upGrad

Session - 2: Clean Coding Principles

2.1 Importance of Writing Clean Code

DevOps removes the silos between different teams, such as development, operations and QA, leading to high-quality software development. Developers write and deploy their code more frequently than earlier. But with such agility, it is also essential to ensure that the code being written has a good quality score in terms of design. Therefore, developers should follow certain clean coding principles whenever they write code. The advantages of following clean coding principles are as follows:

- A clean code is easy to understand, modify and maintain in the long run.
- Whenever a developer writes clean code, it becomes easy for other developers to work on it, if required, in the future.
- Writing clean code serves as a foundation for the entire project in the long run, similar to a building with a strong foundation so that it lasts long.

Now, suppose a developer writes a piece of code with an aim to make the required functionality work and to meet deadlines without following any standard principles of code writing. In that case, the code will work initially but can create trouble in the long run because of the following reasons:

- Suppose, in the absence of the developer who initially worked on the code; a new developer works on it. For the new developer to be able to fix bugs or work on the same code, that is, be productive, will take a lot of time because the code would be very complex and difficult for them to understand.
- With each cycle (sprint), testing becomes very difficult, and the cost of adding a new feature increases. For example, adding a simple trivial feature also takes time.
- Many design issues may be associated with the code, such as tightly coupled components, scaling, reusability, improper variables and class names.

Therefore, not addressing such issues ultimately leads to a messy code base, which becomes very hard to manage with time and can result in application downtime. Thus, in the worst cases, an organisation might also need to start everything from scratch.

2.2 Understanding Clean Coding Principles

The different principles for writing clean code are as follows:

- **Meaningful naming convention**: The variable's name should always be descriptive and highlight its purpose.
- **Code modularisation**: This refers to the process of organising code into functions so that the function can be reused and the code looks concise and readable. It is also known as the 'do not repeat yourself' (DRY) principle.
- **Code optimisation**: Code optimisation deals with the runtime execution and memory utilisation aspects of a program.
- **Code documentation**: Code documentation ensures that the code is well-documented and highlights the details of each aspect of the code, such as function arguments, return types and description.
- **Code linting**: Code linting is the process of checking whether the guidelines associated with the programming language are followed. For example, Python follows the guidelines described in PEP8(Python Enhancement Proposal).

2.2.1 Code Refactoring

Often, development teams compromise the quality of the code to release a feature quickly in the market and to meet customer demands. They can accomplish their task and gain their customers' trust but with some trade-off with the quality. This is called 'technical debt' and needs to be handled by the organisation in the future to maintain the code in the long run. If technical debts continue accumulating, adding any new feature could become almost impossible.

A solution to this problem is code refactoring. It is the process of improving the internal structure of the code so that it is easy to read and modify without changing its external behaviour (functioning).

In Python, there are different ways to refactor your code and make it concise and more readable, some of which are as follows:

- **Using list comprehensions**: This is one of the most elegant or Pythonic ways of creating a list based on existing lists. The code written in this way is concise, clean and easy to read. The basic syntax of list comprehension looks like this.

  [<expression> <for item in list> <if conditional>]

```
## Case 1: List Comprehensions in place of for loop

# Code to create a new squared_list of numbers based on conditon(if odd)
my_numbers = [1,2,3,4,5]
'''
sqrd_list = []
for item in my_numbers:
  if item % 2 == 1:
    sqrd_list.append(item ** 2)
'''

# The above four lines can be replaced by a single line
sqrd_list = [value ** 2 for value in my_numbers if value % 2 == 1]
print(sqrd_list)
```

- Assignment using tuple unpacking: Unlike declaring and initialising each variable on a separate line, Python supports a more idiomatic way to achieve this called tuple unpacking.

```
## Case 2: Value Assignment Using tuple unpacking
'''
name = 'Steve'
age = 44
roll_no = 28
'''
# The above task can be easily done using tuple unpacking on a single line
name, age, roll_no = 'Steve', 44, 28
```

- Use of **any** and **all** functions: You can also use **any** and **all**, two built-in Python functions for refactoring your code. **any** function checks the passed iterable(list, tuple etc) and returns true if any of the elements or the given condition for any element is true. For example, let's say you want to check if an even number is present in the num_list = [12, 22, 33, 44, 55]. Then the code written using for loop can be replaced by the **any** function as shown below.

```
## Case 3: Using any and all functions
# Old code to check if any even number present
num_list = [12, 22, 33, 44, 55]
'''
for value in num_list:
    if value % 2 == 0:
        is_even_present = True
        break
'''

# Using any function
is_even_present = any(value % 2 == 0 for value in num_list)
```

Similarly, **all** function checks if the condition is true for every value or not. As soon as it finds that the condition is not true for any value, it returns false; otherwise, it returns true after running through the complete list.

- **Using lambda functions**: Lambda functions can also be used for code refactoring in Python. Often, a function has a single expression. In such cases, a simple anonymous (unnamed) function called lambda function can be defined. The general syntax of lambda functions is as follows.
  lambda <argument_list>: expression

```
# Case 4: Lambda Function
## A Simple function to calculate_average
'''
def get_greater_number(num_1, num_2):
    if num_1 > num_2:
        return num_1
    return num_2
'''
# You can use lambda(anonymous function) to write this in one line

get_greater = lambda num_1, num_2 : num_1 if num_1 > num_2 else num_2
print(get_greater(12, 22))
```

2.2.2 Code Optimisation

Optimising your code in terms of time and space complexity is very important. Python is one of the most popular programming languages in data science, machine learning, deep learning and AI because of its ease of writing code and performing tasks. However, problems become apparent when you deploy your application on a large scale. For example, you have developed an e-learning application and deployed it in production. Initially, it performed well, but its performance gradually decreased as the number of users increased and your application scaled up. This is where code optimisation is important. It can improve either the code execution time or memory utilisation or both.

Time Complexity

A developer can optimise the execution time of their code in multiple ways. These are as follows:

- Use list comprehensions in place of for loops
- Use built-in functions such as **sum**, **map** and **reduce** wherever possible. These functions are already defined in C and are optimised. For example, to add the numbers present in the list such as num_list = [1, 2, 3, 5, 6], you would do the following.

```
result = 0
for value in num_list:
    result += value
```

  You can perform this addition using built-in functions by simply calling the sum function over the array; for example, result = **sum(num_list)**.

- Local variables should be preferred instead of global variables because of security vulnerabilities associated with global variables.

Space Complexity

Apart from this, developers can also optimize memory in various ways.

- For example, using list comprehensions makes the code fast, but the flaw is that they create a new list in every case. So, if you want to delete the elements from a huge list using list comprehension, it will not be suitable since it will require large memory for deleting elements compared with a 'for loop'. Using a 'for loop', you can delete elements using the functions **list.pop()** or **list.remove()** in the original list instead of creating a new one.
- Similarly, many times, using an array is not suitable because values are stored in a contiguous fashion, and it may be possible that such a high amount of contiguous memory is not available. Therefore, linked lists are helpful in such cases. However, searching for elements is slower in a linked list compared with an array. Therefore, there is always some trade-off between choosing an array and a linked list regarding memory utilisation.

2.2.3 Code Documentation

Many developers argue that clean code speaks for itself. For example, if we have given your variables, classes and functions meaningful and descriptive names, then it becomes easy to understand code. However, as the size of the code base increases, it gets difficult to manage because to understand what a particular code does, you will not reread your code from scratch(beginning). This is where the code documentation serves its purpose. It contains a description of the different portions of the code, which is helpful for you, your team members and new joinees, if any, in the team.

Docstrings

In python, documenting a code is easy and handy with docstrings(built-in strings). These docstrings define the complete behaviour of the objects like class, function etc. The general syntax of writing a docstring is between 'triple single inverted commas' or 'triple-double inverted commas'. Also, the docstring for any class, function, etc., needs to be present immediately after the name of the class and functions. This docstring is stored in the **__doc__** attribute of the object, which can be accessed by **object_name.__doc__**. The docstring can also be accessed using the help function. For example, **help(function_name)** will also print the docstring of the function or the object passed as an argument. The below image shows how a docstring should be defined for a function.

```python
def get_neighbors(some_list, index):
    """Returns the neighbors of a given index in a list.

        Parameters:
            some_list (list): Any basic list.
            index (int): Position of an item in some_list.

        Returns:
            neighbors (list): A list of the elements beside the index in some_list.
        Sample usage some_list = [8,7,"car","banana",10]

    """

    neighbors = [] # creating an empty list
 #checking the boundary cases
    if index - 1 >= 0:
        neighbors.append(some_list[index-1])
    if index < len(some_list):
        neighbors.append(some_list[index+1])

    return neighbors

# Will print the docstring for the above function
print (help(get_neighbours))
```

Comments vs docstrings

In Python, comments improve readability and indicate how the code works, whereas docstrings indicate what the code does. It provides us with a description of functions, methods or classes. Docstrings can be considered a special type of multiline strings or comments placed just the function or class heading in the definition block.

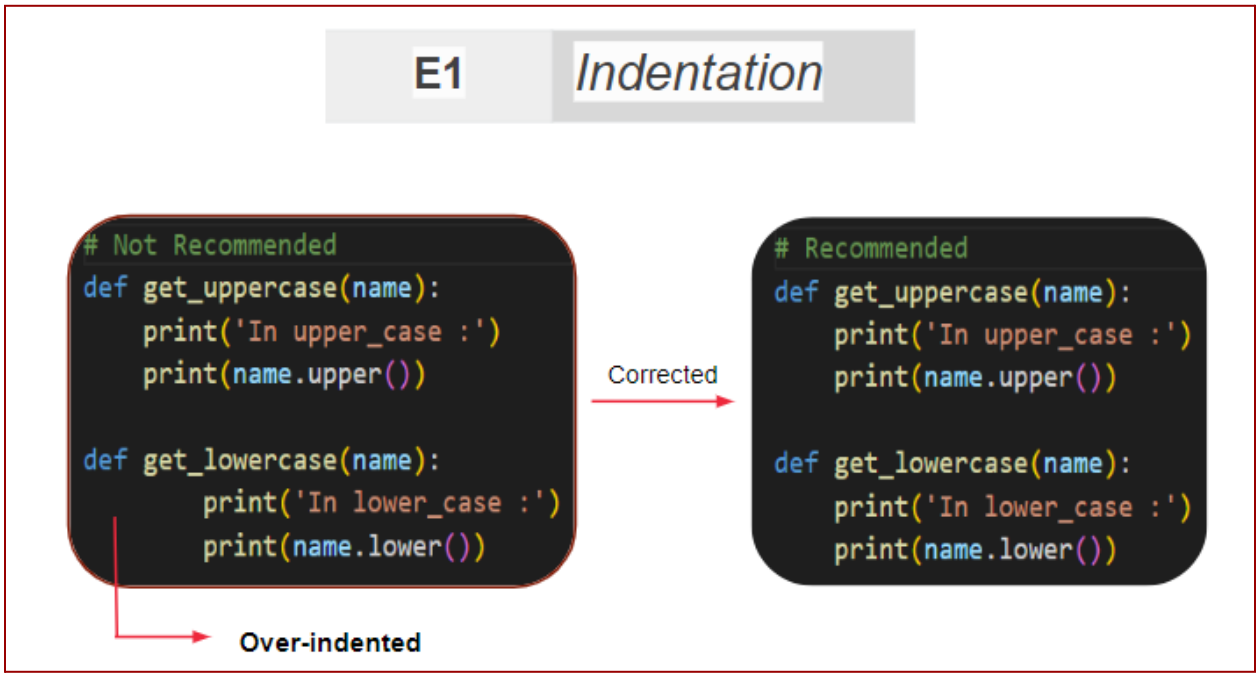
Note: Although a docstring and a multiline comment appear similar, they are different. Multiline comments can be given anywhere in the program, but docstrings can only be at the top of a function or a class.

2.2.4 Code Linting

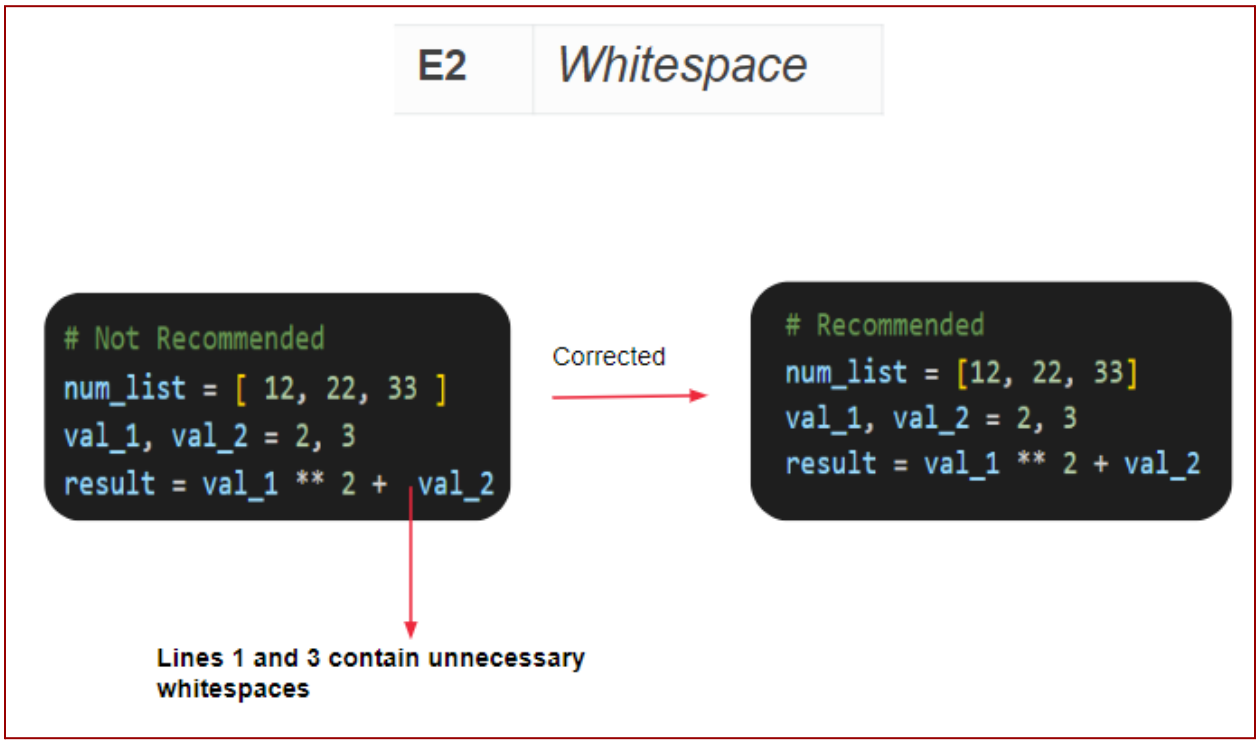
Along with documenting your code, ensuring that the code is of high quality is also essential. Python has a set of guidelines for ensuring that the code quality is high and meets the standards of the language. These guidelines are defined by a document called PEP8, written in 2001. To check whether our code follows these

guidelines, we use linters. Linters are tools that check your code and try to find inconsistencies or improper code constructs. A few of the famous Python linters are pycodestyle, pylint, etc. Some of the code writing guidelines that need to be followed while writing code are as follows:

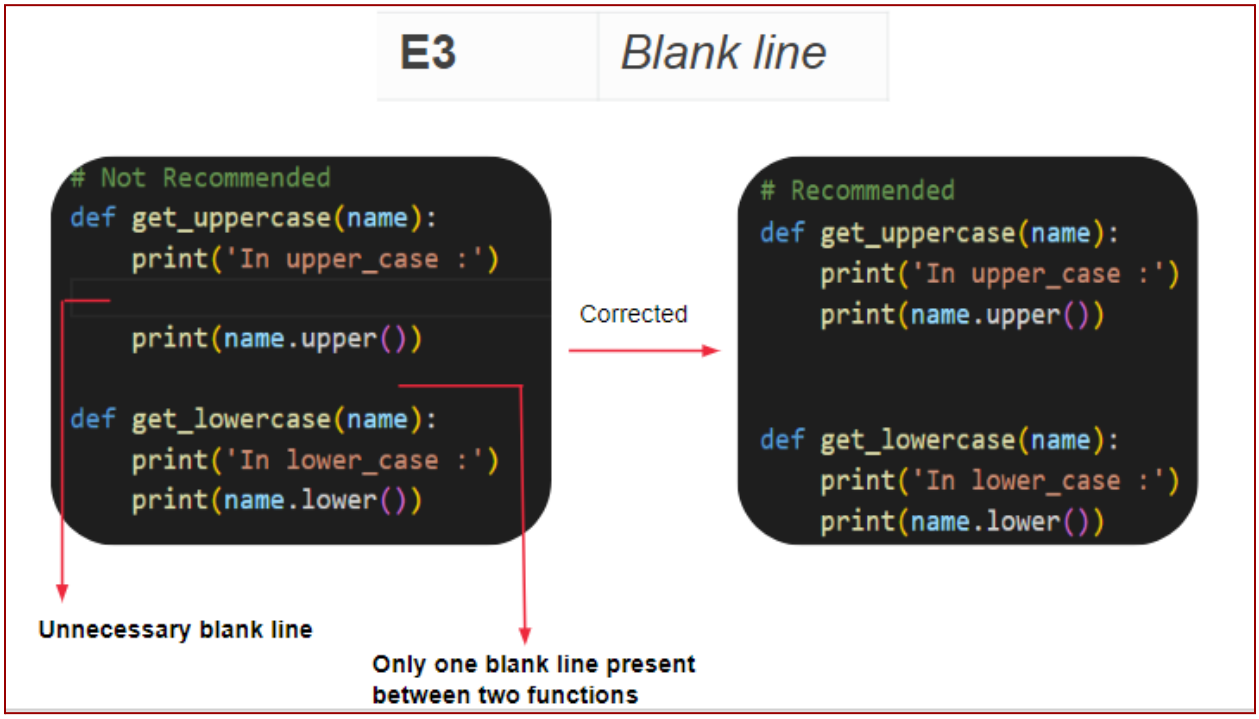
- Indentation



- WhiteSpace



- Blank line



- Line length