## Name:

Duplicate Code / Large Class

## Location:

PR301Assign1/ command.py

## Class Name:

Command

## Method Name:

do\_scatter

Lines:

18 to 29

32 to 40

43 to 51

## Reasons:

In the command class there were multiple “draw\_[graphNameHere]” methods which were all similar enough in behaviour. Some of which were almost identical.

The method body for drawing graphs was unnecessarily written in the command.py file. These methods can be put into another file for readability and reduced complexity (unnecessary complexity due to complicated relationships and large classes rather than the nature of the system class).

## Strategies/ approaches:

Create a graph.py file for methods to be moved to.

Subclass the different graph types as children of the Graph class. The Graph class will contain a draw\_graph abstract method. This method is abstract and therefore will simply pass.

Each graph subclass will have its own method body describing the unique behaviour of that certain graph which will overwrite the empty body of the Abstract Class

Now each of the subclasses (types of graphs e.g. line, bar, scatter) of the Graph super class will have their previous method bodies simply moved from the command.py to the graph.py file.

Each graph can now be drawn by calling Graph.draw\_graph(arg).

E.g. Line.draw\_graph(arg) where arg specifies the table data to use e.g. salary, sales, age

## Name:

Large Class

## Location:

PR301Assign1/ validator.py

## Class Name:

Validator

## ~~Method~~ Attribute Name:

All attributes that were rules. Each rule is a list of accepted characters for a validator category.

Lines: 58 to 67

## Reasons:

There can be a few reasons to break up a large class into smaller components. Additionally all of the attributes that were moved were all of a similar type/category, which was they were all types of rules. These “rules” are actually lists of characters which determined what characters were accepted for the field/input. E.g. birthday cannot consist of alphabetical characters ([A-Z] [a-z])

As for why this is helpful? For example, this validator could be extended for additional functionality such as validating data with a different quantity of fields or the same fields but with different accept characters.

In order to achieve this, you could try different approaches depending on what you are hoping to achieve.

You could simply add another class. Rules2. This class now has 10 different fields it can validate. You can now pass these rules (Rules2 class) as an object to another class (Validator) rather than the original set of rules (Rules class).

Why? Well, now while you are trying to use the same validator to validate multiple different formats of input data with varying accepted and rejected characters. You can simple write up a quick set of rules and save them as a new set as opposed to modifying the existing rules of the original, unedited Validator class every time different rules need to be applied.

(other code is required to extend this program, simply added new rules does not achieve this functionality, I’m merely pointing out how simple it would be to add multiple unique sets of rules after additional extension code is implemented).

## Strategies/ approaches:

The strategy used was to create a class for Rules. This class would produce a rules object which has the list of rules saved in a dictionary as an attribute of the rules class. A dictionary was used as it’s easy to refer to a dictionary value or key using dict[Val] or dict[Key] functions. An array would have required multiple array functions to search the array items for a match and return the associated array index.

## Name:

Long method/Large Class/Switch Statements

## Location:

PR301Assign1/ file\_handler.py

## Class Name:

FileHandler

## Method Name:

open() - 13 to 27

txt\_dict\_reader() – 29 to 56

csv\_dict\_reader() – 59 to 103

xlsx\_dict\_reader() – 130 to 173

## Reasons:

First of all the original state of this class was too big, second, the functionality between the 3 different methods is actually very similar.

The original FileHandler class had ELIF loop function called open() which determined what file extension was being used by comparing values with IF’s and ELSE IF statements which is also considered a bad smell as anytime the system needed to determine what file type was using it went and compared all possible file types.

This is an example of a switch statement bad smell in python. Python doesn’t explicitly use the term ‘switch’ as the language does not use this keyword and rather a ELIF loop fundamentally behaves the same as a switch loop in other languages.

Now rather than a switch case when the loop identifies a match. There is a method that finds the file type extension of the target file, then uses that value to find what file type. E.g. if the method determines that the file ends in .txt then it can quickly match that value to a dictionary of file types.

Why do this? Well.. if we were to extend this program to allow it to accept more file types than just the 3 we have initially chosen, then the developer will be required to add additional methods to the FileHandler class which will increase its size and functionality even more. This class with continue to be an overly complex and large class with too many responsibilities.

To extend the original open() method you would end up adding additional ELIF causes to this already large loop and then write a corresponding method for the new file extension.

## Strategies/ approaches:

The solution for this problem(s) is to simply move these methods out of the FileHandler class and into a FileType class. This FileType class has a Base Class called ‘FileTypes’ which simply has ONE abstract method named read\_file.

Make each type of file extension a subclass of the FileType class. In the bodies of these subclasses you will find the override method body for the read\_file for that particular file extension.

To use these subclass methods the open() method is modified. Now the open() method will determine what the file extension is by looking at the end of the file name (after the . ) and will use that file extension as a dictionary key to point towards the associated subclass.

Open(file) -> returns ‘.txt’ as a key to use for the next part

‘.txt’ key is used to find the value, the value is a reference/call to the method of the TXTReader, a subclass of the FileType base class. This happens inside the open() method.

Open method returns a function call: TXTReader().read\_file(path).

## Summary:

For this third smell I have decided to write a quick summary to explain the reasoning behind applying so many potential smell types to this one problem. This problem was an example/address a few different smells.

It was a ***long method*** as the open file was looping through every possibility and then if it found a match it would run a check\_data method. This is a lot of functionality and was quite large (obviously as this is a small system what is considered large in this context is very small when compared to larger and more complex systems)

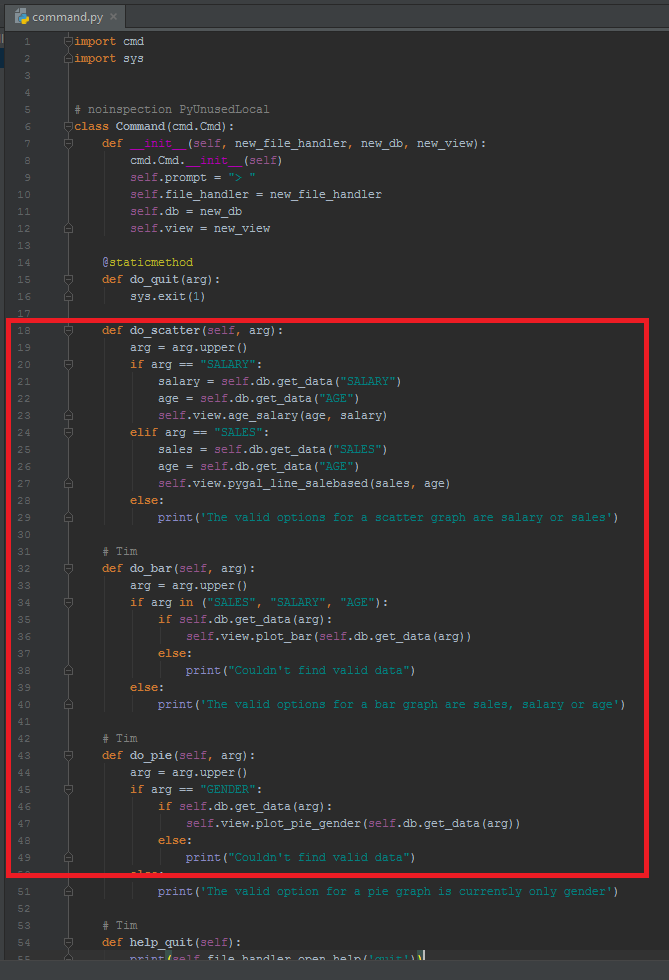
It was a ***large class*** as it had too many methods and was also had functionality that could be moved into another class to encapsulate that behaviour. Again, this does not seem to large as originally the class had around 6 to 8 methods total. The reason you can consider this large was that if you were to continue adding additional file types that the system could handle, then you would end up adding more methods to this class and further increasing its size.

The file handler’s purpose was to open the files and that method was left in that class. Now all accepted file types, and additional ones, can be added to the file\_type class while the file reader continues to do what it is supposed to, read files, using the open() method which was not necessary to change when adding new file types as the open() method was written well to accept any potential input. The success of running the open() method is down to whether the file type of the file matches one that has been programmed into this system or not.

It was an example of ***switch statements*** as the it had a long IF & ELIF loop in the open() function to determine what filetype it was. A smarter way to go about this was to have a method to determine the file type and use that value rather than looping through all possibilities until the correct one matched.

# Screenshots

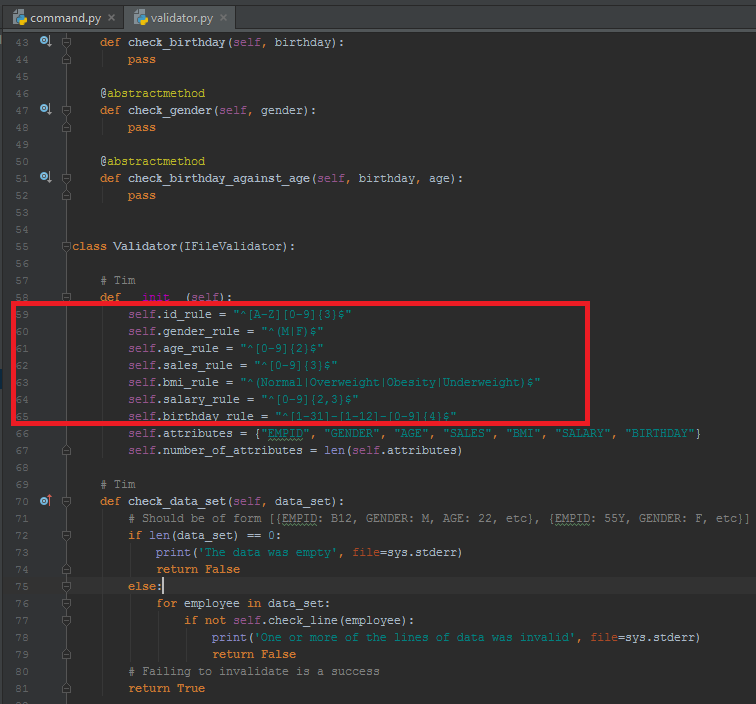
## Smell 1 Before



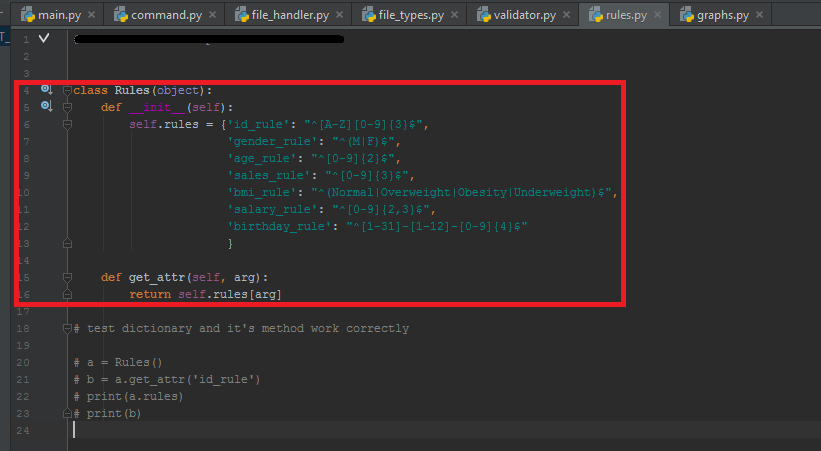
## Smell 1 After

## 

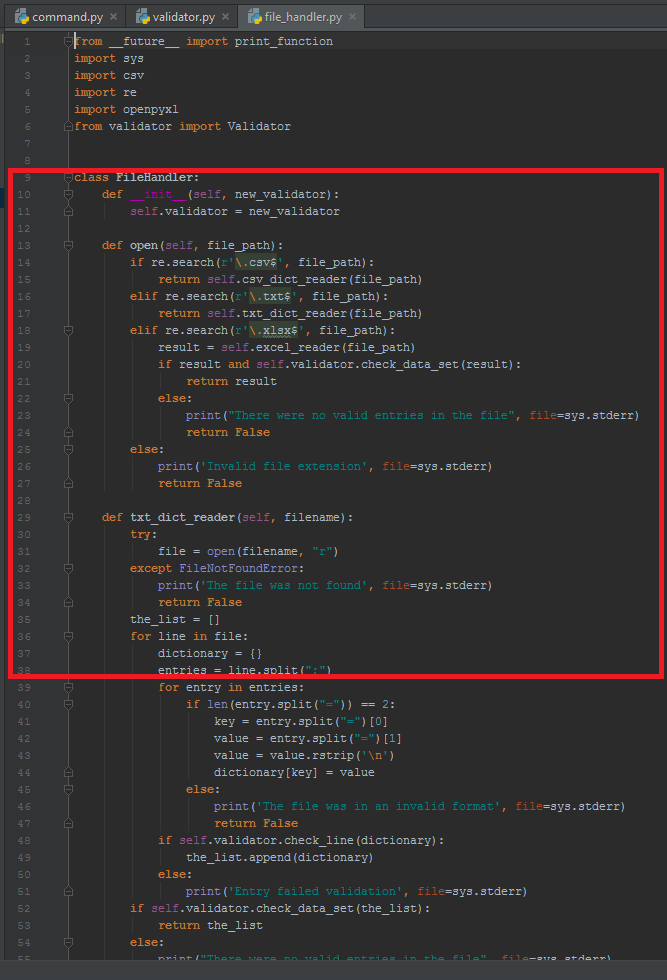
## Smell 2 Before



## Smell 2 After



## Smell 3 Before



## Smell 3 After

