# Coding Tips for Efficiency and Readability

## Introduction

The chances that you will be the only person to ever read the code you write is extremely slim. You may be working on a large-scale project where a whole team of people need to go over code you've written, a potential employer may be looking at a project you've pushed to GitHub, or in the case of this class, a ULA is looking for ways to give you feedback on how to improve your code. In all these scenarios, it is important that people other than yourself understand not only what your code is doing, but also that they can do so with ease. Because of this, we have made this brief style guide to help you write clean code, not only for this class, but for all future projects and assignments.

---

## Naming Conventions

Picking and sticking to a naming convention is key to creating consistent, readable code. The three popular conventions you are recommend to use are camelCase, snake_case, and PascalCase. One reason naming conventions are so important is because of predictability. If you are working on a project and know your function name is "insert at index", knowing the convention saves you time when trying to remember if it was insert_at_index, insertAtIndex or InsertAtIndex. An exception to a single convention is using different conventions for different elements of your code. Classes should start with a capital letter, so you're more likely to use PascalCase for them, whereas with variables you may use camelCase, and your functions/methods use snake_case. What's important is that your naming conventions are obvious and consistent.

```
myString = "The quick brown fox jumped over the lazy dog"
newCap = 10

#snake_case
my_string = "The quick brown fox jumped over the lazy dog"
new_cap = 10

#PascalCase
MyString = "The quick brown fox jumped over the lazy dog"
NewCap = 10
```

---

## Variable Naming

A variable name should be descriptive of the element it represents concisely. A good aim, though not a requirement, is between 10 and 16 characters. Not only should it be easy to write, it should be easy for anyone reading your code to understand what it is representing.

For example, total is short, but is not descriptive. Total what? Hours? Cash on hand? Bees? On the other hand, and yes this is dramatic, totalCashInAfterCashOut is way too long. Not only does it take a long time to write out, it has poor readability.

```
# too short, undescriptive
total = 1000

# too long, hard to remember, read, and write
totalCashInAfterCashOut = 1000

# descriptive but concise
totalRevenue = 1000
```

---

### Function Naming

Function names follow the same general guidelines as variable naming. It should be descriptive enough that, when used in your code, it is clear what that line of code is doing; but concise enough that it is easy to write and read. When working with a Class with private variables, you'll find that you'll need to 'get' information via class methods. Because of this, there will usually be a get method inside a class, maybe more than one. get() on its own is poorly-named as there is no indication as to what the method is getting. get_at_index() is a better example as it short, but tells you exactly what it is doing, which is getting the value at a specified index. All required methods in this class will be pre-named for you; however you are highly encouraged to create helper methods (modularization).

---

### White Spaces

- When indexing or passing parameters into a function, don’t add white spaces between your elements and the parentheses/brackets.
- On the other hand, when writing out equations, make sure to add spaces between your storing variable, equal sign, and the equation.

```
# Bad Practice
self.data[ 3 ]
self.set_at_index( 1, 1 )
z=x+y
x+=1
self.examples( x = x + 1, y = z)

# Good Practice
self.data[3]
self.set_at_index(1, 1)
z = x+y
```

```
x += 1
self.examples(x+1, z)
```

---

### Descriptive Comments and Cluttering Comments

**Every** block of your code should have comments *briefly* describing the purpose of the variables, control flow statements, or calling function directly below. This DRAMATICALLY increases the readability of your code, and will save you time during your debugging process. It will also give your graders opportunities to give *helpful* feedback on your code.

With the above said, relying on comments alone can decrease the readability of your code. Since each line of code is doing something unique, it is important that you name your functions and variables descriptively.

```
def zab():
    # Descriptive comment describing variable assignment
    variableZero = 0
    variableOne = 1
    variableTwo = [foo()]

    # Descriptive comment describing function and loop
    bar(variableZero)
    for x in variableTwo:
        variableOne += faz(x)

    # Descriptive comment describing returning block
    variableThree = 3
    return foo(variableOne, variableThree)
```

```
# Bad Practice
while x != 0:
    y += x
    x -= 1

foo(z)


# Good Practice
# Stores the summation of the value x
while x != 0:
    y += x
    x -= 1

# Prints all the prime values from 0 to y
foo(y)
```

```
# Bad
# This variable is used to store a temporary value
x = z
# squares our temporary value
self.foo(x, y)
# Insert value x into index y
self.bar(x, y)
```

```
# Good
# Inserts an altered value into our array
tempValue = z
self.square_and_divide(tempValue, denominator)
self.insert_at_index(tempValue, idx)
```

---

### Comment Alignment

Although it has already been briefly mentioned above, commenting can either make your code easy to read, or a headache and a half. For the most part, you should comment above blocks of code to briefly describe what it is doing. There may be occasions where leaving a comment right-aligned makes more sense, such as describing variable assignments at the start of a function, and when doing so Python allows you to multi-tab to align comments, making it easier to read.

```
# Bad Practice
for x in arr: #Iterates across a list of nums
    sqr = x*x #squares our value
    print(sqr) # Prints the square of x
    tempArr.append(sqr) # adds squared value to sqrList

# Good Practice
# Prints and adds squared value of all items in arr
for x in arr:
    sqr = x*x
    print(sqr)
    tempArr.append(sqr)

# Situational - use ONLY if the code and the comment are *short*
tempValue = x       # Value to be modified and returned
newCap = y          # Replaces capacity
total = z           # While counter for number of elements in array
```

---

### Docstrings

Not only are docstrings important, **you will lose points if you do not add them.** Documentation in Python is centered around docstrings, so implementing them properly and with thought is the best way to increase the readability of your code.

- A good docstring includes a brief description of the function, parameters, and return.
  - This can be done as shown below, but there are alternative methods.
- A docstring is NOT the assignment's function description.
- They shouldn't stretch into paragraphs, with each element taking up only about a line.
- Each 'section' of the docstring should be separated by a line.

```
# Bad Practice
def foo(arr, y):
    tempArr = []
    for num in arr:
        if num % y != 0:
```

```
            tempArr.append(num)
    return tempArr

# Good Practice
def foo(arr : list[int], y : int) -> list[int]:
    '''
    Create a modified list of values not divisible by y from arr.

    :param arr: an array of integers
    :param y:   integer to divide against for a remainder of 0

    :return:    modified list w/o values divisible by y
    '''
    tempArr = []
    for num in arr:
        if num % y != 0:
            tempArr.append(num)
    return tempArr

# Utilzing Pythons Docstrings
print("__doc__:")
print(mysteryMethod.__doc__)

print("help:")
help(mysteryMethod)
```