

# BIFX 502 Programming Assignment Coding Guidelines

## Coding Guidelines – Part 0

These guidelines apply to all programming assignments.

All assignments must be submitted through the assignment link on Blackboard. In general,

- Assignments must be submitted through Blackboard. Assignments submitted via email, hard copy, or other formats will not be accepted.
- Assignments must be submitted on time. Note that *Blackboard does let you submit late submissions, but they are labeled as overdue*. Late work will *not* be accepted (unless you have spoken to me and received prior approval to do so).
- Program file names should include your name and the assignment identifier with a hyphen or underscore separating them:  
`yourlastname_assignmentidentifier.py`  
For example, if I submitted assignment 37, the file name would be `jim_pa37.py`
- Do **not** use spaces in file names.
- Programs should ONLY use Python that we have covered in class (and the textbook) at the time the assignment is given. Do not go out and copy solutions online or get an expert to write the program for you. The purpose is for you to work out the problem. Programs that use code that we have not covered will be assumed to be copied *rather than created by you*, and may be assigned a grade of zero. I may ask you to explain the features that you used.
- For full credit, programs must have valid syntax, correct computations, and correct output format
- Specific requirements will be provided for each assignment.

## Coding Guidelines – Part 1, Code Basics

These requirements apply to all programming assignments.

- **Comments**

Always include file comments.

Place a descriptive comment heading at the top of every `.py` file in your project with your name, the course number, a description of that file's purpose, and a **citation of any sources** you used to help write your program. Assume that the reader of your comments is an intelligent programmer but not someone who has seen this program or assignment before. The header should describe behavior but should not go into great detail about how each piece is implemented. For example,

```
"""
Program:      jim_pa1.py    August 24, 2020
Name:        Carol Jim
Course:       BIFX 502
Assignment:   Programming Assignment 1
Description:  Computes the average of three input values.
"""
```

- Use inline comments to describe complex sections of code  
Avoid unnecessary and trivial comments

#### BAD

```
avg = (score1 + score2 + score3)/3      # add scores and divide by 3
```

ALMOST AS BAD - because well designed variable names should make the comment unnecessary

```
avg = (score1 + score2 + score3)/3      # calculate average
```

#### Whitespace and indentation:

- Use correct, consistent indentation.
  - All blocks of code within a given function or control flow structure must be distinguished from the preceding code by one indentation level.
  - Use 4 spaces per indentation level. IDLE will automatically indent 4 spaces when you hit the tab key.
- Always place a line break after a colon ( : )
- Avoid long lines.
  - Avoid lines longer than 80 characters, including indentation. Make sure to maintain correct indentation when breaking long lines into shorter ones.

#### Variables

- Make smart variable type decisions and utilize type conversions

Store data using the appropriate data types (making sure to use the appropriate literals as well), and make sure to use the type conversion functions (`str()`, `int()`, `float()`) when necessary. Remember, for example, that if you ask the user to input the temperature outside and you want to use that value in a calculation, you will have to convert their input to a float before you can use it.

- Follow variable naming conventions

Name variables using underscore (AKA *snake case*) like `like_this`. Use concise, descriptive names for variables that precisely describe what information they're storing.

- Create one variable per line of code

Never initialize more than one variable in a single statement.

# bad	# good
a, b, c = 7, -43, 19	a = 7
	b = -43
	c = 19

- Use named constants when appropriate

If a particular constant value is used frequently in your code, identify it as a constant, and always refer to the constant in the rest of your code rather than referring to the corresponding literal value. Name constants in uppercase with underscores between words LIKE\_THIS.

```
DAYS_IN_WEEK = 7
DAYS_IN_YEAR = 365
HOURS_IN_DAY = 24
```

## Part 2 – Conditionals and Loops

These requirements apply to Chapters 3 & 4 and later assignments

### Control Statements

- Avoid empty `if` or `else` branches
  - When using `if/else` statements, you should not have ``if`` or ``else`` branches that are blank. Rephrase your condition to avoid this.
- Avoid unnecessary control flow checks

When using `if/else` statements, properly choose between various `if` and `else` patterns depending on whether the conditions are related to each other. Avoid redundant or unnecessary `if` tests. Ask yourself if all of the conditions always need to be checked.

<pre># bad if points &gt;= 90:     print('You got Gold!') if points &gt;= 70 and points &lt; 90:     print('You got Silver!') if points &gt;= 50 and points &lt; 70:     print('You got Bronze!') ...</pre>	<pre># good if (points &gt;= 90):     print('You got Gold!')} elif points &gt;= 70:     print('You got Silver!') elif points &gt;= 50:     print('You got Bronze!') ...</pre>
---	---

- Beware of infinite loops

Avoid writing loops that never stop. An infinite loop will cause your program to never stop executing. Replace infinite loops with loops that terminate.

- Choose the right loop

Consider when it would be more appropriate to use a `while` loop or `for item in list` loop or `for item in range` loop.

## Part 3 – Functions

### These requirements apply to Chapter 5 and subsequent assignments

- Use descriptive names for functions

Give functions descriptive names, such as `discount(price, rate)` or `make_a_star()`. Avoid one-letter names and non-descriptive names, like `x()` or `go()` or `function1()`.

Function names should be all lowercase, with words separated by underscores to improve readability.

- Keep your main program a concise summary

As much as possible, avoid having too much of your program's functionality directly in your `main()` code. When writing your programs, try to break the problem down into smaller sub-problems. Create functions for these individual sub-problems. This makes your program easy to read and forms a concise summary of the overall behavior of the program.

- Minimize redundant code

If you repeat the same code block two or more times, find a way to remove the redundant code so that it appears only once. For example, you can place it into a function that is called from both places.

- Avoid long functions

Each function should perform a single, clear, coherent task. If you have a single function that is very long or that is doing too much of the overall work, break it into smaller sub-functions. If you try to describe the function's purpose and find yourself using the word "and" a lot, that probably means the function is doing too many things and should be split into sub-functions.

- Consider short functions for common tasks

It can be useful to have a short function **if** it wraps another function with some additional control flow or if it contributes to an easier to understand name.

```
# bad
def square_root(x):
    return math.sqrt(x)

# good
def discount(price, rate):
    discount = price - rate*price
    return discount
```

- Write an overall header comment for every function - use the IPO notation described in Module 05, Part 3

- Eliminate redundancy by introducing functions with parameters

If you repeat the same code two or more times that is nearly but not entirely the same, try making a helper function that accepts a parameter to represent the differing part.

```
# bad
x = foo(10)
y = x - 1
print(x * y)
...

x = foo(15)
y = x - 1
print(x * y)

# good
print(helper(10))
print(helper(15))
...

def helper(p):
    x = foo(p)
    y = x - 1
    return x * yx
```