Style Guide

ME 498 - Programming Fundamentals for Mechancial Engineers Spring 2023

3/30/2023: Initial Release

4/11/2023: Updated docstring section with additional clarifications

---

This guide outlines style requirements and recommendations for writing well-structured, readable code for this class. This guide is a "living document" and will be updated as needed based on course content, assignments, and student questions. It is recommended that you revisit the style guide before starting on each homework assignment.

Style Requirements (used for grading)

Variable and Function Names

Variable and function names should be descriptive:

- DO use names that describe what the variables or functions represent
  - Avoid generic names like `var1` or `fun2`
- DON'T use single-character variable names, unless:
  - you are following a mathematical/engineering convention (e.g., `g` for gravity, `x` and `y` for x and y coordinates)
  - you are following a programming convention (e.g., `i` for index, or as the iteration variable in a for-loop)

Functions

Technically, you could write an entire program without using any functions. However, this would be an incredibly bad idea. Your full program could end up being thousands of lines long, and/or difficult to debug. It's generally considered good practice to factor your code into functions. Here are a few reasons why:

Functions Reduce Redundancy: Often, you will want the exact same or very similar tasks multiple times in your program. Rather than writing the same code multiple times, which would be redundant, you can factor that code into a function that can be called throughout your program to perform that task. We can even use parameters and returns to create functions that are even more modular to further reduce redundancy. If you are tempted to copy/paste code, you should probably be using a function!

Functions Represent Tasks: Even if code isn't redundant, it can still be a good idea to factor it into a function. Factoring code into functions with distinct tasks also makes it easier to reuse your code. If there's a function where you perform 2 tasks, and then later you want to only perform one of those tasks, you

couldn't call your existing function because you don't want to perform both tasks. It would be better to structure each task into its own function to make it more reusable.

A few things to avoid:

Avoid trivial functions: Trivial functions do so little that their existence is pointless. Functions with just one print statement or one function call are good examples. One-line functions can be non-trivial if you are factoring a common calculation into a functions, for instance, but with functions with so little code, you should generally consider whether or not the function is improving your program.

In [12]:
```
# Trivial function: no need to create a new function, just use print(num)
def printingNumber(num):
    print(num)

# Nontrivial function: prints num with some formatting that maybe we will want to reuse.
def printRoundedNumber(num):
    num = round(num, 2)
    print('The provided number rounded to 2 decimal places is', num)
```

Avoid unnecessary parameters and returns: Functions only need to return a value if you plan on catching and using that value when you call your function. Similarly, you should only pass in parameters that you need. Parameters that you never use in your function or whose value could be calculated from other parameters are unnecessary. If you pass a parameter into your function but never use the value passed in (i.e. you immediately set its value to something else), you might want to consider whether you need that parameter at all, or if that value could be a local variable.

Avoid using global variables to bypass parameters: In Python, variables defined in the main body of your script are visible everywhere in your script. This includes inside functions, unless a local variable is defined with the same name (see the Lecture 02 - Functions notes for more information). While it is possible to use global variables to send information into functions, you should use parameters instead because they make the flow of information clear throughout your program.

In [4]:
```
# Using global variables: works, but bad style
def calcReynolds():
    Re = U * L / nu
    return Re

# Define variables outside script with names that match inside of function
U = 10
L = 10
nu = 1e-6

# Call function
calcReynolds()
```

Out[4]:
```
100000000.0
```

In [5]:
```
# Using parameters: good style, flow of information is clear.
def calcReynolds(U, L, nu):
    Re = U * L / nu
    return Re

# Define variables outside of script (name matching does not matter)
velocity = 10
length = 10
viscosity = 1e-6
```

```
# Pass those variables into function
calcReynolds(velocity, length, viscosity)
```

Out[5]:
```
100000000.0
```

Documentation

Program Summary

Each code file that you submit should include a summary comment at the beginning with your name and a short description of what the program does. I will be lenient on grading your program summary, so long as you include one and it is concise. Use a multi-line comment to write your program summary.

In [13]:
```
'''
Aidan Hunt
ME 498 K
3/28/2023

This program processes data from an ATI-Mini45 load cell, and returns the corresponding
'''

# Import statements
import numpy as np

# The rest of your program...
```

Function comments (docstrings)

Every function that you define should include a docstring below the function definition. The docstring should describe the function such that someone who did not write it can figure out how to use it. The docstring should describe the following:

- **The behavior of the function**: what it calculates/plots/prints/etc.
- **The function parameter(s)**: what each input means, what data type is expected, how it should be formatted etc. If a parameter is optional, state that it is and describe the default value.
- **The function return(s)**: explicitly state which values are returned, what these values represent, and what data types they are, how they are formatted, etc.

Note that functions can have many different types of output (printing things to the console, creating plots, etc.). Therefore, we should be explicit about what we mean by output. If something is returned, say it is returned. If something is printed, say it is printed.

Your docstring should meet the requirements above while matching one of the following formats:

In [1]:
```
# Option 1: short description: ideal for simple functions and
# for functions without too many unique data types
def calcReynoldsNumber(U, L, nu):
    '''
    Given a velocity scale (U), a length scale (L), and a kinematic viscosity
    (nu) calculates and returns the Reynolds number. All inputs are assumed
    to be numeric and have compatible units (e.g., all SI units).
    '''
    Re = U * L / nu

    return Re
```

In [2]:
```
# Option 2: Description and list of inputs and outputs: ideal for
# complicated functions
def getExperimentFiles(dataPath, folderFormat, configFormat='HWConfig.mat',
                       dataFormat='pointData.mat'):
    '''
    Returns data and configuration files from turbine experiments that are
    located in the given directory and match the given format.

    Inputs:
    ----------------
    dataPath: string
        Absolute path to the root data repository. Data files
        will be searched for within this repository.
    folderFormat: string
        Name format of folders that represent individual experiments.
        May include wildcard characters.
    configFormat: string, optional
        Name format of experiment configuration files. May
        contain wildcard characters. (default: 'HWconfig.mat')
    dataFormat: string, optional
        Name format of experiment data files. May contain
        wildcard characters (default: 'pointData.mat')

    Outputs:
    --------
    dataList: list of strings
        A list of the absolute paths of the data files found
    configList: list of strings
        A list of the absolute paths of the configuration files found.
        The order of the configuration files in the list matches the order
        of dataList.
    folderList: list of strings
        A list of folder names corresponding to the data files found.
        The order of the folder names in the list matches the order
        of dataList.
    '''

    # (Code inside function...)
```

One way you can check whether your docstring is sufficiently detailed is to imagine that only the function definition is visible to someone who will use it, and that they cannot see the lines of code inside the function. Based on the function definition alone, what does someone need to know to provide the correct inputs and get the correct outputs?

In [ ]:
```
# Imagine this is all someone can see. What does this function do?
def countLetters(nameIn):

```

However, your docstring **should not** describe implementation details of your function. To someone using your function, it's not necessarily important how you decided to code up a particular calculation or procedure, but rather what goes into the function and what comes out.

In [9]:
```
# BAD: Excessive detail in docstring
def countLetters(nameIn):
    '''
    Given an input name, uses the len function to determine the number of letters.
    The name and the number of letters are then displayed using the print function.
    Then the number of letters is returned to whatever called this function.
    '''

    numLetters = len(nameIn)
```

```
    print('The name', nameIn, 'has', numLetters, 'letters in it!')
    return numLetters
```

In [10]:
```
# GOOD: Concise docstring that describes function behavior without implementation detail
def countLetters(nameIn):
    '''
    Given a name (as a string), returns the number of letters that are in
    the name. The number of letters in the name is also printed to the console.
    '''

    numLetters = len(nameIn)
    print('The name', nameIn, 'has', numLetters, 'letters in it!')
    return numLetters
```

Spacing and indentation

Note: I will not be strict on grading your use of whitespace unless your code is notably difficult to read. In general, you should just use your best judgement.

Use whitespace to enhance the readability of your code.

In general, use whitespace to separate terms in code statements.

In [ ]:
```
# Less readable (characters are bunched together)
z=y+x

# More readable
z = y + x
```

However, sometimes lack of whitespace is useful for organizing within individual terms of code statements.

In [1]:
```
# Less readable (extra whitespace makes individual product terms unclear)
x = 3 * 4 * 5 + 6 * 7 * 8

# More readable (removing whitespace in products makes clear the individual terms)
x = 3*4*5 + 6*7*8
```

A few other notes:

- **Avoid excessive whitespace**. which can make code more difficult to read.
- Reserve the use of indentation for functions, for loops, etc.
- **Avoid excessively long lines**. A good rule of thumb is that no line of code should be longer than 80 characters (many IDEs feature visual indicators of when a line has exceeded this). Instead, break long lines of code into multiple lines using `\` or multiple statements. If a function definition is long, you can break it up simply using line breaks since parentheses imply line continuation. You should similarly break long comments up into multiple lines.

In [7]:
```
# Explicitly breaking a line using \
# (If \ isn't used, you'll get an indentation error)
x = 1 + 2 + \
    3 + 4
```

In [8]:
```
# If using paranetheses, line continuation is assumed
x = (1 + 2 +
     3 + 4)
```

In [ ]:
```
# This lets us break up long function definitions, too
def exampleFunction(input1, input2, input3
                    input4, input5, input6)
```

Import statements

If your program imports and uses any packages or modules, your import statements should be at the top of your program, below your program summary.

If importing the following packages, use the corresponding conventional alias name:

- `import numpy as np`
- `import pandas as pd`
- `import matplotlib.pyplot as plt`

In [9]:
```
'''
Aidan Hunt
ME 498 K
3/28/2023

(Program Summary)
'''

import numpy as np
import matplotlib.pyplot as plt
```

General recommendations

The below are not used for grading, but are other best practices for writing code that is readable.

Variable and function names

- Generally, use lower case names for function names, and reserve uppercase names for the class names. Exceptions to this guideline include single-letter variable names that have symbolic meaning (e.g., `D` for diameter).
- Use camelCasing (`thisIsMyVariable`) or underscores (`this_is_my_variable`) for variable names that consist of many words. Personally, I prefer camelCasing, but you will find both used extensively in Python packages.

Comments

- Comments aren't only for docstrings! Use comments to describe sections of code or complex calculations.
- Remember, comments aren't just for me, they are for you, too!