CMSC 201

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. They are based heavily on the Python Enhancement Standard (PEP) 8 for code style. This is the standard followed in industry for Python source code.

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 **`num_students`**.
    - 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 (function names and variable names) with underscores *eg.*, **`grand_total`** (Fun fact: this is called snake case because it makes the variables look kinda like snakes!)

- 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 a single blank line to separate major parts of a function.
- Use two blank lines to separate functions.
- 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 79 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 79 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 79 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: "))
```

File Organization

The Python file itself should be organized to make it easy to find specific functions or information. From top to bottom, your Python files should have:

1. a full header comment
2. any constants used by the program, blank if there are no constants
3. all function definitions, blank if there are no functions

4. the line: **`if __name__ == "__main__":`**[1]
5. the code that will execute when the file is run, indented once

Sample Code

```
"""
File: lab2.py
Author: YOUR NAME
Date: THE DATE
Lab Section: YOUR LAB Section
Email:  YOUREMAIL@umbc.edu
Description:  This program shows the layout of code in a Python
file, and greets
the user with the name of the programmer
"""

TOTALLY_NORMAL_HUMAN_NAME = "Yxcronor"


def where_is_the_cat(cat, location_tried):
    """
    A function to look for the cat
    :param cat: the name of the cat to look for
    :param location_tried: the integer IDs I'll look in
    :return: the integer ids for places I looked
    """
    print("I don't know,", TOTALLY_NORMAL_HUMAN_NAME)
    return location_tried


if __name__ == '__main__':
    # introduces the programmer
    print("Hello, my name is perfectly reasonable. \
          Say, do you know where the cat is?")
    where_is_the_cat("Jules", [1, 2])
```

[1] This line is not required for the python code to execute, but we require it for reasons that may not be clear until near the end of the course. It is a standard widely followed in Python development.

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 before the **`if __name__ == "__main__":`** statement – 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

if __name__ == '__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")
```

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) and (2) will use triple quotes (""") A.K.A. docstrings. (3) will use pound signs (#).

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: lab2.py
Author: YOUR NAME
Date: THE DATE
Lab Section: YOUR LAB Section
Email:  YOUREMAIL@umbc.edu
Description:  This program shows the layout of code in a Python
file, and greets
the user with the name of the programmer
"""
```

2. Function Header Comments

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

- A description of what the function does
- :param parameter_name (name, type and short description)
- :return: description of what is returned

For example:

```
def where_is_the_cat(cat, location_tried):
    """
    A function to look for the cat
    :param cat: the name of the cat to look for
    :param location_tried: the integer IDs I'll look in
    :return: the integer ids for places I looked
    """
    print("I don't know,", TOTALLY_NORMAL_HUMAN_NAME)
    return location_tried
```

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 and Functionality

Python has many useful language features, built-in modules, and built-in functions that easily let a programmer perform a variety of tasks. However, due to the introductory nature of this course, you are not permitted to use any Python construct, built-in module, or third-party library that is not explicitly covered in the lecture slides.

You are also not permitted to use anything that has not yet been covered in lecture.

Using a built-in function or functionality 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 we do not show you how to use it in class, you can assume that it is off limits. If you find yourself unsure if you are allowed to use something, please consult with a member of the CMSC 201 course staff for clarification.

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 any of these will lead to an immediate deduction of points.