

CLEAN CODE

BY: OSCAR A. CHAVEZ ORTIZ & CHRISTINE NGUYEN

PURPOSE OF CLEAN CODE

- So you and others can see the structure of the program
- Outlines the flow of the programming logic well
- They tell you and others what is going on in your code

CLEAN CODE OUTLINE

- **Comments/Docstrings**
- **No Repetition - Write Functions**
- **NO HARDCODING or Magic Numbers!!!!**
- **One Task Per Function**
- **Balancing Conciseness vs. Readability**

COMMENTS

- **Comments are a short description as to what a piece of code is doing.**
- **These necessarily come in because there is some ambiguity into the variable assignment or uncertainty in a piece of code.**
- **These are mainly used in places where the logic of the program may be unclear.**

```
b = 56
```

```
# assigning b a value of 56
```

```
salestax10 = 1.10
```

```
# defining a sales tax of 10%
```

```
salestax20 = 1.20
```

```
# defining a sales tax of 20%
```


Example of bad comment practice

```
b = 56
```

```
# assigning b a value of 56
```

A better use of comments

```
salestax10 = 1.10
```

```
# defining a sales tax of 10%
```

```
salestax20 = 1.20
```

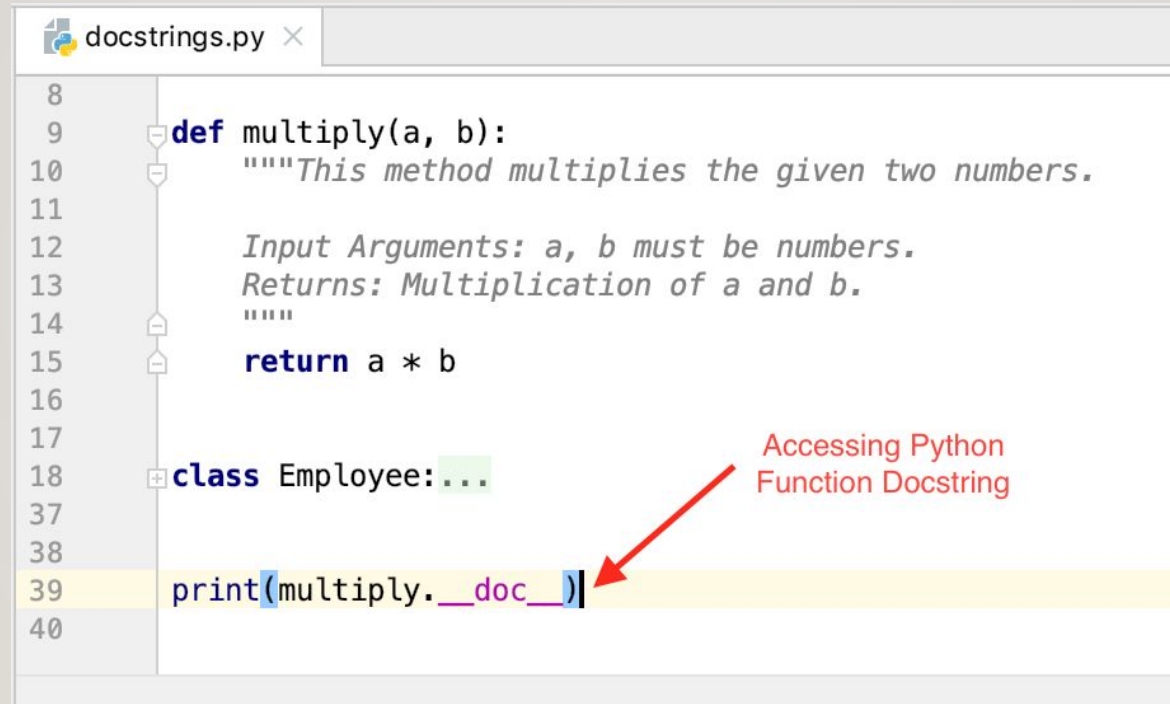
```
# defining a sales tax of 20%
```



DOCSTRINGS

- Docstring are like comments but way more elaborate. These are used to describe chunks of code instead of one line of code.
- To write a docstring you simply write into python
 - `""" DESCRIPTIVE TEXT HERE """`
- You want to use these whenever you make a function
 - The docstring should specify what the function does
 - Include parameters passed into the function and the data type
 - If function returns anything, the name of what it returns and the data type

DOCSTRING EXAMPLE



```
docstrings.py x
8
9 def multiply(a, b):
10     """This method multiplies the given two numbers.
11
12     Input Arguments: a, b must be numbers.
13     Returns: Multiplication of a and b.
14     """
15     return a * b
16
17
18 class Employee: ...
37
38
39 print(multiply.__doc__)
40
```

Accessing Python Function Docstring


```

class reduction:
    """
    This class reduces the data by using dark subtraction.

    It takes in two arguments

    data_files: an array that holds filepaths for the data we want to dark subtract
    dark_files: an array that holds filepaths for the darks corresponding to the data
    """
    def __init__(self, data_files, dark_files):
        self.data_files = data_files
        self.dark_files = dark_files

    def read_data(self, file_paths):
        """
        This method goes through and reads in the data for the respective file path passed in

        Parameter
        -----
        file_paths: this is an array of file paths that we will read the data

        Returns
        -----
        an array of 2D arrays where each element in the array is the data from each file
        """
        store_list = []

        for file in file_paths:
            ccd = fits.getdata(file)
            store_list.append(ccd)

        return np.array(store_list)

```

```

def median_data(self, data):
    """
    This function will take the median of an array of 2D arrays and we will end
    up with a median 2D array.

    Parameter
    -----
    data: an array of CCD images that we want to median combine

    Returns
    -----
    A 2D array that is the median of all the data passed into the data argument
    """

    return np.median(data, axis = 0)

def dark_subtraction(self, data, darks):
    """
    This function will subtract the darks from the data

    Parameters
    -----
    data: The data which we want to subtract the darks from. An array of 2D arrays
    darks: A 2D median Array of darks that will be subtracted from all the entries in data

    Returns
    -----
    An array of 2D arrays that are now dark subtracted
    """

    store_data = []
    for image in data:
        store_data.append(image-dark)

    return np.array(store_data)

```

DOCSTRING EXAMPLE

AVOID REPETITION OF CODE

- If you find yourself in the situation where you are copy and pasting code and tweaking only a few things here and there.
- It would be most advantageous to write one function with the general format of the code you are copying and pasting and put as parameters the things that you are changing.

```
plt.figure(figsize = (12, 7))
plt.title('Flux vs Wavelength')
plt.plot(wavelength, spec1, ls = '--', label = 'Spectrum 1')
plt.xlabel('Wavelength')
plt.ylabel('Flux')
plt.legend()
plt.show()
```

```
plt.figure(figsize = (12, 7))
plt.title('Flux vs Wavelength')
plt.plot(wavelength, spec2, ls = '-', label = 'Spectrum 2')
plt.xlabel('Wavelength')
plt.ylabel('Flux')
plt.legend()
plt.show()
```

```
plt.figure(figsize = (12, 7))
plt.title('Flux vs Wavelength')
plt.plot(wavelength, spec3, ls = '-.', label = 'Spectrum 3')
plt.xlabel('Wavelength')
plt.ylabel('Flux')
plt.legend()
plt.show()
```

```
plt.figure(figsize = (12, 7))
plt.title('Flux vs Wavelength')
plt.plot(wavelength, spec4, ls = '.', label = 'Spectrum 4')
plt.xlabel('Wavelength')
plt.ylabel('Flux')
plt.legend()
plt.show()
```

-
- How would you make a function for this?


```
def plotting_spectrum(spectrum, linestyle, name):  
  
    '''  
    This function will plot a flux vs wavelength for a given spectrum and plot it  
    in the style specified by linestyle.  
  
    Parameters  
    -----  
    spectrum: the spectrum we are trying to plot (array_like)  
    linestyle: the style of the line for the plot (string)  
    name: the name to show up in the legend when we plot the spectrum (string)  
    '''  
  
    plt.figure(figsize = (12, 7))  
    plt.title('Flux vs Wavelength')  
    plt.plot(wavelength, spectrum, ls = linestyle, label = name)  
    plt.xlabel('Wavelength')  
    plt.ylabel('Flux')  
    plt.legend()  
    plt.show()
```

AVOID HARDCODING AT ALL COST!!!



Hardcoding is assigning a number into a calculation or index instead of pre-assigning it to a variable



It would be very hard to understand why there is a random number popping up in your code, this is what is referred to as magic number as is a bad coding practice.



Hardcoding allows the program to not be used as generic as possible.

No Bueno :(

```
def addition(var2):  
    var3 = var2 + 5  
    print('Addition = ', var3, '\n')
```

Bueno :D

```
1 var1 = 3;  
2 def addition(var2):  
3     var3 = var1 + var2  
4     print "Additon = ",var3,"\n"  
5     return  
6  
7 addition(5); #dynamically assigns 5 to var2  
8 addition(10); #dynamically assigns 10 to var2
```

One Task Per Function

For every function you write, it should only have one purpose which can be stated in its function signature.

Examples: `add`, `check_numbers`, `generate_code`, `shift_data`, etc.

If you write functions that do too many things at once, you make it more difficult to test sections of your code, and therefore more difficult to isolate errors.



HOW TO PRACTICE CLEAN CODING HABITS

1. Write your code in a way that is easy for you to understand (Test Phase).
2. Once you have a good sense of what the program is supposed to do and fixed out all the bugs make a finalized draft of the code. (Polishing and Testing)
3. Write comments if you think that someone would not be able to understand what your code is doing.