CMSC 201 Fall 2017

Python Coding Standards

The purpose of these coding standards is to make programs readable and maintainable. In the "real world" you may need to update your own code more than 6 months after having written the original – or worse, you might need to update someone else's. For this reason, every programming department has a set of standards or conventions that programmers are expected to follow.

Neatness counts!

At UMBC, the following standards have been created and are followed in CMSC 201.

Part of every project and homework grade is how well these standards are followed.

It is your responsibility to understand these standards. If you have questions, ask any of the TAs or the instructors.

Naming Conventions

- Use meaningful variable names!!
  - For example, if your program needs a variable to represent the radius of a circle, call it **`radius`**, not **`r`** and not **`rad`**.
  - The use of obvious, common, meaningful abbreviations is permitted. For example, 'number' can be abbreviated as **`num`** as in **`numStudents`**.
  - The use of single letter variables is *forbidden* except in loops.

- Begin variable and function names with lowercase letters.

- Names of constants should be in all caps with underscores between words.
  - *e.g.*, **`EURO_TO_USD = 1.20`** or **`MAX_NUM_STUDENTS = 100`**

- Separate "words" within identifiers with underscores or mixed upper and lowercase.
  - *e.g.*, **`grandTotal`** or **`grand_total`**
  - Be consistent! If you choose to use mixed case (also known as "camel case"), always use mixed case. If you choose to use underscores to separate words, always use underscores.

- Do *not* use global variables! Use of global variables is *forbidden*.

Use of Whitespace

The prudent use of whitespace goes a long way to making your program readable. Horizontal whitespace (spaces between characters) will make it easier to read your code. Vertical whitespace (blank lines between lines of code) will help you to organize it.

- Use blank lines to separate major parts of a program or function.
- Indentation should be 4 spaces long. Using Tab in emacs will accomplish this.
- Use spaces around all operators.
  - For example, write **`x = y + 5`**, NOT **`x=y+5`**.
- Lines of code should be no longer than 80 characters (the default size of an emacs window). Code that "wraps" around a line is difficult to read.

Line Length

Avoid lines of code longer than 80 characters, since they're not handled well by many terminals, and often make your code more difficult to read. If a line of your code is longer than 80 characters, you may be doing too much in one line of code, or you may have nested too deep with loops and conditionals.

If you have a line of code that is unavoidably longer than 80 characters, you can continue the code on the next line by putting a "\" (backslash) after a breakpoint in the code (*e.g.*, after a "+", after a comma, etc.). If you're using emacs, it will automatically indent the rest of the line of code following the backslash.

For example:

```
     choice = int(input("Please enter a number between " + str(min
n) + " and " + str(maxx) + ", inclusive: "))
```

Can become:

```
     choice = int(input("Please enter a number between " + \
         str(minn) + " and " + str(maxx) + ", inclusive: "))
```

Use of Constants

To improve readability, you should use constants whenever you are dealing with hard-coded values. Your code shouldn't have any "magic numbers," or numbers whose meaning is unknown. Your code should also avoid "magic strings," or strings that have a specific use within the program (*e.g.*, choices a user could make such as "yes," "STOP", etc.).

For example:

```
total = subtotal + subtotal * .06
```

In the code above, `.06` is a magic number. What is it? The number itself tells us nothing; at the very least, this code would require a comment. However, if we use a constant, the number's meaning becomes obvious, the code becomes more readable, and no comment is required.

Constants are typically declared near the top of the program so that if their value ever changes they are easy to locate to modify. Constants may be placed outside of the `main()` function – this makes them global constants, which means everything in the file has access to them. (Global variables are only allowed for constants!)

Here's the updated code:

```
TAX_RATE = .06

def main():
    # lots of code goes here
    total = subtotal + subtotal * TAX_RATE
    # other code goes here
    print("Maryland has a sales tax rate of", TAX_RATE, "percent")
main()
```

Comments

Programmers rely on comments to help document the project and parts of the project. Generally, we categorize comments as one of three types:

1. File Header Comments
2. Function Header Comments
3. In-Line Comments

1. File Header Comments

Every file should contain a comment at the top describing the contents of the file and other pertinent information. This "file header comment" MUST include the following information.

- The file name
- Your name
- The date the file was created
- Your section number
- Your UMBC e-mail address
- A brief description of the contents of the file

For example:

```
# File:    proj1.py
# Author:  Taylor Jones
# Date:    11/16/2017
# Section: 04
# E-mail:  tjones1@umbc.edu
# Description:
#    This file contains python code that will simulate a run of
#    Conway's "Game of Life".  It allows the user to choose
#    which cells to turn on, and how many iterations it runs.
```

2. Function Header Comments

Every single function must have a header comment that includes the following:

- Function name
- A description of what the function does
- Input (name, type and short description)
- Output (name, type and short description)

For example:

```
# circleArea() calculates the area of a circle from the radius
# Input:       radius; an int or float of the circle's radius
# Output:      area;   a float of the circle's area
def circleArea(radius):
    area = PI * (radius ** 2)
    return area
```

The only function that does not require a header comment is the `main()` function.

3. In-Line Comments

In-line comments are comments within the code itself. They are normally comments for the line(s) of code directly below them.

Well-structured code will be broken into logical sections that perform a simple task. Each of these sections of code (often starting with an '`if`' statement, or a loop) should be documented.

- Any "confusing looking" code should also be commented.
- Do not comment every line of code. Trivial comments (*e.g.*, `# increment x`) clutter up your code and are worse than no comments at all.
- In-line comments are used to clarify ***what*** your code does, ***not how*** it does it.

An in-line comment appears above the code to which it applies. It is also *indented to the same level* as the code it is a comment for; comments that are not correctly indented make the code less readable.

For example:

```
# go over the list of numbers given by the user
for num in userNumList:

    # if it's odd, print it, if it's even, do nothing
    if num % 2 == 1:
        print(num)
```

Built-In Functions

Python has many useful built-in functions that easily let a programmer perform a variety of tasks. Unless specified by the assignment, you are not to use any of the built-in functions we have not covered in class.

Using a built-in function to solve a problem by having Python do the work for you does not show that you have mastered the concepts behind it, and hence does not fulfill the assignment. If you are unclear on whether you are allowed to use a built-in function, ask a TA or instructor.

Break and Continue

Using **`break`**, **`pass`**, or **`continue`** is not allowed in any of your code for this class. Using these statements damages the readability of your code. Readability is a quality necessary for easy code maintenance. Using of any of these will lead to an immediate deduction of points.