation) is separate, flatter and easier to read. It is also easier to maintain. The mind doesn't need to take in the whole of the code in order to comprehend the logic - instead one can understand the first step, then move on to the next step without having to remember anything.

```
counter = 0
while True:
    max_count = input('enter an int:  ')
    if max_count.isdigit():
        break
    print('sorry, try again')

max_count = int(max_count)

while counter < max_count:
    print(counter)
    counter = counter + 1
```

13. Avoid Repetition and Avoid Using Intermediate Structures

This one is harder to explain in specific, but I have found that many students get used to using one container structure (in particular, **list** is a favorite) and so try to do everything with that structure. They also like to break down steps into multiple loops, by looping through one list, modifying an element and appending it to a new list, then looping through

the new list, modifying an element and appending the element to a third list, etc., when the whole thing could be done in one list, by performing successive operations in the one loop.

In this example, **stripped_lines**, an intermediate structure, is not necessary:

```
stripped_lines = []
ids = []

for line in open(filename).readlines():
    line = line.rstrip()
    stripped_lines.append(line)

for line in uppercased_lines:
    elements = line.split(':')
    ids.append(elements[-1])
```

These two loops could be consolidated into one loop:

```
ids = []
for line in open(filename).readlines():
    line = line.rstrip()
    elements = line.split(':')
    ids.append(elements[-1])
```

14. Put Imports at the Top, Followed by Functions, Followed by Code

This will come into play as we begin to use the **import** statement and to create functions.