CLEAN CODE

Advanced Guide to Learn the Realms of Clean Code from A-Z

MARC ROBERTS

Clean Code

Advanced Guide to Learn the Realms of Clean Code from A-Z

Table of Contents

Introduction

If we talk about the fundamental or core discipline which the very foundation of this book revolves around, it would undoubtedly be the principles taught in software engineering. In a sense, keeping the code clean, making it more robust, removing its fragility, and making the entire software overall more ‘artistic’ is what software engineers actually do, but exploring the realm of software engineering can easily confuse your path and blind you from your goal. Thus, such topics require a guiding hand, and without one, there’s no guarantee that what you learn will actually be useful.

This is the core concept of this book, to be a guiding hand in learning those techniques of software engineering that can help us create ‘clean code.’ Before we jump into the first chapter, a piece of advice is to correct your state of mind. To elaborate, you need to define the perspective through which you will read this book. Usually, many people read books such as this with the mindset that it will have the one and only solution to a problem they couldn’t find anywhere else. This psychology is based on nothing more than a misconception because there is no perfect solution to a problem in software engineering.

Moreover, if you encounter a problem, only you can come up with the solution, that’s how the problems in the programming world have been solved to this day, and this also how new innovations are made.

So, this book will serve as a guiding hand for you, making you explore those topics that are the most crucial for you to develop a robust understanding of programming and thus, help you achieve the goal of this book, to be able to clean any code you want, whenever you want and to come up with your own solutions to any dead ends you might encounter in programming.

Chapter 1: Introduction to Clean Code

This chapter will focus on the question and necessity of clean code, and we will learn everything there is to know about what a clean code is and why it is necessary. This chapter focuses on understanding that a code quality amounts to the code being readable and relaxes an individual from extensive rework when tracing and rectifying an error.

In addition to this, we will also learn the importance of formatting and documenting the code. This might sound like unnecessary or useless work, but we will see the role that formatting and documenting the code plays when it comes to maintainability and accessibility of the work.

Overall, we will learn the proper rules and guidelines that would improve our codes' efficiency and effectiveness. Automated tools have made coders and programmers' lives easier due to the minimum human interaction involved in their tests. Therefore, it is essential to understand the use of these automated tools in maintaining the code in line with the reference. We will shortly discuss the method of enacting and embedding the tools in the codes to run as part of the code itself.

By the end of this chapter, we will have the answers relating to the queries about the meaning, use and importance of a clean code, its impact on our work, and the use of automated tools in the code itself. The information gained through this should be used in creating a clean and automated code practically.

In summary, the following points would be resolved at the end of this chapter:

- Importance of a clean code.
- Identical and maintainable software construction.
- Using features of Python for self-documentation of the code.
- Configuration of automated tools that will provide help in the layout of the code.
- The use of automated tools to detect the problem and error at the first instance.

What is a Clean Code?

To understand a clean code, an individual must have some experience relating to creating or checking the codes. This is because a clean code is not a thing that can be detected by any compilers or machines. There is no definition or rules regarding the creation and use of a clean code. The code, no matter how badly written, will not affect the machine. But only the professional eye can catch the irregularities in a badly written and formatted code.

It has been a misconception for decades that the codes are our way of communication with the machines. This is partly true, but there is another side to this picture. A programming language is not only a mode of communication with the machine but, in fact, a mode of communication to other developers as well. The developers see what their predecessors did to extract any useful information and change it as they please.

Keeping the above point in mind, we can easily assume the importance of a clean code. A clean code is the best mode of communication with other developers and engineers, as it would allow them to read and understand with ease what the previous developer did. If we have to edit a code, we would have to spend time reading and understanding it rather than its result. And only a clean code can be read, understood, and manipulated easily.

In summary, a clean code can only be defined by the developer himself, through experience and professional attitude. Therefore, we have to go through the complete book and define the meaning of a clean code by ourselves.

Merits of Clean Code

Numerous merits and pros exist in creating a clean code that signifies its importance in a good build. Firstly, in a continuous and efficient work environment, it is necessary to have a clean code for other team members to understand and modify. Reading through an unorganized code means taking extra time in solving a problem that could be resolved at a much faster pace if the code was developed with consistency throughout. This would allow the team members to focus on the actual problem rather than spending and wasting time understanding the issue.

This point can be easily proved with an example. Suppose you are cutting a tree. When a tree is being cut, only one side is being axed regularly. If you were to hand the ax to another person in your team to continue your job, he would easily start from the place where you left off. But now imagine that you are chopping the tree at different places all at once. Now, if you were to give your job to your team member, he would first have to find the spot that has been cut the most and then start chopping from there. This is how clean code functions. When everything is regular, and on point, it is much easier for others to pick up all the useful information without needing to understand it.

While we are discussing the importance of clean code, the term "technical debt" comes to mind. This term is one of the reasons why a clean code is a requirement, not a luxury. Technical debt refers to an additional amount of work that is caused by using short term methods or short cuts to solve a problem. This means that something that could have been done regularly and properly was completed by taking short cuts. This would surely amount to a debt that affects the person who has to modify or rectify an already existing, improperly performed task.

Technical debt in coding and development comes into play when a programmer makes a code and does not bother to regulate it with regard to an existing framework. This would impact the modifier or rectifier of the code as he would have to incur an additional amount in either time or money or both to rectify and modify the code.

The technical debt does not cause an alarming or attention-seeking situation. Instead, it stays there silently like a mine, ready to explode on the next person stepping on it.

Code Formatting and Its Role

A clean code is independent of coding standards, linting tools, and formatting. A clean code is all about making good and regular software and a maintainable, tough, and regular framework. In addition to this, it must also be able to avoid technical debt.

Although quality code is not as much dependent on formatting, formatting forms a good part of how much a code is clean. Therefore, it is also necessary to learn about formatting and its role and impact on a clean code.

Uniform Coding Style

Many guidelines need to be followed in order to create a successful and quality build. This part of the chapter will discuss the coding styles, and we will also learn the use of automated tools to enforce a uniform coding style.

Consistency and regularity are the first things that come to a programmer’s mind when exploring a code layout. If we were to look into a code, we would rate it based on the level of consistency that is being followed. If a code is consistent, it will reflect the team's professionalism and commitment or that of the programmers creating it. On the other hand, it can be said with certainty that an irregular code will have more bugs and errors and would be time-consuming to rectify and correct.

For this reason, especially in a large organization, all members of the team must agree on one standard and layout of coding. This would result in increased efficiency and reliability among the members, as one mistake could be quickly identified and corrected by the others.

Let us understand this phenomenon through an example. Let us suppose that a person is building a computer with a number of components that are each attached to each other through numerous wires and cables. If this computer was to be repaired or another component was to be inserted by another person, he would have to figure out what the previous individual did. If the previous person created his build in an unorganized manner, the next person would surely have to spend more time understanding the complex system of cables used. If the previous person had arranged everything systematically and logically, the next individual will have no problem understanding the build.

This is how a clean code communicates between professionals. If the code is clean and logically, and systematically organized, then the new programmer will have no trouble understanding and modifying it. Python is also not exempt from this principle. In python, it is advisable to use PEP-8 as a coding style. This style can be used as a whole or in part, according to the project we are working on.

PEP-8 for Python is a strong style that needs no further improvement. This is because this syntax was developed and formulated by the Python developers, who were the contributors to Python's elements. This is why PEP-8 hardly requires any modification or improvement.

PEP-8 bears the following attributes that set it apart from others like:

Searchable: PEP-8 allows the programmer to search in certain files for a required string. For example, the following grep command can be utilized to find the file and line required:

```
$ grep -nr "location=" .

./core.py:13: location=current_location,
```

The following command will be used to understand the position where the value is being assigned to this variable:

```
$ grep -nr "location =" .

./core.py:10: current_location = get_location()
```

In PEP-8, spaces are not used to pass arguments by keywords to a function, but use spaces when we want to assign variables. This makes the code run systematically and in uniformity.

Uniformity: For building a successful, efficient, and effective team, it is necessary that everything must be uniform. If a code is consistent and uniform throughout a project and the team has decided a standard of styling the code, then even the new set of developers in the team would easily identify the message that is being communicated. Here, PEP-8 comes into play by giving a uniform and consistent style of build.

Standard of Quality: Code quality is the sole standard of understanding the programmer's professionalism and skill. If the code is of bad quality, it will reflect the developer's unorganized and non-professional behavior. In addition to this, bugs and other errors in the code could be seen at a glance if the code is of good quality. PEP-8 ensures a code of good quality and gives the impression of a strong build.

Documentation of the Code

One of the basic things that a programmer should focus on is the documentation of the code being written. A good code is not only straightforward but also very well documented. Therefore, we must learn about how we can document our code so that it explains its creation and its function by itself. In this chapter, we shall learn about the different ways in which we can successfully document our code.

An important thing to be kept in mind is that documentation does not amount to comments. Comments must be avoided, as they are generally not considered a good method for documentation. Instead, in an active code like python, the best method of documentation for future programmers and developers is by an explanation of data types, stating the crude information, and providing examples and annotations.

Meaning and Use of Docstrings

A docstring is a vital part of the documentation of a code. It can be simply explained as a literal string that is embedded in the code. The purpose of a docstring is to explain that part of the logic of the code.

The question arises that if docstrings are also giving information about the code, what is the difference between them and comments? In addition to this, why are comments considered bad but docstrings good?

The answer to this is that docstrings are different from comments, as the first one intends to document part of the logic of the code while the latter gives an explanation and function of the code. We discussed earlier that a good and clean code needs to be self-explanatory, so comments defeat this attribute of a clean code. In addition to this, comments become outdated with each modification in the source code and may become a source of misdirection and misrepresentation if a modifier forgets to edit and change it according to the modifications made by him. Due to these reasons, it is highly advised to use docstrings.

A function taking anything as a value in its parameters is one of Python's characteristics, which makes it an extremely dynamic programming language. To modify a function, it is extremely difficult to find it, let alone its parameters, in Python. The only support that a modifier can get is through the docstrings embedded by the developers.

To understand this concept, the following example is of good help:

```
In [1]: dict.update??
```

```
Docstring:

D.update([E, ]**F) -> None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] =

E[k]

If E is present and lacks a .keys() method, then does: for k, v in E: D[k]

= v

In either case, this is followed by: for k in F: D[k] = F[k]

Type: method_descriptor
```

Here, the docstring for the update method on dictionaries gives us useful information, and shows us that we can use it in different ways:

1. We can pass something with a .keys() method (for example, another dictionary), and it will update the original dictionary with the keys from the object passed per parameter:

```
>>> d = {}

>>> d.update({1: "one", 2: "two"})

>>> d

{1: 'one', 2: 'two'}
```

2. We can pass iterable pairs of keys and values, and we will unpack them to update:

```
>>> d.update([(3, "three"), (4, "four")])

>>> d

{1: 'one', 2: 'two', 3: 'three', 4: 'four'}
```

Whatever the case is, the dictionary will update itself along with the rest of the keyword arguments given to it.

As we can see in the first example, the docstring of the function was obtained by us through the use of a double question mark (dict.update??). This feature of IPython interactive will give the docstring of the expected object.

Another point to remember is that a docstring is not independent of the code. Rather, it is and becomes part of the code due to its _doc_ attribute:

```
>>> def my_function():

... """Run some computation"""

... return None

...

>>> my_function.__doc__

'Run some computation'
```

So, it becomes clear that accessibility at runtime, generation, or compilation of documentation from source code is now possible. For this purpose, if we run Sphinx, a basic structure for our documentation will be generated. And furthermore, with the use of autodoc extension (sphinx.ext.autodoc), the docstrings will be placed in the pages which document the function.

As we know, everything has merits and demerits. There are always two sides to a coin. The same is the case with docstrings. Even though they help us in the documentation of our code, yet constant manual maintenance is a con of docstrings that cannot be ignored. But when we weigh the pros and cons of docstrings, it can be said with certainty that the advantages of docstrings outweigh the disadvantages.

A developer is constantly facing the challenge of communication with other programmers. The use of docstrings can easily overcome this challenge. Even though human intervention is required from time to time, the cost of intervention is extremely nominal compared to the cost incurred due to the loss in efficiency in deliverance.

What a developer needs to understand is that software is not only a code but also a system of documentation that comes along with it. Therefore, it can be said with certainty that clean software is a combination of quality code as well as its documentation.

Meaning and Use of Annotations

Annotations are introduced in PEP-3107. Basically, annotations provide coders with hints of the expected values of a function's arguments. A characteristic of annotations is type hinting. Type hinting and other uses of annotations will be discussed at a later stage in this book.

Annotations describe what a variable represents in actuality. It describes the type and any useful property, aspect, and object that describes the data in question.

The following example should be considered,

```
class Point:

        def __init__(self, lat, long):

                self.lat = lat

                  self.long = long

def locate(latitude: float, longitude: float) -> Point:

        """Find an object in the map by its coordinates"""
```

In the above example, the use of float is to figure out the expected types of longitude and latitude. As it is discussed previously, this is a piece of mere information about the type for the readers and will not be enforced in Python.

Another function of "Point" is available in annotations. This is a user-defined class utilized to specify the expected type of value that will be returned by a function.

Annotations can not only be defined by types or built-ins. In a broader sense, anything that is allowed by the interpreter of Python can be used as annotations. For example, to get an explanation regarding the intention of a variable, a string can be used. Another good example is a callable. In that case, we can say that annotations have a much broader scope than a typical type or built-in.

Annotations also contain an attribute called _annotations_, which, being a dominant attribute, allows the user access to a dictionary that records the name of the annotations with their corresponding values that were defined by us previously. This could be explained better with the following illustration.

```
>>> locate.__annotations__

{'latitude': float, 'longitude': float, 'return': __main__.Point}
```

This can be used for multiple reasons like:

- Generation of documentation
- To run validations
- Enforcing checks in the code.

When we discuss the system of checking codes with the use of annotations, PEP-484 comes to our mind immediately. PEP-484 explains the use of checking the types of our functions and type hinting through the use of annotations.

The question that arises here is, what is type hinting? To be straightforward, type hinting can be defined as an indicator that shows the type of value within our Python code statically. The concept reflected by type hinting is the use of other tools that are not dependent on the interpreter for checking the types within the code. Type hinting also shows and indicates any incompatibilities in the code to the user. But for this to run, a tool called Mypy is required, which we will study in detail in the succeeding chapters. In short, Mypy is an excellent tool to identify bugs in the early stages of coding, and thus it is advisable to run Mypy along with other tools during a build.

It can also be said that the annotations have become a much more meaningful and useful concept in a way that now we can easily understand the meaning and purpose of a code. If we were to say that a function will work with lists and tuples in one of its parameters due to this concept, we would not be wrong.

Another improvement with regards to Python 3.6 is that variables can be annotated directly. Before Python 3.6, annotations were used as function parameters and return types, but now, annotations can be directly applied to variables. The concept of annotating the variables directly arises in PEP-526 and it explains the declaration of some variable types that are defined without assigning them a value. This can be explained more easily with the following example:

```
class Point:

lat: float

long: float

>>> Point.__annotations__

{'lat': <class 'float'>, 'long': <class 'float'>}
```

Are Docstrings Replaceable with Annotations

After going through the above information, a question arises - do annotations replace docstrings? In the older versions of Python, the only way of documenting the code was by the use of docstrings. There was also a detailed structure and format regarding the use of docstrings in order to achieve the maximum information and detail regarding functions, meaning of parameters, etc.

On the other hand, annotations can be regarded as a compact and concise documentation method compared with docstrings. Resultantly, it can be said that docstrings and annotations are harmonious to one another and should be used together instead of utilizing one as the replacement of the other.

Even though docstrings and annotations complement each other, it is a crude fact that annotations can utilize the previously embedded information in the docstrings. This can be taken as an opportunity for better documentation through the use of more docstrings in the space left empty by the utilization of annotations. This is useful, particularly in the case of nested data types, where an example regarding the expected data might provide a better understanding to the user.

The following lines of code shown feature the ‘response’ function, and this function requires that the parameters being passed to it should be in the form of dictionary values.

```
def data_from_response(response: dict) -> dict:

if response["status"] != 200:

raise ValueError
```

```
return {"data": response["payload"]}
```

As we discussed previously, this small block of code uses a function to which we pass a dictionary value (remember that it will not accept any other type of data value). Once we execute this code, we come to notice that the output value of this function is also a dictionary value. In addition, we also use an ‘if’ conditional placed right after the initial function as well. This is to ensure that if the dictionary value output by the function does not conform to a key which the code expects, then the conditional will trip and raise an error.

Now, we will use docstring to elaborate it better:

```
def data_from_response(response: dict) -> dict:

"""If the response is OK, return its payload.

- response: A dict like::

{

"status": 200, # <int>

"timestamp": "....", # ISO format string of the current

date time

"payload": { ... } # dict with the returned data

}

- Returns a dictionary like::

{"data": { .. } }

- Raises:

- ValueError if the HTTP status is != 200

"""

if response["status"] != 200:

raise ValueError
```

```
return {"data": response["payload"]}
```

The illustration above clarifies our concept regarding the expected value of this function, reception, and return. By following this method of documentation, a developer can communicate his ideas with clarity to the other developer. Furthermore, for the purpose of unit tests, it serves as a valuable source. Following this process would give us data that could potentially be used as an input to derive information regarding the correct and incorrect values to utilize on the test.

Understanding the possible values of the keys and their types are a few of the benefits of annotations. This helps us significantly in the interpretation and comprehension of the data. But, with every merit, there are also a few demerits. Annotations are undoubtedly very useful in the documentation of code, yet they take up a lot of space due to their comprehensive nature.

Configuration of Tools

This section will help us in understanding the configuration and use of automated tools for developing a strong system of check and balance.

The quality of code and how good or bad it is, is a subjective view. Professionals like us determine the quality of the code and how clean it is, or it should be. This is because the code is made easy to read and understand, and our judgment is the validation of its quality. The following points regarding quality should be considered while reading the code in order to have a critical analysis:

- Comprehension of other programmers
- Easy to follow
- Understanding of domain of the problem
- Adaptation of new team members

In addition to proper indentation, formatting, and consistency, another attribute is required for code to be considered clean. This trait is mostly taken for granted and considered a powerful unit in the system of repetitive checks. Programmers like us do not wish to waste our time in searching for the problem. Rather, we want to focus purely on creating and implementing a solution. We want a code that can be understood easily by taking a glance at it so that our time can be consumed on providing valuable results efficiently.

For this purpose, a system of checks should be integrated that must be automated. The build should fail if these checks do not pass. By adopting this approach, the build will only be functional after passing through the system of checks. Thus, experienced or leadership intervention will be reduced, and the team members will have a reference to the logic followed and an idea regarding how to resolve the issue of build failure.

Mypy and Its Use

Mypy (http://mypy- lang. org/) is used mostly for type hinting. In Python, it is considered the main tool for optional static type checking. The use and feature of Mypy is that after installation, it starts to analyze and check any inconsistency present in the code. If type annotations are utilized in a code, then mypy can be run to type check the code and find common bugs. Mypy is a static analyzer, and therefore, type annotations do not interfere with the code and only act as a hint for Mypy. A basic advantage is the early detection of bugs, but a disadvantage is the detection of false positives.

It can be installed by using pip. It is also advised to use it as a dependent for a project on the setup file. As soon as it is installed, the following command should be run:

```
$ pip install mypy
```

Following this command will result in a report consisting of all the conclusions in the type checks. An important thing to consider here is that we should try to follow the report provided by this command to the maximum extent. This is because most of the time, this report provides necessary information for avoiding any unwanted errors timely. As discussed above, a disadvantage of using this command is that this tool might report a false positive. If it does so, it is a good approach to use the following marker as a comment to ignore the false positive.

```
type_to_ignore = "something" # type: ignore
```

Pylint

A good characteristic of Python is the presence of multiple tools for inspection and testing the structure of the code that are all in compliance with PEP-8, i.e., pycodestyle, Flake8, etc. In addition to this, these tools are not only easily configurable but also simple in application. A compact and less flexible tool among them is Pylint, which can be installed in the code easily with the help of a pip command which is as follows:

```
$ pip install mypy
```

The code, after installation of the pylint tool, will be verified.

Moreover, it is also possible to use Pylint by the configuration of the pylintrc file. The file will allow us to set rules and give a parameter to others.

Automatic Checks Setup

In the programming language Unix, a common method of compiling, running, etc., is done by the use of its most dominant tool, i.e., makefiles. For the purpose of automatic checks of formatting and structure in the code, makefiles can be used in our builds. A main advantage of makefiles is that it can be configured easily with some commands.

One recommended way of implementing this tool in code is to split the work-load into three different stages. In the first stage, the test that the tool will define will only be responsible for measuring the target that is achieved in the corresponding test. In the next stage, one specific test will be defined, and it will come to feature a target that needs to be achieved by the test. Finally, the previous two stages will need to be executed concurrently. The following lines of code demonstrate this implementation.

```
typehint:

mypy src/ tests/

test:

pytest tests/

lint:

pylint src/ tests/

checklist: lint typehint test

.PHONY: typehint test lint checklist
```

The command to run here would be following,

```
make checklist
```

The first step it would take would be a compliance check with the guidelines of coding. Afterward, it will check the use of types, and then finally, tests will be run.

The benefit is that if any one of these steps fails, the entire build will fail.

In addition to these automatic checks, a good approach is the adaptation of a convention and coding structure by a team. Here, it is important to mention that tools such as Black (https://github. com/ambv/black) format a code automatically. A unique approach to formatting, including being dogmatic and deterministic, sets Black apart from all the other formatting tools.

This can be illustrated with a simple example. In Black, we observe that a string is always double-quoted. This format is followed by the order of its parameters as well. An advantage to this rigidity is the minimal difference in the design and style of the code. In this scenario, the pull requests will be the only place where the changes would emerge with the genuine modifications granted if the code follows the same style. Formatting the code with the tool reduces stress and extra burden and allows the developer to shift his focus to the actual issue at hand.

The line size is the only thing that can be fashioned, and the rest can be rectified as per the standards of the build.

Consider the following example, which does not follow the conventions of black but is correct in terms of PEP-8

```
def my_function(name):

"""

>>> my_function('black')

'received Black'

"""

return 'received {0}'.format(name.title())
```

The command can be run now, for the purpose of formatting,

```
black –l 79 *.py
```

Afterward, it can be observed that the tool has given us the following result,

```
def my_function(name):

"""

>>> my_function('black')

'received Black'

"""

return "received {0}".format(name.title())
```

If we include some more sophisticated components to this block of code, then sure, the code will essentially become a little more sophisticated, but the core concept will remain unchanged. This philosophy was widely practiced by a programming community by the name of '**Golang**' back in the day. It would not be an exaggeration that this community is responsible for creating a programming module, which has now become a standard library in many programming languages, the '**got fmt**' library. This library is responsible for code formatting while considering the programming conventions being used.

Synopsis

The purpose of this chapter is to give us a basic comprehension of a clean code, its implementation, and its decisive nature regarding the quality of a build. From here on, a guiding principle is set for us, which will help us understand the rest of this book.

In addition to this, we also realize that clean code is an idea beyond the code's structure and formatting. Although structure and formatting are integral components of a clean code yet, there are numerous factors included in the definition and concept of a clean code. Clean code, as understood above, depends on readability, maintenance, and effective integration and communication of our ideas and purpose in the code. In addition to this, the reduction of technical debt is also a key ingredient for deciding the quality of the code.

Moreover, it was also learned that following a set of principles, guidelines, formatting, and structuring style is also a component of a clean code. For this purpose, our understanding of automated tools and their integration and utilization in the build for automatic checks is a factor to be necessarily considered.

The next chapter focuses on Python-specific codes and the method of expression of our ideas in idiomatic Python. The idioms in Python will be explored to understand the compact and efficient aspect of the code. Understanding the ideas and methods for accomplishing a task in python will allow us to distinguish the attributes and qualities of this language from other programming languages.

Chapter 2: Idioms in Python

This chapter will focus on the methods of expression of ideas in python. Python provides its own mechanism to accomplish tasks, different from the method of accomplishment in traditional programming languages like C++ and Java.

The use of idioms in programming is a particular method of writing a code to achieve a specific result. Idioms follow the same structure and have an attribute of repetition in them. Sometimes, idioms are referred to as patterns, but we will see further in this book that they are not patterns. The differentiating aspect between idioms and designed patterns is that patterns are sort of independent from the language and cannot be translated into the code directly, whereas idioms are coded. When our goal is to perform a specific task, the method to be used is the application of idioms.

As discussed above, idioms are code. Therefore, they must be written in the programming language, which is utilized due to their language dependency. Resultantly, we realize that every programming language has its own set of idioms. Being idiomatic means that a code follows these idioms. This phenomenon in Python is referred to as Pythonic.

Writing the pythonic code at the start has multiple reasons and benefits. Performance is enhanced when the code is written in an idiomatic way. The code is easier to understand and is very condensed. The attributes mentioned before are the things we require in order to make our work and build more efficient. In addition to this, we have discussed previously that a team of developers should follow a similar pattern and structure of the code in order to spend less time understanding the code itself and keep their focus on the issue at hand. Pythonic code helps in this regard.

Understanding of Indices and Slices

Some data structures or types in Python allow accessing their elements by indexes. This feature is not only limited to Python, but other programming languages also contain this attribute. One more similarity between Python and other languages is the placement of the first element in the index at number zero. The difference here is the access of elements in different orders for which Python provides extra features when compared with other languages.

This can be explained with an example, suppose we are asked to access the last element of an array in C. How would we do it? Considering the same method as in C, we would position the element in the length of the array minus one. A better way to do this would be the use of a negative index number. This would start its count from the last. It can be elaborated further with the following command:

```
>>> my_numbers = (4, 5, 3, 9)

>>> my_numbers[-1]

9

>>> my_numbers[-3]

5
```

We can also obtain more elements instead of only one by using the slice. This can be demonstrated as follows:

```
>>> my_numbers = (1, 1, 2, 3, 5, 8, 13, 21)

>>> my_numbers[2:5]

(2, 3, 5)
```

As seen from above, the syntax in the square bracket means that we would resultantly get all the elements in the tuple. The elements would start from the index of the first number and end at the index of the second one. An important thing to remember here is that the first number will be included in the result whereas, the second number will not be included in the list, and this is how slices operate in Python.

In the example below, we can clearly see that either the start interval or the stop interval can be skipped.

```
>>> my_numbers[:3]

(1, 1, 2)

>>> my_numbers[3:]

(3, 5, 8, 13, 21)
```

```
>>> my_numbers[::]

(1, 1, 2, 3, 5, 8, 13, 21)

>>> my_numbers[1:7:2]

(1, 3, 8)
```

The first example demonstrates how it will get all the information up to the index in position number three. Whereas, in the second example, it can be observed that it will get all the numbers from position three until the end, and position three will also be included. In the second to last example, a copy of the original tuple is created as both ends are excluded.

A third parameter which is a step, is included in the last instance. What this indicates is the number of elements to jump when iterating over the interval. It means that jumping by element two, the result of elements between positions one and seven, will be given.

Taking all these cases into consideration, we can deduce that we are passing a slice when we pass intervals to a sequence. A key point to note here is that a slice is a built-in function of Python. It can be built and passed by us directly.

```
>>> interval = slice(1, 7, 2)

>>> my_numbers[interval]

(1, 3, 8)

>>> interval = slice(None, 3)

>>> my_numbers[interval] == my_numbers[:3]

True
```

We can notice now that when one of the elements, i.e., start, stop, or step, is missing; it is considered none.

Creation of Sequences by Ourselves

A magic method named _getitem_, is the reason why our function discussed above works. When something similar to myobject[key] is called, then this method is called by passing the key inside a parameter. When we want to implement both _getitem_ and _len_, we use an object called a sequence. Due to this nature of the sequence, it can be iterated over. In the standard library, lists, tuples, and strings are examples of sequence objects.

Building sequences or iterable objects is the focus of chapter 7. Right now, we are more focused on getting particular elements from an object by using a key. In order to follow a Pythonic approach for the implementation of _getitem_ in a custom class of our domain, we would have to take into account some considerations.

We might have to delegate the behavior as much as possible to the underlying object if our class is a wrapper around a standard library object. What it means is that we should call all the same methods on that list if our class is a wrapper on that list. This would, in turn, ensure compatibility. The following listing shows the wrapping of an object to a list and how we can delegate it to its corresponding version on the list object.

```
class Items:

def __init__(self, *values):

self._values = list(values)

def __len__(self):

return len(self._values)

def __getitem__(self, item):

return self._values.__getitem__(item)
```

Encapsulation is used in the example above. One more way of performing the same would be by the use of inheritance. In inheritance, we extend the collections.userlist base class, with the concerns and cautions mentioned in the previous part of this chapter.

You must keep the following points in mind when building custom sequences to use in Python (it is also worth to mention that many of the sequences that you will build on your own will not feature a dependency on any native object of the class):

1. When performing an index operation on data contained within a

specified range, the output needs to have the same type as the one which the class is using as well.

2. Carefully consider the range which is given by ‘**slice**.’ In addition to this, users are advised to be very careful with the programming semantics in Python as well.

Using Context Managers

In this section, we will talk about a very productive feature in Python that improves the code's overall quality and effectiveness. This feature is known as ‘**Context Managers**.’ Generally, the way context managers work is based on the fundamental principle of recognizing patterns inside the code and then responding to them appropriately. Saying ‘patterns’ is a little ambiguous. When we talk about the patterns in code, we are essentially referring to the various conditions that have been defined within the sections of the source code. To elaborate, any portion of the application where we want to execute a block of code is governed by two types of conditions, i.e., preconditions and postconditions. These conditions make up the patterns in code, and context managers can interpret the conditions we have set for the code to be executed before a specific action is executed and after a specific action has finished execution.

Generally, context managers are usually implemented for very specific tasks, such as ‘resource management’ in code. To understand, let’s briefly look at an example. Let’s say that when we specify an action in our code, let’s say opening files from a specified directory, this task uses up system resources. Once the code is done processing the file which it opened, we don’t want to leave the file to remain open as it would hog precious system memory and other resources. Thus, we want the code to close the file once it has no use for it anymore. This can be done by using context managers, which will interpret the preconditions for opening the file and the postconditions for closing the file (which would be the finishing of the processing task).

Implementing Context Managers in Code

The implementation of context managers is not that difficult. We just need to have the right ingredients to perform a proper and successful implementation. First, we need ‘magic methods,’ namely, the ‘**__enter__()**’ and ‘**__exit__()**’

methods. Once we have these methods available in our code, the object will be capable of using the context manager tool's protocol. There are other ways in which context managers can be implemented, but most of them are usually complex and confusing. The approach we just discussed is the simplest and easy way of implementing context managers. In addition to using magic methods, we can also take advantage of the ‘contextlib’ library module's functionality, which features functions and objects that can help implement context managers.

Here’s an example of using a decorator to implement a context manager (we will explore more about decorators in detail in chapter 5 of this book).

```
import contextlib

@contextlib.contextmanager

def db_handler():

stop_database()

yield

start_database()

with db_handler():

db_backup()
```

In order to use the ‘**contextlib**’ module to implement a context manager, we can use the class ‘**contextlib.ContextDecorator**’ for this purpose.

```
class dbhandler_decorator(contextlib.ContextDecorator):

def __enter__(self):

stop_database()

def __exit__(self, ext_type, ex_value, ex_traceback):

start_database()

@dbhandler_decorator()
```

```
def offline_backup():

run("pg_dump database")
```

Underscore Conventions in Python

When working with Python, we occasionally come across several conventions and even implementations that utilize ‘**underscores**’.

Unlike other programming languages where objects can have three potential properties, i.e., private, public, or protected, in Python, the objects are given the ‘public’ property by default. Here’s a demonstration that illustrates this property of objects in Python.

```
>>> class Connector:

... def __init__(self, source):

... self.source = source

... self._timeout = 60

...

>>> conn = Connector("postgresql://localhost")

>>> conn.source

'postgresql://localhost'

>>> conn._timeout

60

>>> conn.__dict__

{'source': 'postgresql://localhost', '_timeout': 60}
```

In this example, we see that the code is creating an object by the name of ‘**Connector**,’ and this object is appointed two attributes, namely ‘source’ and ‘timeout.’ By default, the first attribute is set to ‘public’ by Python, but if we query the second attribute, we will find that its property is set to ‘private.’

Although this attribute is private, we can still access it without changing its property.

Thus, we need to use underscores to limit the details that are exposed to a caller. No matter what we do, we can never truly set any object or its attributes to private. Let’s say that we want to set the ‘**timeout**’ attribute of the ‘**Connector**’ object to ‘private.’ To do this, we will add two underscores beside the timeout attribute as a prefix. The following lines of code demonstrate this.

```
>>> class Connector:

... def __init__(self, source):

... self.source = source

... self.__timeout = 60

...

... def connect(self):

... print("connecting with {0}s".format(self.__timeout))

... # ...

...

>>> conn = Connector("postgresql://localhost")

>>> conn.connect()

connecting with 60s

>>> conn.__timeout

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'Connector' object has no attribute '__timeout'
```

Even though the attribute's property has been set to private, we should expect that when we try to access this attribute, we would be given an error saying that we don’t have permission to access it or it simply can’t be accessed. This does not happen when we actually try to do that. Instead, the error raised by the compiler tells us that the attribute doesn’t exist in the first place. This should not be the case because we definitely know that the attribute was created and is currently being used by the object, but then why would the compiler give us an illogical error instead of the one that should have happened. The only reason for this is that using double underscores to set the attribute's property to ‘**private**’ did something else.

If we pry further into this matter, we can find out that when we use two underscores to specify the private property, what Python actually does, in this case, is change the attribute’s name. This phenomenon is commonly referred to as ‘**name mangling**.’ So, the two underscores provided to the attribute during its definition ended up creating an attribute with a different name. So, when we try to access the attribute with the name which we specified during its definition, we are returned with an error that it does not exist, and this exception now makes sense since we are querying the attribute with the wrong name in the first place. So, the attribute which was created has a different name from the one we specified, and the attribute we are trying to access doesn’t even exist in the first place since it was never created with the name we specified.

The name chosen by the Python when we use two underscores follows this convention **“_<class-name>__<attribute-name>.”** So, keeping this convention in mind, the name which would be set by Python during the attribute definition would be ‘**_Connector__timeout**.’ Now, let’s try to access this attribute using this name and see if it exists or not.

```
>>> vars(conn)

{'source': 'postgresql://localhost', '_Connector__timeout': 60}

>>> conn._Connector__timeout

60

>>> conn._Connector__timeout = 30

>>> conn.connect()
```

```
connecting with 30s
```

We successfully access the attribute once we use the new name. Thus, we should always be careful when using underscores to set the properties of objects and attributes and be vigilant of potential side-effects of using them as well to avoid being clueless as to what is causing the error in our code.

Making Objects That Can Iterate

In Python, iteration is a very important feature and can allow for cleaning code that performs a specific action repeatedly but in a sloppy manner. For instance, instead of having to write each line of code to display a number, we can create an object, store all the numbers we want the code to display in this object, and then make the object iterate. In this way, as it iterates through each cycle, it will take the first element inside it generate a copy of it to pass to a method or function, then the next element and the next, and so on. In this way, we can easily automate a repetitive task, thus making the code cleaner, more concise, and manageable.

Not all objects can be iterated freely. Usually, experienced programmers would use magic methods to create an iterable object, but if a normal object is asked to iterate, then it would simply call and use the function '**iter()**' to do so. However, even if the '**iter()**' function is called, if the compiler does not find the magic method **'__iter__()'** anywhere in the object, then the iterate instruction will just be ignored. However, if the '__iter__()' method is indeed available, then the iterate instruction will be performed on the object.

Here's a demonstration of how to create an object which has the capability to perform iteration.

```
from datetime import timedelta

class DateRangeIterable:

"""An iterable that contains its own iterator object."""

def __init__(self, start_date, end_date):

self.start_date = start_date

self.end_date = end_date
self._present_day = start_date

def __iter__(self):

return self

def __next__(self):

if self._present_day >= self.end_date:

raise StopIteration

today = self._present_day

self._present_day += timedelta(days=1)

return today
```

The object contains elements specifying the dates. When we issue an instruction for this object to iterate, it will produce the dates which we have specified according to the intervals. Here’s the output of this block of code.

```
>>> for day in DateRangeIterable(date(2018, 1, 1), date(2018, 1, 5)):

... print(day)

...

2018-01-01

2018-01-02

2018-01-03

2018-01-04

>>>
```

We will learn more about iterating objects in the upcoming chapters in detail.

Chapter 3: Characteristics of Good and Clean Code

In this chapter, we will discuss different principles that make up good software design. Good quality software should be worked on these thoughts, and they will serve as a design apparatus. This does not mean that all of them should be applied, in reality, some of them depend on the context and are not always useful to apply. Some of them represent different points of view, such as the case with the **Design by Contract (DbC)** approach, as opposed to defensive programming.

The objectives of this chapter are as follows:

- ➢ To understand the concept behind strong software
- ➢ To learn how to deal with incorrect data during the workflow of the application
- ➢ To design maintainable software that can easily be increased and adapted to new requirements
- ➢ To design reusable software
- ➢ To write the powerful code that will keep the productivity of the development high

This chapter is focused on the designed principles at a higher level of abstraction. The goal is to make the code robust and write it to minimize defects and make it as evident as possible. The techniques that we will be discussing in this chapter include.

- · Common characteristics of great code design by contract
- · Design of contract-conclusion
- · Defensive programming
- · Separation of concerns
- · Acronyms to live by
- · Composition and inheritance

Common Characteristics of Great Code

The Design by Contract (DbC) Feature

When we design software, we divide it into different layers or components. Every layer or component has a specific responsibility, and as such, we need to assign these components and layers their respective tasks. In addition, to make sure that the program can work properly, we design an ‘interface’ that allows the client class to access the features of the components themselves. This ‘interface’ is known as the ‘Application Programming Interface.’

During the process of creating an API for the program, the developers need to take notes of what type of input the API will expect, the output of the API, and any ‘side-effects’ that need to be taken into consideration as well by the code. Simply making personal notes will not force the program's components to follow them; it’s like asking a chicken to fly simply because we wrote in our notes that it has wings. Things don’t work like that, in order to ensure that the API works correctly with the code and the client class, we use a concept known as ‘contracts’ in our code design. This is the basis of the ‘**Design by Contract**’ software code design.

In this software code design, a contract is basically a set of ‘rules’ that need to be followed by the components that are connected to each other. A contract has four fundamental aspects, which are the following:

- Preconditions
- Postconditions
- Invariants
- Side-effects

Preconditions

Preconditions are basically a set of conditions that need to be fulfilled before a method (that is in a contract) is allowed to execute. In other words, the code would check to see if the conditions defined by the contract are being fulfilled or not. If the conditions are fulfilled, then the code will execute the corresponding method. Otherwise, the compiler will skip this method and move on to the next block of code.

Postconditions

Postconditions are basically those set of conditionals that are applied to the method. Once the method has finished execution, it then refers to the postconditions, and if the postconditions are being fulfilled, then the next plan of action is executed in the code. Otherwise, the code is halted, or a specified task is not executed, and instead of that, some other code is executed. When we implement postconditions, we can encompass the methods and their corresponding arguments and values under conditionals and bind the system's preceding state and the current state of the system with conditionals by using postconditions. Postconditions serve as a checkpoint to ensure that the classes are upholding the contractual design and the conditions which have been fulfilled for the preconditions haven’t been modified. Similarly, postconditions serve as the link between the client class requesting access to a specific portion of the code. So, if the client class meets the postcondition requirements set forth by the preconditions, and the method does not face any hiccups during execution, the client class’s request is granted by the postcondition.

Invariants

Just as the name suggests, ‘invariants’ are elements that don’t change or have the ability to be modified when their corresponding function is being executed. Thus, invariants are used to determine whether the logic of the function being executed is correct or if it is flawed.

Side-Effects

Side-effects don’t have any other meaning in this context than the one found in the dictionary. We won’t go into details about ‘side-effects’ since it would derail the discussion. A programmer who is personally writing his code will be well-aware of any side-effects of any techniques or elements he is using in his code. If the code has any side-effects, they can be recorded in the ‘docstring’ of the program.

Contractual Relationship in Python Code

The best way to deal with approving this is by adding control frameworks to our procedures, limits, and classes, and if they fail, raise a Runtime Error unique case or Value Error. It's hard to devise a general norm for the correct kind of unique case, as that would depend upon the application explicitly. These, as of late, referred to as unique cases, are the most typical kinds of exceptions, yet if they don't fit exactly with the issue, making a custom unique case would be the best choice. We may similarly need to keep the code as restricted as normal in light of the current situation. That is, the code for the preconditions in a solitary part, the one for the postconditions in another, and the focal point of the work detached. We could achieve this division by making more unobtrusive limits, anyway, in a couple of cases realizing a decorator would be an intriguing choice.

Final Thoughts on the DbC Code Design

The thing that makes the DbC code design so useful is that if the code encounters any problem, the developer can easily navigate to the root cause without wasting too many resources on debugging the entire source code. To elaborate, if a code is based upon the ‘Design by Contract’ architecture, then this means that different components of the code have contracts with each other. When the compiler raises an error message when the code is being tested, then a contract between some components of the code must have been broken. In this way, the developer can easily pinpoint the part of the code, causing this runtime error and figure out the element responsible for the breach of the contract.

In conclusion, implementing this software code design will make our application’s source code more sturdy and less vulnerable to encountering errors. To accomplish this, this design forces every component within the source code of the application to exercise the limits in which they are bound to function in as well as holding up a few invariants. If the software components do not exceed these functioning limits and the invariants defined by them are upheld, the code will have no logical inconsistencies that could lead to runtime errors.

As praised as this design is and the benefits it provides are indeed not to be taken lightly. If implementing it would be as simple as it seems, then using the ‘DbC’ design would be standard in today’s programming practices. But that’s not the case, and there’s a good reason for it. Implementing the ‘DbC’ code design increases the developer's workload by two-fold because, for proper implementation, the developer has to define the program's core logic. Still, he also needs to define the contracts between the software components as well properly. Moreover, to ensure that the defined contracts don’t have any problems, the developer also needs to implement ‘unit tests’ to validate these contracts to avoid unnecessary complications in the future. The amount of effort required by the developers and programmers is no doubt a lot when considered what they are being asked to deal with at face value, but the results are also nothing to scoff at. The quality of the code as a result of this practice is very commendable and clearly states that a lot of effort went into the development of the application.

The Defensive Programming Code Design

This section will briefly talk about another design that can be implemented in our application's code and this design is '**defensive programming**.' Just as the name suggests, the core principle of 'defensive programming' is to ensure that each software component can 'defend' against any potential inputs that are considered 'invalid.' This design approach is quite different from the one we just talked about, i.e., 'DbC.' In the 'DbC' design, the software components are provided with contracts and if any of these contracts are broken (i.e., a condition that is set by the contracts are not fulfilled), then an error is raised by that part of the code, but in 'defensive programming' the components of the code block any prospects of errors that might be raised by implementing appropriate logic to block input which is not considered as valid parameters.

The main aspect which makes defensive programming a technique worthy of mention is that it can be used to bring out the full potential of other code design implementations as well, but that is a topic that is currently outside the scope of this book. If this concept intrigues you, then you will have ample information about unifying different design approaches with 'defensive programming' on community forums on StackOverflow.

Before we move on to the next topic, let's first understand how the software components actually '**defend**' themselves against errors in the first place. Well, the answer is very simple, the developer programs the code in such a way that any possible error that the code might encounter has appropriate counter-measures already implemented. So, most of the errors that the programmer knows the code might encounter are already dealt with by implementing corresponding logic to deal with these error exceptions.

Techniques for Error Handling

In programming, 'Error Handling' is a concept taught as one of the fundamental pillars of developing applications. The reason for this is, when we are coding, we assume the interaction from the application user as well. For instance, if we are building a calculator application, we will write code lines that will prompt the user of this application to input numbers and the mathematical operation they want to perform on these numbers. There are cases where the user may make a mistake while inputting an appropriate mathematical operation, if that happens, then the code will not know what to do, and thus the program will break. We need to account for such situations and implement lines of code that handle these errors if they arise, so the user knows that they made some type of mistake and the application does not break during runtime.

So, error handling techniques are absolutely necessary if we want the code to keep running if the error encountered is really simple (like the input of an incorrect mathematical operation which can be remedied by telling the user ‘Syntax Error’ and the user will then go back and fix the mistake) or in other cases, forcefully terminate the code if the encountered error is complicated.

In this section, we will briefly discuss some of the most popular ‘error handling’ techniques, three of them to be exact:

- Value substitution
- Error logging
- Exception handling

Value Substitution

There are cases where the code outputs an incorrect value because of encountering an error or just break completely. One technique that handles such types of errors is ‘value substitution.’ In this technique, we take the possibility into account that the code might return an incorrect value due to an error and then specify another value that will be used instead. In other words, if the code encounters an error affecting its output, then instead of returning an incorrect value, it returns the value which we specified. Having a value that we define by ourselves is better than having the code deal with a value that is outside of our expectations. In essence, the ‘value substitution’ technique allows us to ‘substitute’ the value, which is influenced by an error with one that will not cause any major disruptions in the code.

Even though this technique is actually a really good option for handling errors in code, unfortunately, it cannot be used all the time because any careless implementation might even lead to bigger complications. The ‘value substitution’ technique should only be used when we are sure that the specified replacement value won’t be disruptive. If the substituted value also causes complications in the code, using this technique will be pointless. If we use the ‘value substitution’ technique to handle errors, then sure, the application will not break, but it doesn’t also mean that the program is working correctly since it didn’t break when encountering an error. Thus, the developer needs to carefully maintain the balance between correctness and robustness in his code when implementing this error handling technique.

Since we already know that this technique's core functionality is to swap out the erroneous value with some other value, it makes no sense to implement it in a program dealing with sensitive data.

It is recommended to use this technique when we are sure that the code will have no problems in dealing with the default values of some input. So, if the program is missing an input, instead of raising an error and stopping the code execution, it will substitute the empty value with its default value and continue to work. For instance, we can implement value substitution to cover for errors that might arise due to some undefined variable, incomplete configuration files, or even parameter values that have been left out for some functions.

Here’s a demonstration of a code implementing the value substitution technique.

```
>>> configuration = {"dbport": 5432}

>>> configuration.get("dbhost", "localhost")

'localhost'

>>> configuration.get("dbport")

5432
```

The implementation of this technique is also more or less the same when we are dealing with some variables that might have undefined values or simply missing values.

```
>>> import os
>>> os.getenv("DBHOST")

'localhost'

>>> os.getenv("DPORT", 5432)

5432
```

If we look at any one of these demonstrations, we will notice that the function expects two parameters. If it is only provided with one parameter, then '**None**' will be used as the second parameter because we use value substitution to define this value when the original value is missing in the code.

Using this technique to specify any 'default value' to be used by custom created functions is also possible. The following lines of code demonstrate how to do this.

```
>>> def connect_database(host="localhost", port=5432):

... logger.info("connecting to database server at %s:%i", host, port)
```

Exception Handling

Error handling should be done after carefully considering the entire context in which the program will function. For example, it makes sense to substitute the erroneous values with non-disruptive values because we know that the program's core functioning will not be affected as much. There are also cases where it is better to just stop the code from executing rather than have it work with incorrect assumptions and produce wrong results. This would make the entire program itself a failure since it is doing the wrong job. If your program encounters the latter scenario, then the only thing you can do as a programmer is to write a few code lines that will tell the user that a fatal error has occurred. That way, they can try to fix it from their end if possible or simply contact the developer and make him aware that the program is misbehaving.

However, the concept of error handling is not as simple as it might seem from our discussion up till now. Sending data to other components and receiving data from corresponding software components isn’t the only function of code. Certain portions of code have certain logic as to how they are implemented and work based on this logic. In addition, they can also be connected to components outside the application as well, for instance, connecting to a database to load data from or even establishing a connection to a printer to send instructions. As such, it is incorrect to implement error handling from a one-dimensional perspective, and we also need to take such elements and factors into account, which is the domain where the ‘Exception Handling’ technique thrives.

Since we have already established that certain parts of the code are connected to external elements, this external element is also potentially the source for an error happening inside the code. For example, let’s say that the code we are working with has a function call, and this function call is misbehaving because of an external element, thus causing an error. Considering the nature of this error, we know that there’s no fault in the function call. If we want the code to handle the error properly, then we will have to deal with the external component causing it. We must not confuse this with resolving the error itself. Instead, what we want the code to in such scenarios is to simply relay the error message to the rest of the application as clearly as possible. This course of action in programming is known as ‘exception handling.’ This notifies the rest of the code of the problem, and the execution temporarily halts or continues if it is possible. As a result, if the program is able to continue execution, then the core logic of the code will remain untampered.

Now let’s discuss two scenarios where using exception handling is the right choice. Like value substitution, this technique also has some caveats if used carelessly, the primary one being it makes code encapsulation considerably weaker.

Considering the Degree of Abstraction When Using Exception Handling

When we define an ‘exception’ for a function in case it encounters an error, the exception itself cannot be based on a logic that is different from the logic the function is using. In other words, the exception needs to be defined on the same logic that was encapsulated on the corresponding function.

Thus, it is very important to carefully consider the level of abstraction in the code where exceptions are being raised before implementing any counter-measure.

The following block of code demonstrates a scenario where the method ‘**deliver_event**’ is raising an exception. Considering the degree of abstraction in the code, we apply an appropriate exception handler, as demonstrated below.

```
class DataTransport:

"""An example of an object handling exceptions of different levels."""

retry_threshold: int = 5

retry_n_times: int = 3

def __init__(self, connector):

self._connector = connector

self.connection = None

def deliver_event(self, event):

try:

self.connect()

data = event.decode()

self.send(data)

except ConnectionError as e:

logger.info("connection error detected: %s", e)

 raise

except ValueError as e:

logger.error("%r contains incorrect data: %s", event, e)
```

```
 raise
def connect(self):
for _ in range(self.retry_n_times):
try:
self.connection = self._connector.connect()
except ConnectionError as e:
logger.info(
"%s: attempting new connection in %is",
e,
self.retry_threshold,
)
time.sleep(self.retry_threshold)
else:
return self.connection
raise ConnectionError(
f"Couldn't connect after {self.retry_n_times} times"
)
def send(self, data):
return self.connection.send(data)
```

To understand what’s happening in this block of code, we need to focus our attention on the ‘**deliver_event()**’ method, where the exception is being handled. Upon carefully looking at the code, we come to know that there are primarily two errors being raised, namely:

1. **ValueError**
2. **ConnectionError**

Now that we have some clues to work with let’s do some detective work. We have a ‘ConnectionError’ and a ‘ValueError’ exception being raised. Now we have to figure out the correct places where these exceptions will be handled. Since the exception, ‘ConnectionError’ obviously refers to a failure to connect to an external component, let’s find the method responsible for performing this task. This leads us to the method ‘**connect**,’ so the ‘**ConnectionError**’ exception needs to be handled in this method. One way of handling it would be to specify the method to try connecting to the target again and specify the number of retries it should make.

On to the ‘ValueError’ exception. Since we know it’s an error related to ‘values,’ keeping in mind the context of the program, we are led to the method ‘**decode**.’ So, the ‘ValueError’ exception will be handled in the ‘**decode**’ method.

Now let’s make things easier and implement these exception handles into their corresponding parts, i.e., specify the exception handler for ‘ConnectionError’ and the appropriate exception handler for ‘ValueError’ in their respective methods.

This is the demonstration for handling the exception ‘ConnectionError’ in the ‘**connect**’ method.

```
def connect_with_retry(connector, retry_n_times, retry_threshold=5):

"""Tries to establish the connection of <connector> retrying

<retry_n_times>.

If it can connect, returns the connection object.

If it's not possible after the retries, raises ConnectionError

:param connector: An object with a `.connect()` method.
```

```
:param retry_n_times int: The number of times to try to call

``connector.connect()``.

:param retry_threshold int: The time lapse between retry calls.

"""

for _ in range(retry_n_times):

try:

return connector.connect()

except ConnectionError as e:

logger.info(

"%s: attempting new connection in %is", e, retry_threshold

)

time.sleep(retry_threshold)

exc = ConnectionError(f"Couldn't connect after {retry_n_times} times")

 logger.exception(exc)

raise exc
```

This is the demonstration for handling the exception ‘**ValueError**’ in the method ‘**decode**.’

```
class DataTransport:

"""An example of an object that separates the exception handling by

abstraction levels.
```

```
"""

retry_threshold: int = 5

retry_n_times: int = 3

def __init__(self, connector):

self._connector = connector

self.connection = None

def deliver_event(self, event):

self.connection = connect_with_retry(

self._connector, self.retry_n_times, self.retry_threshold

)

self.send(event)

def send(self, event):

try:

return self.connection.send(event.decode())

 except ValueError as e:

logger.error("%r contains incorrect data: %s", event, e)

 raise
```

Keeping Error Tracebacks Private

The practice of keeping tracebacks private, i.e., details about the error only available for the developer or programmer of the application, is to keep the structure or elements of the source code secure. Some brainy hackers can reverse engineer the source code of the application or find loopholes if the error message returned to the user has a lot of details that aren’t of any use to the end-user in the first place.

Aside from that, if the programmer deems that the element of correctness is more important than the element of robustness for the application, then it’s fine to let the program stop its execution when it encounters a specific error. Moreover, if you are choosing to let the application terminate due to an error, then it’s better to use a simple error message then to outline all the intricate details.

Dodge Empty ‘Except’ Blocks in Your Code

In games, having high defense stats is never a bad thing, but in real life, especially in programming, too much of anything can prove to be bad (just as how taking abnormally high doses of medicine is very bad even though the prescribed quantity is actually beneficial). To elaborate, in programming, focusing too much on the defensive capabilities of your code can also backfire and cause more problems than it would originally be handled. One such problem that stems from abnormally focusing on defense while programming is the scenario where ‘except’ blocks that have basically nothing inside them get executed without any issues. This is a waste of memory, resources, and performance.

This is possible because of the inherently flexible nature of the programming language. We can write code that has no problems if we consider its syntax but is logically incorrect. In other words, no point in executing the code when it does literally nothing. Here’s an example.

```
try:

process_data()

except:

pass
```

On the surface, we know for sure that this small block of code is fundamentally flawed, but the Python compiler sees no issue with it and executes it as it would any other instruction. What’s even more annoying is that no errors are raised when such lines of code are executed, which makes it hard to figure out what's making the program misbehave. So, whenever using error handling, make sure that there aren't any silent blocks in the program.

Here are some tips on avoiding having to deal with 'except' empty code blocks.

- Instead of targeting a general error, focus on a specific error. Programming to deal with a certain exception can help avoid empty 'except' blocks and addressing general errors all the time is not good in the first place.
- If there is an 'except' block in your code and it's empty, you can also just simply write some error handling lines of code in there. In this way, the 'except' block is no longer empty.

To get the best results, just use both of these tips side-by-side. Relying on one tip will only take you so far. Besides, it's worth the effort if you're eliminating those annoying empty blocks of code, anyway.

Never Forget the First Exception You Used

When dealing with errors in programming, it's common to fall down the rabbit hole of implementing exceptions so much that when you realize it, the first exception you used to handle the error is very different from the current exception you raised. This happens to the best of us, and instead of just saying 'be careful,' we need to have some proper guidelines when in such a situation. The tip that we can give you is that always remember the first exception you used and if this is the one that started the chain of exception handling, then always remember to refer to it in the error traceback.

In Python 3, new syntaxes were introduced, which made implementing the original exception in nested tracebacks even easier. To be more precise, this can be done from the following syntax:

```
raise <e> from <original_exception>
```

Here's a demonstration where we are using personally designed exceptions, but at the end, we are including the original exception in the traceback log.

```
class InternalDataError(Exception):
```

```
"""An exception with the data of our domain problem."""

def process(data_dictionary, record_id):

try:

return data_dictionary[record_id]

except KeyError as e:

raise InternalDataError("Record not present") from e

Always use the raise <e> from <o> syntax when changing the type of

the exception.
```

The Purpose of 'Assertions'

In programming, there is a common practice used to determine if the application has a serious problem or its logic is fundamentally flawed, which is done by using 'Assertions.'

Assertions are basically conditions that should never be fulfilled if the program doesn't have any serious problems. For example, if we use a condition to determine whether the project we built doesn't have any serious problems, then such a condition would be known as an assertion. So, if any program fulfills the conditions set by the assertion, then it means there's a serious problem with the code.

In error handling, we had the option of letting the program continue to run or have it forcefully stop. It all depended on whether we preferred robustness or correctness in our code. However, when using assertions, we don't have any such option because it is illogical to have the program continue to run when it has a serious defect. Thus, assertions are used as a fuse that prevents the program from causing any damage to the system by killing the root process or simply letting the process crash.

Keeping these characteristics of assertions in mind makes no sense to encapsulate assertions in the same logic as the portion of the code that is raising the error. In addition, assertions should not be allowed or given the power to influence the code's inherent mechanism itself.

Here’s a demonstration of an improper implementation of assertions.

```
try:

assert condition.holds(), "Condition is not satisfied"

except AssertionError:

alternative_procedure()
```

In this scenario, it would be better to have the program close when the assertion condition is not met. Moreover, always remember to log all the important details about an error in the error message so that when you read the traceback, you can easily fix it.

Understanding the Concept of Separation of Concerns

In this section, we will talk about another software code design that has various degrees of implementation. In other words, the ‘Separation of Concerns’ design is appropriate not only as a low-level design for the code but also appropriate when considering higher degrees of abstraction in the code as well.

According to this design, each major element of a program should only be concerned with their responsibilities. Before we go on and misinterpret this as insinuating that the software components unnecessarily meddle in each other’s affair, what we actually mean by this is that the collective responsibility of the code needs to split among the major components of the code. Each component is only concerned with what it is responsible for and nothing else.

We will now discuss some important elements that need to be considered when implementing a software code design.

Cohesion

The concept of ‘cohesion’ refers to the approach where the software components, particularly ‘objects’ are better if their sizes are small and the objects themselves have a clear-cut goal to accomplish within the code.

Moreover, a code is more cohesive if the objects are responsible for performing as few tasks as possible. Assigning multiple tasks to objects is detrimental to the cohesiveness of the code.

Coupling

The concept of ‘coupling’ is also primarily related to objects as well. The word ‘coupling’ itself refers to a ‘pair’, and rightfully so, it explains the effects of objects being dependent on each other. In programming, objects should be as independent as possible but having all the objects become independent is impossible. There will be some dependencies no matter what we do, but too much dependency is detrimental to the robustness of the code design.

Here are some common problems when objects are coupled tightly.

- The code is no longer reusable
- Any modifications made to one portion of the code will carry over to the other portion, making modifications more sensitive and complicated.
- The degree of abstraction in the code becomes extremely low.

Some Useful Programming Acronyms

Acronyms are a fun and convenient way to remember a lot of things. Nobody wants to memorize six entirely fancy and hard-to- articulate words or even talk about them in conversations. Acronyms help us remember a lot of different things and help us talk about them without giving the impression that we’re showing off (admittedly, nobody cares about this stuff anyway).

Here are a bunch of acronyms that will help you remember some key principles to always keep in mind when programming to ensure that your code is based on a good design.

- **D.R.Y:** The full form of this acronym is ‘Don’t Repeat Yourself.’ This acronym emphasizes the fact that your code should never be repetitive, i.e., you shouldn’t manually repeat an instruction. Instead, you should use loops or generators to do so.
- **O.A.O.O:** The full form of this acronym is ‘Once and Only Once.’ The main message being told by this acronym is basically the same as the

previous one. The important definitions, modules, or libraries being used by your code should always be kept in one location. That way, when you need to access them, you can do so in one fell swoop instead of having to direct the code to one file directory to access a module, then to another to access another module, and so on.

If we focus on the repetitiveness in the code, then it brings about some serious problems. Some of the most annoying problems that we encounter as a result of code duplication have been listed below:

· Duplicated lines of code increase the chances that the program will encounter an error. Moreover, if all of these lines are based on the same logic, then when the time comes to make a change to the logic, we will have to change the logic of each instance manually. Otherwise, the program will have bugs.

· When applying modifications to the code, it takes up more effort and time since we have to deal with duplicated lines of code.

· Because we need to change something in the code while keeping the context in mind, if there are duplicated lines or lines that have been repeated several times, then we are forced to rely on the person who is the actual developer of the code. That makes the maintainability of the program more difficult and unreliable.

To understand this better, let’s go through a quick example. Let’s say that we are given a dataset highlighting the ranking of the students along with the criteria according to which they were ranked. The threshold marks for each student to be labeled as ‘passed’ or ‘failed’ have been listed below, along with an extra ‘condition’ as well to make things more interesting.

· Awarding a total of Eleven Points for each exam that has been passed

· Deducting a total of Five Points for each exam failed

· Deducting a total of Two Points for each extra year the candidate spends in the institute

Since we know the candidate conditions to be labeled as ‘passed’ or ‘failed’, we now only have to process the points each candidate has gathered and then output their ranking. The following code demonstrates this:

```
def process_students_list(students):

# do some processing...

students_ranking = sorted(

students, key=lambda s: s.passed * 11 - s.failed * 5 - s.years * 2

)

# more processing

for student in students_ranking:

print(

"Name: {0}, Score: {1}".format(

student.name,

(student.passed * 11 - student.failed * 5 - student.years *

2),

)

)
```

This code is plagued by duplication. The following block of code demonstrates how we can fix this.

```
def score_for_student(student):

return student.passed * 11 - student.failed * 5 - student.years * 2

def process_students_list(students):

# do some processing...

students_ranking = sorted(students, key=score_for_student)

# more processing
```

```
for student in students_ranking:

print(

"Name: {0}, Score: {1}".format(

student.name, score_for_student(student)

)

)
```

This demonstration utilizes what is arguably the most basic technique to remove duplication in code, i.e., simply making an appropriate function to handle the problem, but there is no perfect solution, and each scenario requires a different approach to be used in order to be resolved. Unfortunately, going into the detail of code duplication is outside the scope of this book since we can write entire chapters on such a topic alone.

Inheritance

Object-oriented programming languages are by far the most popular languages used back in the day. Even now, a lot of different languages have been created with better features and functionalities, but back in that era, there were some problems and issues that were puzzling programmers and developers in the community. They were trying to come up with a solution for this problem by staying within the programming paradigm through the concepts of ‘polymorphism,’ inheritance, and encapsulation.

Out of these programming paradigms, the concept of ‘Inheritance’ is arguably the most widely used for tackling many of the problems they faced in their day and age. We will not go into a history lesson as to what these problems were and how they were resolved. Right now, we will only focus on Inheritance.

When we say ‘inheritance,’ the meaning isn’t too far off from its other contextual meanings. The concept of ‘inheritance’ is generally used for classes. We create a class that has all the methods, functions, and objects we want to use. Then we create another class but give it part of the elements the original class had. In this way, this class is ‘inheriting’ its properties and features from the original class. So, we term this new class as the ‘child class, derived class or subclass' and the original class as the 'parent class, base class, or superclass.' This creates a hierarchy between the classes, with the base classes ranked at the top, and the subclasses ranked at the bottom.

Even though an inheritance is a very useful feature to have in programming, it is by no means a holy grail. In fact, it has its fair share of demerits. For example, when a class is derived from a base class, the two classes become 'tightly coupled' to each other, and in clean programming, we established the fact that 'coupling' between software components should be kept to the minimum. So, we cannot blindly use inheritance whenever we want if we want to maintain a clean and good software code design.

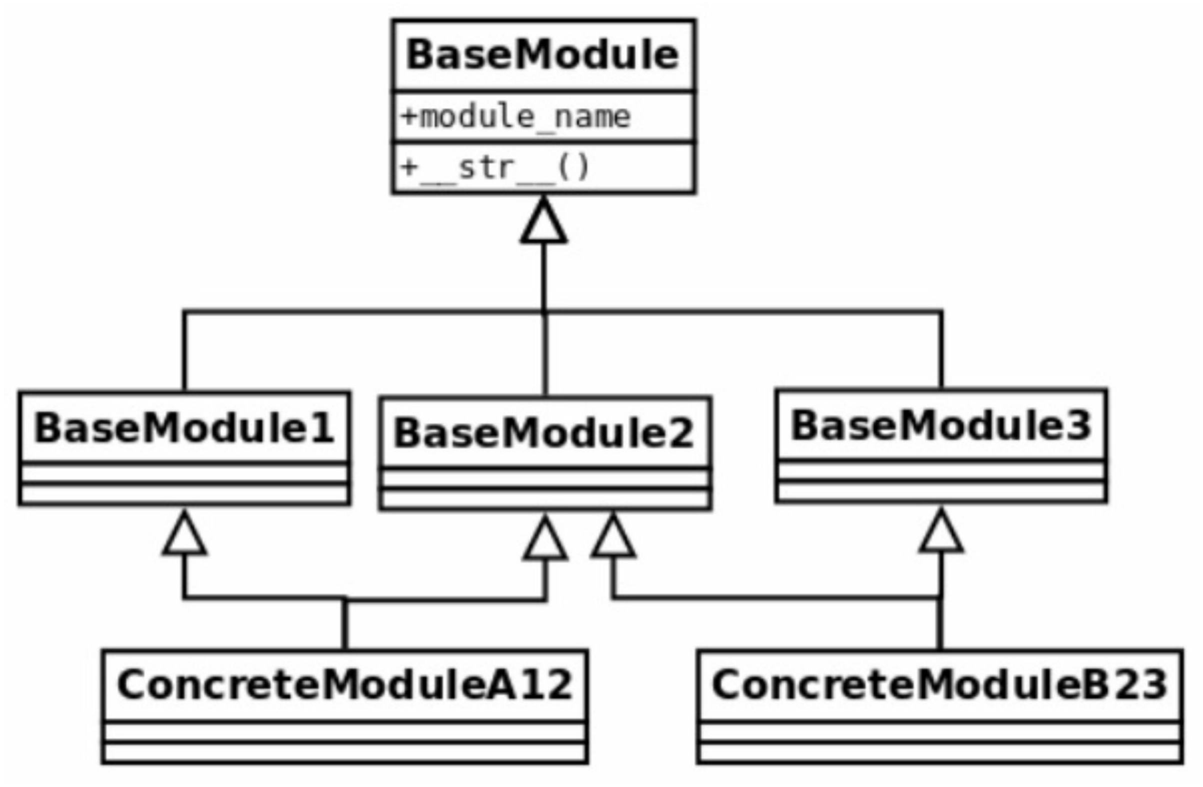
Not everything is doom and gloom. Sure, inheritance causes some problems to the software code design, but it doesn't mean that it can never be used as well. The main feature that inheritance brings to the table is 'reusability.' So, if we want to plan for our code to be reusable, then we can use inheritance in it.

Multiple Inheritance

Since we have established the fact that inheritance is a double-edged sword that can greatly compromise the code's design if used incorrectly or unwisely, one would assume that multiple inheritances are an even bigger double-edged sword with greater problems. To be very honest, this assumption does not have any flaws. If there are advantages to using 'multiple inheritance,' then there are also disadvantages to using them as well. However, the only scenario where we would have to hold the burden of multiple inheritance is when we foolishly implement it.

Method Resolution Order

One way of implementing multiple inheritance is through the concept of Method Resolution Order (also known as MRO). Here's a visual representation of how a code implementing multiple inheritance would shape its design.



Once we have the blueprint in mind, we can easily implement classes featuring multiple inheritance. The following lines of code demonstrate this.

```
class BaseModule:

module_name = "top"

def __init__(self, module_name):

self.name = module_name

def __str__(self):

return f"{self.module_name}:{self.name}"

class BaseModule1(BaseModule):

module_name = "module-1"

class BaseModule2(BaseModule):

module_name = "module-2"
```

```
class BaseModule3(BaseModule):

module_name = "module-3"

class ConcreteModuleA12(BaseModule1, BaseModule2):

"""Extend 1 & 2"""

class ConcreteModuleB23(
```

We will now run this code and check out the method that is being called by it.

```
>>> str(ConcreteModuleA12("test"))

'module-1:test'
```

As is obvious from the test, the code does not show any signs of chaos or collisions between the methods of the different classes. This is due to the fact that we are using an algorithm known as '**C3 Linearization**' a.k.a '**Method Resolution Order**.' This algorithm resolves any internal conflicts between the derived classes when they need to call the method which they inherited from their respective base class. Moreover, we can see the detail of the order through which the algorithm performs the internal resolution between the classes. The following example demonstrates this.

```
>>> [cls.__name__ for cls in ConcreteModuleA12.mro()]

['ConcreteModuleA12', 'BaseModule1', 'BaseModule2', 'BaseModule',
'object']
```

This extra information is not useless or just a one-pony trick of the C3 linearization algorithm. Instead, this provides us with very useful insight helping us to design the classes since we already know how they are going to interact within the hierarchy. Similarly, we can also effectively implement '**mixins**' with the help of this information.

Mixins

The term 'mixin' basically refers to a type of base class (a parent class or even a superclass, whatever you want to call it) that features the encapsulation of logic that is primarily geared towards ensuring the reusability of the code. By itself, a '**mixin**' base class is of no use, and the attempts to extend this class while it is alone are also fruitless efforts and practically useless. This is because the very nature of the mixin class is highly dependent on others. To elaborate, the mixin class's worth is determined by the number of classes it has to depend upon because it heavily relies on the properties, functions, methods, attributes, and objects that are defined in other classes. This inherent dependent nature of the mixin class makes it the perfect candidate to be used in multiple inheritance.

We will now briefly talk about how we can actually implement a mixin class. Let's say that we are working with a '**parser**' and some strings and the purpose of this parser is to perform iteration and give output values of the specified strings, which are separated by a character '**hyphen (-)**.' This has been demonstrated in the following lines of code.

```
class BaseTokenizer:

def __init__(self, str_token):

 self.str_token = str_token

def __iter__(self):

 yield from self.str_token.split("-")
```

If we execute this block of code, we will get the following result: exactly what we wanted.

```
>>> tk = BaseTokenizer("28a2320b-fd3f-4627-9792-a2b38e3c46b0")

>>> list(tk)

['28a2320b', 'fd3f', '4627', '9792', 'a2b38e3c46b0']
```

Now let's say instead of strings displayed in lower-case, we want them to be displayed in uppercase, and in this process, we don't want the base class to be modified even in the slightest of manner. One potential way to solve this problem would be to build a new class, but we also have to account for the fact that there could be various other classes being derived from the base class '**BaseTokenizer**' as well, and this would cause unneeded complications if we begin replacing these already existing classes. This is where the 'mixin' class comes in handy. Since we don't want to replace the existing subclasses, we can simply build a new class responsible for transforming the lowercase strings into uppercase and then add it into the existing hierarchy of classes. The following lines of code demonstrate this.

```
class UpperIterableMixin:

def __iter__(self):

return map(str.upper, super().__iter__())

class Tokenizer(UpperIterableMixin, BaseTokenizer):

pass
```

Chapter 4: The Principles of Clean Software Code

This chapter will focus on some fundamental principles that will help us in creating an elegant software design when coding in Python. In this chapter, we will explore five main principles for cleaning code. These principles are:

1. Single Responsibility
2. Open or Closed
3. Liskov's Substitution
4. Interface Segregation
5. Dependency Inversion

These principles highlight a set of practices and conventions that help increase the software code's quality. Implementing these principles in our daily coding habits allows us to avoid baking software that has clunky and messy code. These principles are sometimes easy to forget and difficult to recall for many people, so here's a trick that can help you memorize the exact order and each principle's name. The trick is to use an acronym with each letter of the acronym representing the principle's initial letter. In comparison, memorizing the acronym is much easier than recalling each principle's name and much more convenient than remembering their order. So, let's take the first alphabet from each of these five principles, from the "Single Responsibility Principle," we will take the alphabet '**S**', from the "Open and Closed Principle," we will take the alphabet '**O**', from "Liskov's Substitution Principle," we take the alphabet '**L**,' from the "Interface Segregation Principle," we get '**I**' and finally, from the "Dependency Inversion Principle," we get the alphabet '**D**.' Combining these letters, we get the acronym, "**SOLID**." In this way, recalling this acronym will help us recall each of the principles in their correct order, quite convenient.

By the end of this chapter, the reader will:

· Become familiar with the five principles of clean coding and software design.

· Be able to produce components of a software package based on the "**Single Responsibility Principle**."

- Be able to make their code easier to maintain by implementing the "**Open and Closed Principle**."
- Learn how to properly implement the hierarchies of the classes being used in the code based on an object-oriented framework. This is demonstrated by following "**Liskov's Substitution Principle**."
- Finally, learn about different designs for the software code that is robust, efficient, and most importantly, clean by following both the "**Interface Segregation Principle**" and "**Dependency Inversion**."

Now let's move on and explore the very first principle, i.e., the Single Responsibility Principle.

The Single Responsibility Principle

The SRP (Single Responsibility Principle) is a very straightforward convention. According to this principle, each class being used in the code of a software package must only be assigned to do one specific job. In other words, each class should only have one responsibility to fulfill. Thus, the name "Single Responsibility Principle." Moreover, when we are talking about classes, we are referring to the 'component' of the software, and while software has many components, this principle largely addresses classes.

Now let's talk about what it means for a class to have one responsibility. In this context, 'responsibility' refers to the task or job to be carried out by the specific class. In order for a class to handle more than one responsibility, it needs to have adequate resources to do so. This means that such a class will have various methods, function calls, objects, and so on.

When we follow the Single Responsibility Principle, we are essentially making the software easier to maintain and patch. For example, if the software is misbehaving when performing a certain action, we can easily navigate to the class that is responsible for this job. Since the task is not being properly executed, it means that something is wrong with the associated class, and thus, we can easily make changes to it. In this way, diagnosing the root of the problem becomes extremely reliable. On the other hand, let's say that each class is used in the software has many responsibilities. If a problem does occur, we will have to look over the multiple tasks assigned to each class and find the one which is the root cause for this misbehavior. This makes the software very difficult to maintain. This principle tells us that if the software encounters a single problem, the solution will most likely be making changes to a particular class. However, needing to make changes to that specific class for different reasons means we are not following the principle. The class has not been properly abstracted according to this principle.

In this way, the design of the software is built in a way that each object is responsible for doing only one thing. If every object starts doing multiple tasks, their behaviors become chaotic and unpredictable (to a certain extent), making it harder for the developer to maintain the software. Keeping this principle in mind, we should always use small classes in our software. If the class is big, it is bound to have many different objects, methods, and function calls, and restricting it to one single task becomes tricky.

The goal of the Single Responsibility Principle is to make the code cohesive. This means that since each class is responsible for a specific job and tasks are correlated to each other, we can group different classes together to increase code maintainability. This is only possible if their corresponding methods frequently use a class's properties and attributes.

A Class That Has Various Responsibilities

This section will discuss a software application that will read data from a source file (for example, a database file or a log event file, etc.) and then proceed to classify the tasks that are being detailed in the source file data.

Since a class that has multiple responsibilities does not conform to the Single Responsibility Principle, the resulting software code design will be like this:

SystemMonitor

+load_activity()

+identify_events()

+stream_events()

To clarify, ‘SystemMonitor’ is the class and ‘load_activity()’, ‘identify_events()’ and ‘stream_events()’ are its methods. Now let’s see the implementation of the code that defines the class responsible for performing the specified actions.

```
# srp_1.py
```

```
class SystemMonitor:

def load_activity(self):

"""Get the events from a source, to be processed."""

def identify_events(self):

"""Parse the raw source data into events (domain objects)."""

def stream_events(self):

"""Send the parsed events to an external agent."""
```

The ‘SystemMonitor’ class performs its tasks through its three methods (load, identify, and stream methods). In this way, a single class is responsible for performing three different tasks, and this is clearly against the Single Responsibility Principle. To be more precise, the resulting interface, which is being defined by the ‘SystemMonitor,’ consists of different methods, and each method performs its task without relying on another method from the same class. With regards to code maintainability, this is a big problem. In addition, the software code design makes the class we are using (SystemMonitor) more susceptible to errors, difficult to modify, and hard to extend its functionality to other tasks.

The concept of ‘responsibility’ basically helps us understand the fundamental reason why and how the class needs to be changed if there is a need for it in the future. So, if one of the methods is causing an error, then we need to consider the other two methods' responsibility when thinking of how to fix it. So, in this scenario, if we need to apply some modifications to the SystemMonitor class to fix an error, we will need to consider all three responsibilities. If we applied the Single Responsibility Principle, we would only need to consider only one responsibility when making changes. Moreover, if we modify certain aspects of data (especially in this case), we will be forced to make some changes to the corresponding class as well. Thus, the maintenance of the software code is relatively easier and straight-forward.

Let’s discuss this in a bit more detail while considering the example shown above. Let’s talk about the ‘**load_activity()**’ method. The responsibility of this method is to capture or simply load data from a given source. Irrespective of the way this loading task is executed, the fact remains that the method will have its own series of steps through which it completes the task. To elaborate, this method will first connect the application to the source file containing the data. Once the connection has been established, it will then carry the corresponding data over to its own application and then parse it into an appropriate format for the next methods to understand, etc. Now let's say that we want to change the structure in which the data is stored. If we change the data structure, we will also need to modify the class (SystemMonitor). Generally, this shouldn't even be a concern for us because it is simply illogical for us to change the SystemMonitor class objects simply because we are modifying the structure of data in the source file. This trend also carries on to the other two methods as well.

In this way, we can conclude that having external factors affect our code is extremely undesirable and hectic to deal with. Imagine having to tweak the code every time you make a change to something outside of the source code. Such classes are very fragile and sensitive to errors and the solution to this is to use small classes with cohesive abstraction.

Dividing the Responsibilities Among Different Classes

We will continue our previous discussion and demonstrate a solution to the problems outlined previously. As you might have guessed, the solution is to follow the Single Responsibility Principle and create different classes and assign each class only one responsibility. In this case, since we know that there are three primary functions that the software needs to perform, and each function is handled by one method of the same class, we will simply divide each of those methods into independent classes. In this way, we create a total of three classes, and the responsibilities of the original class are now divided among the three classes. So, the software code design now conforms to the Single Responsibility Principle since each class is handling a single responsibility.

Before we move on, we must first understand the classes being formed. We are fissuring the '**SystemMonitor**' class into '**ActivityReader** (contains the method '+load()'), '**Output** (contains the method '+stream()'), '**SystemMonitor** (contains the method '+identify_event()') and the '**AlertSystem** (contains the method '+run()') classes. The last class is crucial because it signals the other three classes to execute their corresponding methods.

In order to interact with each instance of these classes, we introduce corresponding objects. However, this does not mean that a change in one method will impact the others. To the contrary, each method belonging to a class is completely independent of the other methods belonging to their corresponding classes. In this way, if we modify the **SystemMonitor** class, we will not be forced to make changes to the other classes as well. Similarly, each method and class now has its own distinct identity. So, if we want to change the way the events are loaded from the source file by the 'ActivityReader' class, we can easily do so without worrying about making changes to other corresponding classes. For instance, the 'AlertSystem' class is oblivious to the fact that we are modifying something. Since it doesn't know about any changes being made, it doesn't need any tweaking. Thus, we are saved from the hassle of making changes to the 'SystemMonitor' class all because we had to modify the 'AlertSystem' class. In this way, if the relationship (or the cohesiveness) of the classes are left untouched, then this chain of modifications will never arise.

To put it into simpler terms, the modifications that we make are now treated locally (previously, it would affect every class). The impact of a modification is now negligible, and consequently, the software code is very easy to maintain.

The Open/Closed Principle

The Open and Closed Principle defines such a software code design where a user can extend its functionality (open) by adding new lines of code but cannot make any modifications to the original code (closed). This is done by implementing appropriate logic in the code.

To elaborate, if we are to follow the Open and Closed Principle (OCP), then we want our code to have the capability to be extended later on if need be. In other words, we want to have the freedom to add different functionalities on a future update to the application. So, if we encounter a problem on our domain and to fix that, the most viable solution is to introduce new functionality, then we can easily do so by adding a bunch of lines of code to the application and call it a day. However, we don't want something to misbehave when we add something new to the software. It would become a real hassle to change an original software component as we add a new software component, which would be a nightmare when it all piles up. So, the goal is to be able to add new elements to the software but not make any changes to its original functionality, thus, open for extension and closed for modification. In this way, if we add new software components to the code and find ourselves constantly making changes to the original components, the software design (or the logic) we are using is not clean or, simply put, poorly implemented.

The main focus of this principle is on classes and modules. We will discuss how the Open and Closed Principle applies to both software components in the following sections.

A Software Code Without the Open and Closed Principle

This section will demonstrate a software code that is not implementing the Open and Closed Principle. This will help highlight the resulting difficulties in maintaining the code as well as the code structure's inflexibility.

We will be dealing with the scenario that we have software running on one system monitoring another system on its network. This code is responsible for capturing and logging events that are happening on the monitored system. The code will have to correctly interpret an event by referring to a set of data values that have already been gathered. To keep things simple, we will consider that these values were stored in a dictionary file and the software already has access to this dictionary. In this case, we will have to create a class based on these data values. The class is then responsible for interpreting the event.

Here's a demonstration of how we would code for such an application without following the Open and Closed Principle.

```
# openclosed_1.py

class Event:

def __init__(self, raw_data):

self.raw_data = raw_data

class UnknownEvent(Event):

"""A type of event that cannot be identified from its data."""
```

```
class LoginEvent(Event):
"""A event representing a user that has just entered the system."""
class LogoutEvent(Event):
"""An event representing a user that has just left the system."""
class SystemMonitor:
"""Identify events that occurred in the system."""
def __init__(self, event_data):
self.event_data = event_data
def identify_event(self):
if (
self.event_data["before"]["session"] == 0
and self.event_data["after"]["session"] == 1
):
return LoginEvent(self.event_data)
elif (
self.event_data["before"]["session"] == 1
and self.event_data["after"]["session"] == 0
):
return LogoutEvent(self.event_data)
return UnknownEvent(self.event_data)
```

Once the code is executed, we expect that the resulting output would be something like this:

```
>>> l1 = SystemMonitor({"before": {"session": 0}, "after": {"session": 1}})

>>> l1.identify_event().__class__.__name__

'LoginEvent'

>>> l2 = SystemMonitor({"before": {"session": 1}, "after": {"session": 0}})

>>> l2.identify_event().__class__.__name__

'LogoutEvent'

>>> l3 = SystemMonitor({"before": {"session": 1}, "after": {"session": 1}})

>>> l3.identify_event().__class__.__name__

'UnknownEvent'
```

If we analyze this output, we can come to the conclusion that the events which are identified by the code follow a strict hierarchy. Moreover, the code uses logic to define this hierarchy of events. To elaborate, when code is executed, it notices that there were no previous events happening on the monitored system. So, when it identifies the first event, it interprets it as the user logging on to the system, thus, a login session event. Similarly, if the code logs an event and after that, no events are being recorded, the logic of the code tells it that the last event it logged must be a logout session event. If the code encounters an event and can’t properly identify it, then it logs this event as ‘**unknown**’. Note that the event’s type is set to ‘unknown’ just as how an event’s type was set to ‘login’ or ‘logout’.

Now that we know how this code fundamentally works let’s discuss the flaws in this software code design. The most obvious problem is the logic that the code uses to classify each event's type as it is recorded. To elaborate, this logic is the core of a ‘monolithic’ method. In other words, this method adjusts its size according to the number of events the data values support. So, if we increase the event types that this software will cover, the method will also become larger in size, and this keeps on happening as we keep supporting more event types. Ultimately, we will reach a point where it will become unnecessarily large, which could be detrimental to the software application's performance. This means that we cannot freely extend this application's capabilities without having to worry about some side-effects.

In addition to this, this software design also supports modification, and we discussed that for a clean code design, we would want the infrastructure to be extensible while blocking any pathways for modification to the source code. This is not the case for this code infrastructure. This means that whenever we include or specify a new type of event for the application to support, we will have to go back to the method and make some changes to it to avoid errors. Thus, making maintaining this code a very difficult and arduous task.

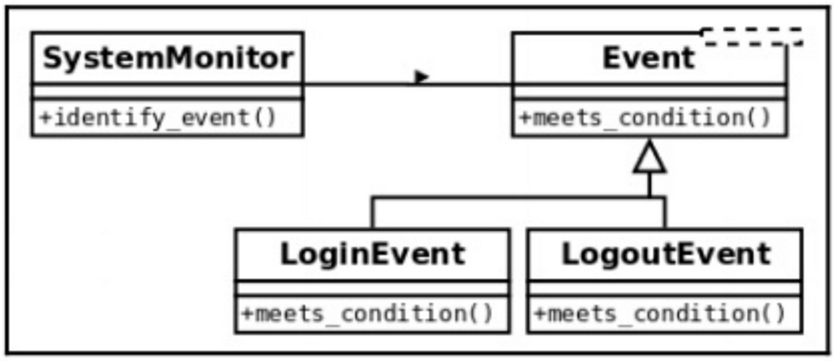
Ideally, we want this code to allows us to freely add support for new event types without impacting the core method, i.e., we only want to add new lines of code, not change the previous code every time we add a new line. In addition, we also don't want to go back and make modifications to the method every time we add a new event type (a new line of code).

Making the Code Extensible

If we compare the '**SystemMonitor**' example with this scenario and focus on the fundamental problem, we notice that the SystemMonitor class had an unnecessary interaction with the other 'concrete' classes. In other words, the '**SystemMonitor**' class would be aware of the changes that were being to any of the classes it is in touch with. Any changes made to these concrete classes would also require us to make appropriate modifications to the SystemMonitor class. A similar scenario is happening in this coding design, as well.

If we want this coding design to follow the Open and Closed Principle, we need to rework the infrastructure to lean towards abstraction.

One possible solution to achieve this abstraction would be to consider the behavior of the class as it attempts to interpret an event and then assign corresponding logic for classifying each event by a specific class. The software code design following this approach would look like this:



Now that we have the blueprint of the design we will be using for the code, we can begin to implement all of the necessary components to make the application correspond to the Open and Closed Principle. For each event type we want the application to support, we will be implementing a corresponding method. Each method will only have a single responsibility of checking the data values and determining whether they correspond to a specific supported event type or not. Similarly, since we are changing the way the code identifies event types, we also need to change the logic as well.

Once we make all of the necessary changes, we will end up with the following code (although it might differ from other implementations made independently by different people, but give or take, the fundamental design should be the same).

```
# openclosed_2.py

class Event:

def __init__(self, raw_data):

self.raw_data = raw_data

@staticmethod

def meets_condition(event_data: dict):

return False

class UnknownEvent(Event):

"""A type of event that cannot be identified from its data"""
```

```
class LoginEvent(Event):
@staticmethod
def meets_condition(event_data: dict):
return (
event_data["before"]["session"] == 0
and event_data["after"]["session"] == 1
)
class LogoutEvent(Event):
@staticmethod
def meets_condition(event_data: dict):
return (
event_data["before"]["session"] == 1
and event_data["after"]["session"] == 0
)
class SystemMonitor:
"""Identify events that occurred in the system."""
def __init__(self, event_data):
self.event_data = event_data
def identify_event(self):
for event_cls in Event.__subclasses__():
try:
if event_cls.meets_condition(self.event_data):
```

```
return event_cls(self.event_data)

except KeyError:

continue

return UnknownEvent(self.event_data)
```

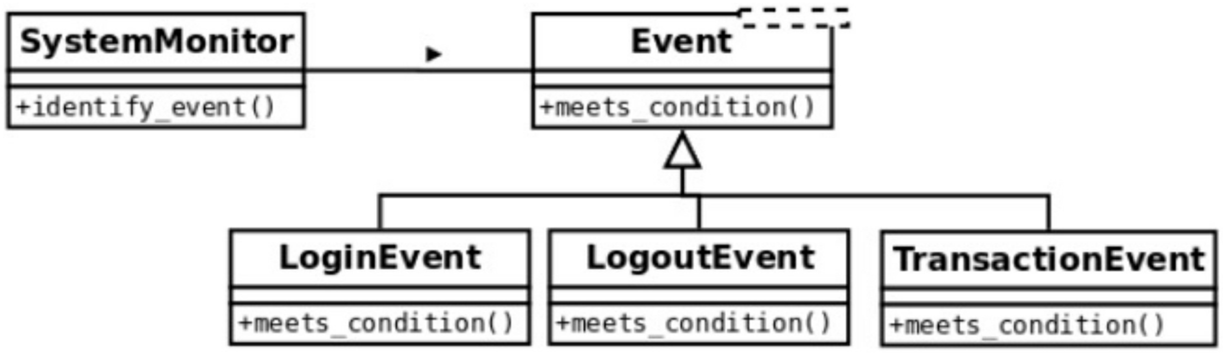
By implementing the following coding design, the software component’s interaction is now focused on abstractions instead of concrete classes. In other words, instead of using concrete classes that specify events, we use abstract classes that specify generic event types by following the conditional logic.

In terms of extensibility, code achieves this by identifying events by using the method ‘**__subclasses__()**.’ If we want to add new event types to the list of the application’s supported events, we simply need to generate another class corresponding to this event type. If we add multiple event types, we will also create multiple classes for each event type. One thing to note is that when we create a new class to accommodate a newly added event type, we are essentially creating a child class that will inherit the properties of the ‘**Event**’ class. However, this child class (also known as a subclass) will feature its own method (which will be the ‘**meets_condition()**’ method that all the other subclasses use for determining the event type) in order to follow the logic implemented in the application code.

Demonstrating the Extensibility

Now let’s talk a bit more in detail as to how the software code design we implemented in the previous section actually holds up to the claim of being easily extensible. To demonstrate this, we will first have to consider a scenario where we need to extend the application's functionality because of a new requirement. In addition, let’s say that instead of just simply identifying and logging events happening on the monitored system, we are now required also to add support for identifying transaction events as well.

To understand the structure of the code that we will demonstrate shortly, the following relationship diagram between the different classes might prove to be helpful.



From this relationship diagram, we can see the addition of a new class by the name of ‘**TransactionEvent**”. The following block of code demonstrates how to implement this new class to extend the functionality of the application.

```
# openclosed_3.py

class Event:

def __init__(self, raw_data):

self.raw_data = raw_data

@staticmethod

def meets_condition(event_data: dict):

return False

class UnknownEvent(Event):

"""A type of event that cannot be identified from its data"""

class LoginEvent(Event):

@staticmethod

def meets_condition(event_data: dict):

return (

event_data["before"]["session"] == 0
and event_data["after"]["session"] == 1
)
class LogoutEvent(Event):
@staticmethod
def meets_condition(event_data: dict):
return (
event_data["before"]["session"] == 1
and event_data["after"]["session"] == 0
)
class TransactionEvent(Event):
"""Represents a transaction that has just occurred on the system."""
@staticmethod
def meets_condition(event_data: dict):
return event_data["after"].get("transaction") is not None
class SystemMonitor:
"""Identify events that occurred in the system."""
def __init__(self, event_data):
self.event_data = event_data
def identify_event(self):
for event_cls in Event.__subclasses__():
try:
if event_cls.meets_condition(self.event_data):
```

```
return event_cls(self.event_data)

except KeyError:

continue

return UnknownEvent(self.event_data)
```

Extending the functionality of the application in this scenario does not affect the existing logic of the code nor does it require us to make additional modifications to the previous code as well. Upon executing the code, we can see that the code gives the same output as before and it also now gives us an additional output, i.e., the identification and logging of the new event type (Transaction event).

```
>>> l1 = SystemMonitor({"before": {"session": 0}, "after": {"session": 1}})

>>> l1.identify_event().__class__.__name__

'LoginEvent'

>>> l2 = SystemMonitor({"before": {"session": 1}, "after": {"session": 0}})

>>> l2.identify_event().__class__.__name__

'LogoutEvent'

>>> l3 = SystemMonitor({"before": {"session": 1}, "after": {"session": 1}})

>>> l3.identify_event().__class__.__name__

'UnknownEvent'

>>> l4 = SystemMonitor({"after": {"transaction": "Tx001"}})

>>> l4.identify_event().__class__.__name__

'TransactionEvent'
```

To elaborate, we can see that the method ‘**SystemMonitor.identify_event()**’ has not been modified and is implemented in the same way as it originally was when we included the support for the transaction events in this code.

This is a practical representation of the Open and Closed Principle which says that this method is closed for modifications.

Similarly, when we encountered a new requirement that the application needed to fulfill, we simply introduced a new subclass that inherits its properties from the superclass '**Event**.' In this way, we say that the 'Event' class is open to extending its functionality.

Liskov's Substitution Principle

According to Liskov's Substitution Principle, an 'object type' must have a collection of specific properties in order for it to have a reliable design.

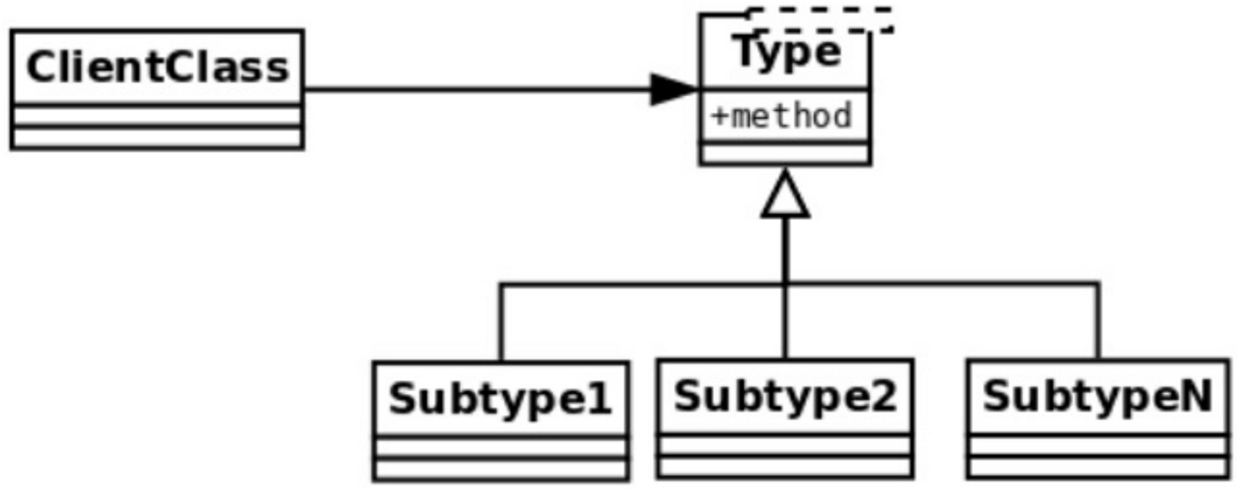
This principle works on the philosophy that if a person is interacting with an application, then every class being used in this application must be ready to allow the user to be able to use any of the corresponding subtypes. However, only allowing free access to the subtypes is not enough, the experience should be such that the users are not even aware that they are using a class's subtype. This incredibly lowers the chances of encountering any unexpected behavior while the application is running. To put it in simpler terms, the user is not aware of any changes happening in the hierarchy of classes, and basically, they are interacting with the classes in an isolated setting. However, implementing the principle does not actually involve a human user. Instead, the user is replaced with a client class that interacts with the other classes of the system.

Liskov's Substitution Principle also has technical and formal definitions for a better understanding of such concepts when provided with a technical perspective. Here's one such definition of Liskov's Substitution Principle:

"If S is a subtype of T, then objects of type T may be replaced by objects of type S, without breaking the program."

Now, we can conclude the fundamental working of this principle, i.e., the hierarchy implementation. If the class hierarchy is implemented in the application appropriately, then there should be no problem for the client class to interact with any subclass in the system. Moreover, the objects of these subclasses should have the ability to be easily substituted without causing any complications.

The software code design which follows Liskov's Substitution Principle needs to have the following relationship between the different classes.



This is also somewhat related to other clean code practices, such as creating efficient interfaces. As long as a class outlines a simple and small interface and the subclasses follow this interface, then the program will not break.

We will now explore how we can identify if a code is violating Liskov's Substitution Principle by using tools such as '**Mypy**' and '**Pylint**.'

Using Mypy to Identify Erroneous Datatypes in Method Signatures

In the first chapter of this book, we explored the advantages and benefits of using type annotations while coding. In this chapter, we will follow this advice of using type annotations and then configure and use a Python tool named 'Mypy' to identify and diagnose fundamental errors in the code and whether it complies with Liskov's Substitution Principle or not.

Let's say that we are using subclasses and superclass in our code. If any of the subclasses behave incorrectly or try to override a method in a way that does not comply with Liskov's Substitution Principle, the Python tool, '**Mypy,**' can immediately report this behavior by simply observing and analyzing the type annotations we used in the code.

```
class Event:

...

def meets_condition(self, event_data: dict) -> bool:

return False
```

```
class LoginEvent(Event):

def meets_condition(self, event_data: list) -> bool:

return bool(event_data)
```

Let’s use the Python tool ‘Mypy’ on this block of code and observe the output of the tool.

```
error: Argument 1 of "meets_condition" incompatible with supertype "Event"
```

As we can see from the result, the Mypy tool displays an error message that refers to the code's non-compliance with Liskov’s Substitution Principle.

To elaborate, the origin of the error is caused by the subclass using a different type for the parameter, namely, ‘**event_data**.’ This type is not the same as the type used by the superclass (or the class from which the subclass is derived from). Thus, the subclass and the superclass cannot function on the same terms causing problems. We must remember the fact that in Liskov’s Substitution Principle, any class that is in the lower spectrum of the hierarchy (the derived classes) must be able to properly work with the ‘**Event**’ class and ‘**LoginEvent**’ classes since they are ranked higher in the hierarchy of classes. Since the subclass is using a different type as opposed to the one defined in the superclass, we cannot expect that the code will not encounter an error when there is an attempt to substitute the objects. The very purpose of this principle is to allow objects to be interchanged without any errors and the error this code is encountering is essentially breaking the polymorphism of the hierarchy itself.

However, the error shown by the Mypy tool is not only limited to the use of a different type by a subclass. On the contrary, if any derived class were to change the object type in which it returned its output, the Mypy tool would raise this exact error message in this case as well. For instance, if the base class defines the return value type to be a Boolean value, then the derived classes should also return values in the Boolean type. If any derived class returns a value in any type other than a Boolean, then this error would be raised. Since the client classes of this application expect to be given a Boolean value, if the derived class gives them a value other than that, then the client class would not know how to deal with this data, thus breaking the program.

Using the ‘Pylint’ Tool to Identify Incompatible Signatures

Another error that breaks the code’s compliance with Liskov’s Substitution Principle is when each method in the code has a completely different signature. Although, on the surface, this problem is quite big and should not have happened in the first place, detecting such an anomaly is usually not possible by relying on the Python compiler alone in the early stages of the code development. These types of errors are only pointed out when the complete code is executed at the final runtime stage. However, we can use tools to diagnose such errors instead of being at the mercy of the Python compiler during code development. We can use ‘Mypy’ as well as ‘Pylint’ for such error diagnosing. However, it is recommended to use the ‘Pylint’ tool as opposed to ‘Mypy.’ This does not mean that ‘Mypy’ performs poorly or is not suited for this job. The only reason for this preference is that ‘Pylint’ offers a little more insight into the error message it displays, which can be very useful.

The following lines of code do not comply with Liskov’s Substitution Principle because they change the signatures of a method, making them incompatible.

```
# lsp_1.py

class LogoutEvent(Event):

def meets_condition(self, event_data: dict, override: bool) -> bool:

if override:

return True

...
```

Once we run the ‘Pylint’ tool on this block of code, it will return the following error message:

```
Parameters differ from overridden 'meets_condition' method (argumentsdiffer)
```

An important piece of advice here would be to never ignore or try to backline such errors. Instead, you should always improvise your code so that these errors disappear.

Other Liskov's Substitution Principle Violations

There are certain scenarios where a software code design does not comply with Liskov's Substitution Principle, and it is not entirely obvious such that even tools lose their purpose. In such cases, the only option is to inspect the code very vigilantly during the review stage of the development cycle.

One such scenario is when the expected interaction between base classes and derived classes are changed, or to put it simply, the contract between the components is changed. To elaborate, Liskov's Substitution Principle allows the client class to be able to interact and work with the derived classes in the same way as it would interact and deal with the base class. If this is to happen, the contractual behavior in the class hierarchy should be respected and upheld. If his contractual behavior is changed, then the class hierarchy also changes, and in this way, Liskov's Substitution Principle is violated. This is not easy to detect even when using tools like Pylit and Mypy, so the only thing we can do is review the code and see if any such scenario is happening.

In the third chapter of this book, we explored the contractual designing of code. As a reminder, a contractual code design means that there is an agreement between two software components, the client class, and the supplier class. This agreement is basically a set of rules, or in simpler terms, a set of conditions that need to be fulfilled in order to activate the method. So, the client class can define a set of preconditions to the method, and the supplier class will then confirm these preconditions and then give an output value, which the client class will then cross-check with its own set of postconditions.

So, any agreement defined between the base class and the client class should be upheld by the derived class. Deviating from the conditions set forth by this agreement will be a direct violation of the contractual code design and the core foundation of Liskov's Substitution Principle. When we say that the derived class needs to abide by the agreement by which the client class and base class are also bound, we basically mean that:

1. The derived class does not have the freedom to set any precondition that is harsher than the base class's ones.
2. The derived class does not have the freedom to set any

postcondition, which is duller than the ones defined by the base class.

To better understand this concept, let’s go back to the previous section and consider the event hierarchy example. However, this time, we will make slight changes to the scenario to properly explain the concept of the contractual agreement between client classes, derived classed and base classes, and their ability to set postconditions and preconditions.

In this example, we will consider that the client class has set a precondition for the method to follow. According to this precondition, the method needs first to validate that the parameter it is being given is a data dictionary, and this dictionary has two essential components, i.e., keys. Let’s say that there are two keys, namely ‘**before**’ and ‘**after**.’ The values of these keys are essentially in the form of ‘nested dictionaries.’ Due to this precondition, we now have the ability to perform even deeper encapsulation in the code. This is because of the fact that the client class does not need to use any error handling exception to deal with the scenario where the parameter is other than a dictionary. Instead, it can just call the preconditioning method, and all’s well.

After implementing some changes to slightly modify the code structure of the example discussed in the previous section, we will get the following block of code.

```
# lsp_2.py

class Event:

def __init__(self, raw_data):

self.raw_data = raw_data

@staticmethod

def meets_condition(event_data: dict):

return False

@staticmethod

def meets_condition_pre(event_data: dict):
```

```
"""Precondition of the contract of this interface.

Validate that the ``event_data`` parameter is properly formed.

"""

assert isinstance(event_data, dict), f"{event_data!r} is not a

dict"

for moment in ("before", "after"):

assert moment in event_data, f"{moment} not in {event_data}"

assert isinstance(event_data[moment], dict)
```

The following block of code determines the event's type and classifies it correctly by referring to the precondition set forth by the client class.

```
# lsp_2.py

class SystemMonitor:

"""Identify events that occurred in the system."""

def __init__(self, event_data):

self.event_data = event_data

def identify_event(self):

Event.meets_condition_pre(self.event_data)

event_cls = next(

(

event_cls

for event_cls in Event.__subclasses__()
if event_cls.meets_condition(self.event_data)

),

UnknownEvent,

)

return event_cls(self.event_data)
```

According to the contract between the classes, the dictionary is passed as a parameter only needs to have two keys, namely ‘**before**’ and ‘**after**.’ Moreover, the values of these corresponding keys need to be in the form of a dictionary as well. So, the derived classes must follow this contract as well and not go against it. Moreover, no derived class can specify an even stricter parameter. Otherwise, the contract between the client class and base class will be broken.

If we take a closer look at the block of code shown above, then we will see that it also uses a key by the name of ‘**transaction**.’ However, there is no restriction defined by the contract between the classes that specify the use of this key. So, the code uses this key if it is present within the dictionary. If the ‘**transaction**’ key is not present, then the code is unaffected and resumes its operation as usual.

```
# lsp_2.py

class TransactionEvent(Event):

"""Represents a transaction that has just occurred on the system."""

@staticmethod

def meets_condition(event_data: dict):

return event_data["after"].get("transaction") is not None
```

Now here’s come a complication. The first two methods originally being used by the code are the root cause of the problem. This because of the fact that these methods have a requirement similar to the precondition. In the precondition, the method requires the dictionary to have two keys, ‘**before**’ and ‘**after**.’ Similarly, these two methods also require a dictionary key named ‘**session**.’ This requirement is not included in the contractual agreement that was made initially between the base class and the client class, thus imposing a restriction after the contract has been made leads to the dissolving of the contract itself. Once the contract no longer exists, the client class no longer has access to the base class and the derived classes. Thus it raises an error flag, namely, ‘**KeyError**.’

Luckily, resolving this error is not that complicated, all we need to do is modify the method ‘**.get()**’ and introduce square brackets. The code now complies with Liskov’s Substitution Principle. Here’s the output of the block of code shown previously:

```
>>> l1 = SystemMonitor({"before": {"session": 0}, "after": {"session": 1}})

>>> l1.identify_event().__class__.__name__

'LoginEvent'

>>> l2 = SystemMonitor({"before": {"session": 1}, "after": {"session": 0}})

>>> l2.identify_event().__class__.__name__

'LogoutEvent'

>>> l3 = SystemMonitor({"before": {"session": 1}, "after": {"session": 1}})

>>> l3.identify_event().__class__.__name__

'UnknownEvent'

>>> l4 = SystemMonitor({"before": {}, "after": {"transaction": "Tx001"}})

>>> l4.identify_event().__class__.__name__

'TransactionEvent'
```

Thus, we cannot completely rely on tools like Mypy and Pylit to identify such types of errors, because at the end of the day, they are tools with imperfections and following principles like Liskov’s Substitution can have many different scenarios where it can be violated. So, tools cannot always properly determine if there even is a problem much less identify it.

The Interface Segregation Principle

This principle formally introduces something that we have already discussed and emphasized many times at this point, i.e., using small interfaces in the code.

Before we explore the Interface Segregation Principle's details, let’s first briefly discuss what we actually mean when we say the word ‘interface.’ In the context of Object-Oriented Programming Languages, an ‘Interface’ basically refers to the collection of methods that are revealed by an object. In other words, every message that the object is capable of fetching or even making sense of is collectively known as its interface. Moreover, a client class can request access to the interface of an object.

When working with Python, all of the methods belonging to a class collectively form its interface. This is due to the fact that the Python programming language follows a different principle known as ‘**duck typing**.’

The duck typing principle's core concept defined the very representation of an object in a programming environment. To elaborate, according to this principle, the main element which defines an object is one and only one thing, and this is the corresponding collection of methods the object has. In other words, it doesn’t matter as to what the name of the class is or what is its docstring or attributes, etc. The only thing that is of importance is the methods the class has since they are what defines the object, not anything else. So, the type of methods within a class is the final verdict on what the object will end up becoming. The reason why this principle is known as ‘**duck typing**’ because of the following idea, which is rather crude but still, gets the message across pretty decently.

“If something walks like a duck, speaks like a duck, then it surely must be a duck.”

Previously, the use of the duck typing principle was a popular approach towards defining interfaces in Python. However, the fundamental approach towards defining interfaces was changed when Python 3 launched. Python 3 brought with it a new idea of abstraction. According to this concept, the interfaces were defined on the basis of the abstract base classes being used in the code. So, the abstract superclasses (also known as base classes) were responsible for defining certain features, behaviors, and elements of the interface, which would then be implemented by the abstract subclasses. The primary use of this approach is for opening the possibility of overriding a bunch of critical functions in the code as well as giving the user the freedom to append additional functionalities to existing methods (like ‘**isinstance()**’).

To understand the Interface Segregation Principle, we must first fully understand how the Python programming language interprets the interfaces. Moreover, this understanding of Python and Interfaces will help us easily learn the fundamentals of the next principle (which is the Dependency Inversion Principle). Since we have already discussed the concept of interfaces in both object-oriented programming languages as well as in Python, let’s move on to discussing what does the implementation of the Interface Segregation Principle actually refers to.

When talking about the Interface Segregation Principle, what we actually mean is that if we are developing an application and during the coding process, we are required to define interfaces for the objects, then the code will be easier to maintain if they use many smaller interfaces instead of a few large interfaces. A ‘small’ interface means one that has a lesser number of methods as compared to other interfaces. So, if we do not follow the Interface Segregation Principle, then our code will feature a few interfaces, each having a larger number of methods. This makes the interface difficult to reuse. If we only want one specific attribute or specific property of the interface, we will either be forced to use the entire interface as it is or just leave it be. So, by dividing a big interface into smaller ones, we ensure that each interface has a very specific and narrow scope and responsibilities. If we want to reuse the code, we can pick up interfaces that have highly specific features for the classes making the reusability process resulting in a highly cohesive code design.

The Dependency Inversion Principle

The Dependency Inversion Principle's most appealing aspect is that it gives us a robust and solid structure for the software code design. According to the design proposed by this principle, we can improve the resistance of our code against errors and breakings by isolating the important parts and removing their dependencies from components that are prone to errors and can also break easily and have the potential to get out of hand.

The concept of removing these potential elements which hold the code back from running perfectly, the Dependency Inversion Principle, introduced the idea of 'Inverting the Dependencies.' To elaborate, we do not want the code to be forced to adapt to corresponding implementations. Instead, we want the implementations and other such elements to adapt to the code, thus, inverting the dependencies. This is usually done through the use of an Application Program Interface (also known as an API).

Similar to the example we have just seen, any abstraction in our code would usually depend on the concrete implementations (also known as details). This causes the code to become fragile and weak. If we were to apply the Dependency Inversion Principle, then instead of the abstractions depending on the details, the details would depend on the abstractions. Thus the code doesn't have to forcefully compromise its robustness in order to accommodate the fragility of the concrete implementations. Thus, the fragility of the concrete implementations remains isolated to themselves, and the robustness of the code becomes isolated from being affected by the implementations.

To understand the concept of dependency inversion even better, let's consider a simple example. We are dealing with two objects in our coding project at the moment, and the coding design we have chosen requires these two objects to cooperate with each other. In other words, if the two objects are represented by the letters '**A**' and '**B**' respectively, then we want object **A** to cooperate with an instance of object **B**. However, we soon come to know that the module we are using does not have any direct authority over the behavior of object **B**. So, if there were a scenario where the code we have written so far is greatly impacted by object **B**, then there is a high chance that such a change will break the entire program itself. This means that our code depends on a very fragile component, thus making the rest of the code fragile. So, the solution is to reverse the dependency, i.e., instead of having object **A** depend and adapt to object **B,** we make object **B** depend on object **A**. To perform this dependency inversion, we simply introduce an interface. Once we have the interface in the arena, we simply need to force the code to depend on the interface instead of object **B**'s concrete implementation. In this way, object **B** will also be forced to adapt to the interface in the end as well. Thus, the dependency has been inversed.

Chapter 5: Implementing the Decorators to Improve the Quality of the Code

In this chapter, we will discuss a very useful feature in Python that can help improve the quality of the code in which it is implemented by manifold. Decorators have many uses in coding, and not only can they help us avoid sticky situations in code development, but they can also help us implement a better code design for our software, among many other advantages. Before we can even talk about such topics, we must first get acquainted with the concept of 'decorators.' The rest comes after only if we understand the fundamental idea revolving around decorators.

In this chapter, we will explore what decorators are and then discuss the various methods and scenarios where they can be implemented during code development.

Once we become familiar with decorators, how they work and how they can be used, we will then explore the ideas and concepts that were outlined in the previous chapters but this time, see them in a new light by observing how much code can be improved by using decorators.

In this chapter, you will learn how:

- Decorators actually work in the Python programming language.
- To implement decorators through functions and classes.
- To avoid common mistakes in decorator implementation.
- To review some exemplary implementations of decorators in coding.

Decorators in Python

Decorators have been used in the python language since PEP-318, which was introduced long ago. Basically, they are tools that allow the programmer to simplify the definition of functions and methods after their original definition is to be modified.

Before decorators, the "classmethod" and "staticmethod" functions were used whenever the programmer needed to provide extra functionality to a method by modifying it. But these two functions needed another extra line along with the original definition to modify it.

In this case, the function the programmer needs to transform is called the "modifier" function and also requires another step of reassignment to the original name of the function.

To understand this concept, consider the example where a function "original" is taken, and another function "modifier" is applied onto it, which transforms it to give it extra functionality, then the code would be written as given below:

```
def original(...):

   ...

original = modifier(original)
```

It can be seen that the function after it was modified was reassigned to the same name that the original function had. This method was abandoned, and decorators were invented because programmers often made mistakes and found it confusing and tiring. Some common errors were something like forgetting to reassign the function or reassigning it farther down the program instead of immediately after the function definition. Decorators are syntax support that simplifies the previous code to look like this:

```
@modifier

def original(...):

...
```

To elaborate, the decorator calls the function following after it as its own first parameter and returns the final result.

In the previous example, if we were to use the Python terminology, then the decorator would be the modifier, and the original is the function that was modified. Another name for the decorated function is "wrapped object."

Decorators were originally intended only for functions and methods, but their functionality extends to any kind of object. As such, it is possible to explore and elaborate on the usage of decorator syntax on methods, functions, generators, and classes.

The decorator is named as such because of its uses, such as modifying, extending, or working on top of any wrapped object, but the programmer should know not to mix it up with the decorator design pattern.

Working with Functions to Implement Decorators

Out of all the objects, functions are the easiest and simplest choice to demonstrate decorators' use. Using the decorators on functions allows the programmer access to different types of logic, such as:

- Validation of parameters.
- Checking of preconditions.
- Alteration of behavior.
- Signature modification.
- Cache results.

To show the functioning of the decorator, we will take an example where a decorator of a basic level is used to create a “retry” mechanism where a specific domain-level exception is controlled and tried over again and again.

```
from functools import wraps

from unittest import TestCase, main, mock


from log import logger



class ControlledException(Exception):

    """A generic exception on the program's domain."""



def retry(operation):

    @wraps(operation)

    def wrapped(*args, **kwargs):

        last_raised = None
        RETRIES_LIMIT = 3
        for _ in range(RETRIES_LIMIT):
            try:
                return operation(*args, **kwargs)
            except ControlledException as e:
                logger.info("retrying %s", operation.__qualname__)
                last_raised = e
        raise last_raised

    return wrapped


class OperationObject:
    """A helper object to test the decorator."""

    def __init__(self):
        self._times_called: int = 0

    def run(self) -> int:
        """Base operation for a particular action"""
        self._times_called += 1
        return self._times_called

    def __str__(self):
        return f"{self.__class__.__name__}()"

    __repr__ = __str__


class RunWithFailure:
    def __init__(
        self,
        task: "OperationObject",
        fail_n_times: int = 0,
        exception_cls=ControlledException,
    ):
        self._task = task
        self._fail_n_times = fail_n_times
        self._times_failed = 0
        self._exception_cls = exception_cls

    def run(self):
        called = self._task.run()
        if self._times_failed < self._fail_n_times:
            self._times_failed += 1
            raise self._exception_cls(f"{self._task!s} failed!")
        return called
```

```
@retry
def run_operation(task):
    """Run a particular task, simulating some failures on its execution."""
    return task.run()


class RetryDecoratorTest(TestCase):
    def setUp(self):
        self.info = mock.patch("log.logger.info").start()

    def tearDown(self):
        self.info.stop()

    def test_fail_less_than_retry_limit(self):
        """Retry = 3, fail = 2, should work"""
        task = OperationObject()
        failing_task = RunWithFailure(task, fail_n_times=2)
        times_run = run_operation(failing_task)

        self.assertEqual(times_run, 3)
        self.assertEqual(task._times_called, 3)

    def test_fail_equal_retry_limit(self):
        """Retry = fail = 3, will fail"""
        task = OperationObject()

        failing_task = RunWithFailure(task, fail_n_times=3)

        with self.assertRaises(ControlledException):

            run_operation(failing_task)


    def test_no_failures(self):

        task = OperationObject()

        failing_task = RunWithFailure(task, fail_n_times=0)

        times_run = run_operation(failing_task)


        self.assertEqual(times_run, 1)

        self.assertEqual(task._times_called, 1)



if __name__ == "__main__":

    main()
```

As can be seen in the example, the “@wraps” can be ignored for now, as it will be discussed later on. In the for loop, the symbol “_” represents a number assigned to a variable that is ignored here because it has no use.

No parameters need to be applied to the retry decorator and so, it can be used with any function.

```
@retry def run_operation(task):

"""Run a particular task, simulating some failures on its execution."""

return task.run()
```

The @retry preceding the run_operation function is the simple way of the decorator to execute the code:

```
run_operation = retry(run_operation).
```

Consequently, the example shows a decorator creating a common retry operation that will call the decorated code numerous times as long as some specific conditions are met. In this case, the certain condition was the exception related to timeouts.

Working with Decorator Classes

The object class is decorated in the same manner as the function is where the only difference is that the fact that a class is being decorated is taken into consideration when coding for the decorator.

There is a common opinion among programmers that decorating classes is a complex process, and it might actually affect the readability in a bad way because even though the attributes and methods are declared to the intended class, the decorator makes such alterations and result in an entirely different class.

Although these concerns are true, it is not entirely the case. Such convolutions can only occur if this method is not used appropriately. The technique of decorating classes is similar to that of decorating functions because ultimately, classes are also Python objects as functions. Putting aside the issues regarding decorating classes, let us discuss the advantages of this method:

- By decorating classes, we can mainly reuse code and avoid repetitions (DRY principle). In the case where different classes have to be set so that they all follow a certain set of rules, they are all applied with a single decorator with all the checks written in its code once.
- The small classes which are made with simple coding can be modified as seen fit later by the use of decorators.
- If a certain class is needed to be transformed and maintained, instead of using the convoluted and difficult technique such as using metaclasses, it is preferable to use a decorator.

The simple example shown below is for the purpose of showcasing the applications of a decorator in the case of a class. The given code can have many other possible solutions as well, where each has its own advantages and disadvantages. Our approach to the solution will be via decorator so that their usefulness can be explored.

The monitoring platform requires the event systems to be recalled, after which the event data is modified and sent to the external system. This process is not as simple as it sounds, as each event has its own way of choosing which data it will send.

Looking at some of the events, the event related to “log in” will choose the data carefully. It contains sensitive information such as credentials that the user wants to keep private. The “timestamp” event will require certain modifications as it must be shown in a particular format. The method to fulfill the requirements of each and every event is quite simple as all we need is a class that maps to each specific event and has sufficient data to serialize it:

```
import unittest
from datetime import datetime


class LoginEventSerializer:
    def __init__(self, event):
        self.event = event

    def serialize(self) -> dict:
        return {
            "username": self.event.username,
            "password": "**redacted**",
            "ip": self.event.ip,
            "timestamp": self.event.timestamp.strftime("%Y-%m-%d
%H:%M"),
        }
```

```
class LoginEvent:
    SERIALIZER = LoginEventSerializer

    def __init__(self, username, password, ip, timestamp):
        self.username = username
        self.password = password
        self.ip = ip
        self.timestamp = timestamp

    def serialize(self) -> dict:
        return self.SERIALIZER(self).serialize()


class TestLoginEventSerialized(unittest.TestCase):
    def test_serializetion(self):
        event = LoginEvent(
            "username", "password", "127.0.0.1", datetime(2016, 7, 20, 15, 45)
        )
        expected = {
            "username": "username",
            "password": "**redacted**",
            "ip": "127.0.0.1",
            "timestamp": "2016-07-20 15:45",
        }
        self.assertEqual(event.serialize(), expected)


if __name__ == "__main__":
    unittest.main()
```

The code contains a class that maps directly to the login and timestamp event. It has the required logic for these events, that is, hide the password field for the login and correctly format the timestamp.

In the beginning, this technique seems easy and simple, but as time goes on and the system is further extended, many issues arise, which cause a lot of headaches:

- **Numerous classes:** The extending system will have more and more events, which in return leads to more serialization classes because they are mapped directly one to one with each event.
- **Inflexible solution:** If a part of an event is needed in another place, such as hiding the password field in an event other than login, we need to extract that part into a function. But the repetitive calling from multiple classes will result in less reuse of code.
- **Boilerplate:** The method “serialize()” needs to be input in all the event classes, so they call the same code. The alternative method of extracting this into other classes and creating a mixin might seem like a better choice. It isn’t an appropriate use of inheritance.

Another good solution is creating an object that takes a set of filters and an event instance and serializes that event by applying the specific filters related to it. In this case, only the functions that are involved in transformation are defined, and the serializer simply contains all of these functions.

After such an object is constructed, the class is decorated and modified to contain the serialize() method. This is used to call the serialization objects within itself:

```
import unittest
from datetime import datetime


def hide_field(field) -> str:
    return "**redacted**"


def format_time(field_timestamp: datetime) -> str:
    return field_timestamp.strftime("%Y-%m-%d %H:%M")


def show_original(event_field):
    return event_field


class EventSerializer:
    """Apply the transformations to an Event object based on its properties and
    the definition of the function to apply to each field.
    """

    def __init__(self, serialization_fields: dict) -> None:
        """Created with a mapping of fields to functions.

        Example::
```

```
        >>> serialization_fields = {
        ...     "username": str.upper,
        ...     "name": str.title,
        ... }

        Means that then this object is called with::

        >>> from types import SimpleNamespace
        >>> event = SimpleNamespace(username="usr", name="name")
        >>> result = EventSerializer(serialization_fields).serialize(event)

        Will return a dictionary where::

        >>> result == {
        ...     "username": event.username.upper(),
        ...     "name": event.name.title(),
        ... }
        True

        """
        self.serialization_fields = serialization_fields

    def serialize(self, event) -> dict:
        """Get all the attributes from ``event``, apply the transformations to
        each attribute, and place it in a dictionary to be returned.
        """
        return {
            field: transformation(getattr(event, field))
            for field, transformation in self.serialization_fields.items()
        }


class Serialization:
    """A class decorator created with transformation functions to be applied
    over the fields of the class instance.
    """

    def __init__(self, **transformations):
        """The ``transformations`` dictionary contains the definition of how to
        map the attributes of the instance of the class, at serialization time.
        """
        self.serializer = EventSerializer(transformations)

    def __call__(self, event_class):
        """Called when being applied to ``event_class``, will replace the
        ``serialize`` method of this one by a new version that uses the
        serializer instance.
        """

        def serialize_method(event_instance):
            return self.serializer.serialize(event_instance)

        event_class.serialize = serialize_method
        return event_class


@Serialization(
    username=str.lower,
    password=hide_field,
    ip=show_original,
    timestamp=format_time,
)
class LoginEvent:
    def __init__(self, username, password, ip, timestamp):
        self.username = username
        self.password = password
        self.ip = ip
        self.timestamp = timestamp


class TestLoginEventSerialized(unittest.TestCase):
    def test_serialization(self):

        event = LoginEvent(

            "UserName", "password", "127.0.0.1", datetime(2016, 7, 20, 15,
45)

        )

        expected = {

            "username": "username",

            "password": "**redacted**",

            "ip": "127.0.0.1",

            "timestamp": "2016-07-20 15:45",

        }

        self.assertEqual(event.serialize(), expected)



if __name__ == "__main__":

    unittest.main()
```

It can be easily understood from the code above that the user can grasp the treatment of each field with the help of the decorator and the hassle to refer to the code of another class is not required. Going further into this plus side's details, it is clear upon the overview of arguments how the username, password, and timestamp have been treated. The username and IP address has not been modified, the password is kept hidden and the timestamp is formatted accordingly.

Examining the code of the class, it is clear that the serialize() method does not need definition and extension from the mixin it is implemented from. The decorator will do the work of adding it to the class. This is the only part where the class decorator can be reasonably created because it is possible that the serialization object was made to be a class attribute of the “Login” event. What makes this impossible is the fact that the class is being modified with the addition of a new method.

Another class decorator can be possible as well where only the attributes of the class are defined, and the logic of init method is implemented, but the scope of the example is limited so it cannot be utilized here. This technique is mostly used by libraries like the “attrs” (ATTRS 01) and the standard library of PEP-557.

To use this class decorator in the previous example compactly, we will take it from the PEP-557 and use it in Python 3.7+:

```
import sys

import unittest

from datetime import datetime


from decorator_class_2 import (

    Serialization,

    format_time,

    hide_field,

    show_original,

)


try:

    from dataclasses import dataclass

except ImportError:


    def dataclass(cls):

        return cls


@Serialization(
    username=show_original,
    password=hide_field,
    ip=show_original,
    timestamp=format_time,
)
@dataclass
class LoginEvent:
    username: str
    password: str
    ip: str
    timestamp: datetime


class TestLoginEventSerialized(unittest.TestCase):
    @unittest.skipIf(
        sys.version_info[:3] < (3, 7, 0), reason="Requires Python 3.7+ to run"
    )
    def test_serializetion(self):
        event = LoginEvent(
            "username", "password", "127.0.0.1", datetime(2016, 7, 20, 15, 45)
        )
        expected = {
            "username": "username",
            "password": "**redacted**",
            "ip": "127.0.0.1",
            "timestamp": "2016-07-20 15:45",
        }
        self.assertEqual(event.serialize(), expected)


if __name__ == "__main__":
    unittest.main()
```

The Different Types of Detectors

With the knowledge of how the @syntax for decorators actually works, it is now evident that decorators' use is not restricted to just functions, classes, or methods. Its functionality extends further by being able to define generators, coroutines, and previously decorated objects as well, in which case the decorators are stacked on top of each other.

The decorators used in the previous example were shown to be chained. To achieve this, the class was first defined and @dataclass was applied on top of it. This transformed it into a data class. Now the class had an additional functionality of being able to act as an attribute container. The decorator @serialization stacks on top of it and applies the logic to the existing class, which gives a new class where a new @serialize method is added.

Decorators are used on generators as well when they are to be used as coroutines. Mainly, a new generator is created and before any data is sent to it, the function “next()” is called upon it to advance it to the “next” yield statement. And so, a decorator can be used in this case where the generator is assigned to it as the parameter, then the decorator calls “next()” to it, and the generator is returned afterward.

Providing Parameters to Decorators

In the Python programming language, decorators are already powerful enough with their capabilities, but this can be enhanced further if parameters are passed onto them, resulting in their logic to be even more abstracted.

There are quite a few ways by which arguments are passed onto decorators, but some common ones are given below:

- One way is to create the decorators in such a way that they take on the form of nested functions with a whole new level of indirection. This method is intended to make everything contained in the decorator to fall one level deeper.
- The second way is to use a class for the relevant decorator.

In terms of readability, the second method is better than the first due to the ease with which the user can understand objects rather than three or more nested functions closely working. However, to understand both methods completely and to know where to implement which method, it is best to examine both of them.

Understanding the Implementation of Decorators in Terms of “Nested Functions”

To look at it generally, the decorator creates a higher-order function which is responsible for returning another function. The internal function that is being called is defined in the body of the decorator.

The next step is to pass on parameters to the decorator, prior to which we need to have another level of indirection. The first function which is the higher-order function will take the passed-on parameters and within this function, a new function i.e., the decorator is defined. This decorator will contain another level of function within itself which is the function that will be returned at the end of the decoration process. At this point, we have three levels of nested functions.

For further clarity of the above description, an example will be provided below.

The first example of decorators that we encountered was implementing the retry functionality over some functions. Even though it seems like a good choice, but the problem poses is that the implementation did not allow the specifications of the number of retries whereas the decorator had a fixed number inside of it.

Now in case, the user wants to assign a specific number of retries for each instance and also maybe even be allowed the addition of a default value to this parameter, then he will need another level of nested functions. The levels of functions will first be for the parameters and then for the decorator.

The form we are left with is:

```
@retry(arg1, arg2,... )
```

As the @syntax applies the result of the executing code to the decorated object, the return of this code will be a decorator. The code when expressed in terms of function, will look like this:

```
<original_function> = retry(arg1, arg2, ....)(<original_function>)
```

In addition to controlling the number of retries, the user is also allowed to add exceptions that need to be controlled as well. Along with the new requirements, the code is transformed into the following:

```
RETRIES_LIMIT = 3

def with_retry(retries_limit=RETRIES_LIMIT, allowed_exceptions=None):

allowed_exceptions = allowed_exceptions or (ControlledException,)

def retry(operation):

@wraps(operation)

def wrapped(*args, **kwargs):

last_raised = None

for _ in range(retries_limit):

try:
```

```
return operation(*args, **kwargs)

except allowed_exceptions as e:

logger.info("retrying %s due to %s", operation, e)

last_raised = e

raise last_raised

return wrapped

return retry
```

To understand the application of this decorator to functions, and showcase how it accepts different options, examine the example below:

```
from functools import wraps


from decorator_function_1 import ControlledException

from log import logger



RETRIES_LIMIT = 3



def with_retry(retries_limit=RETRIES_LIMIT, allowed_exceptions=None):

    allowed_exceptions = allowed_exceptions or (ControlledException,)


    def retry(operation):

        @wraps(operation)

        def wrapped(*args, **kwargs):
            last_raised = None
            for _ in range(retries_limit):
                try:
                    return operation(*args, **kwargs)
                except allowed_exceptions as e:
                    logger.warning(
                        "retrying %s due to %s", operation.__qualname__, e
                    )
                    last_raised = e
            raise last_raised

        return wrapped

    return retry


@with_retry()
def run_operation(task):
    return task.run()


@with_retry(retries_limit=5)
def run_with_custom_retries_limit(task):
    return task.run()
```

```
@with_retry(allowed_exceptions=(AttributeError,))
def run_with_custom_exceptions(task):
    return task.run()


@with_retry(
    retries_limit=4, allowed_exceptions=(ZeroDivisionError, AttributeError)
)
def run_with_custom_parameters(task):
    return task.run()
```

Implementing the Objects of Decorators

The example that we used and studied previously had three levels of nested function in it. The first level is a function that takes up the decorator’s parameter while the functions contained within it are just closure that uses these parameters according to the decorator’s logic.

To implement this clearly, a class is used to define the decorator. So, we pass the arguments or the parameters in the method “__init__” followed by implementation of decorator logic on the method “__call__” which is a magic method.

The code for this type of decorator is:

```
from functools import wraps


from decorator_function_1 import ControlledException
from log import logger


RETRIES_LIMIT = 3

```

```
class WithRetry:
    def __init__(self, retries_limit=RETRIES_LIMIT, allowed_exceptions=None):
        self.retries_limit = retries_limit
        self.allowed_exceptions = allowed_exceptions or (ControlledException,)

    def __call__(self, operation):
        @wraps(operation)
        def wrapped(*args, **kwargs):
            last_raised = None

            for _ in range(self.retries_limit):
                try:
                    return operation(*args, **kwargs)
                except self.allowed_exceptions as e:
                    logger.info(
                        "retrying %s due to %s", operation.__qualname__, e
                    )
                    last_raised = e
            raise last_raised

        return wrapped
```

```
@WithRetry()
def run_operation(task):
    return task.run()


@WithRetry(retries_limit=5)
def run_with_custom_retries_limit(task):
    return task.run()


@WithRetry(allowed_exceptions=(AttributeError,))
def run_with_custom_exceptions(task):
    return task.run()


@WithRetry(
    retries_limit=4, allowed_exceptions=(ZeroDivisionError, AttributeError)
)
def run_with_custom_parameters(task):
    return task.run()
```

Applying this decorator to a function, the code will look as follows:

```
@WithRetry(retries_limit=5)
def run_with_custom_retries_limit(task):
return task.run()
```

The effect of Python syntax in the example is an important point to be noticed. First, the creation of the object required the @operation where the object is created along with the parameters that were passed on to it. As per defined in the init method, the newly created object will be initialized with the passed-on parameters. After invoking the @operation, the object wraps the function "run_with_custom_retries_limit" which deals with passing this on to the magic method "call".

Finally, in the "call" magic method, the logic of the decorator is defined where the original function is wrapped and a new one is created which contains the desired logic.

Some Advantageous Scenarios for Decorators

There are certain situations where decorators make a good choice of usage and it must be known what those common patterns and situations are.

Out of all the various uses decorators have, some of the most common and relevant ones are explained below:

- **Transformation of parameters:** In this case, the decorated function has its signature changed such that it gives a nicer API while also including the details on the treatment of parameters and their transformation.
- **Tracing the code:** The execution of a function is logged along with its parameters.
- **Validation:** The parameters passed onto the function are validated.
- **Implementation of retry operations**
- **Class simplification:** The code of the classes is simplified only by moving all the repetitive logic (and non-repetitive as well), into decorators.

We will keep our discussion about these different uses for decorators limited to only the first one and explore it in a bit more detail.

Changing the Parameters (or Transforming Them)

In the previous sections, we have already discussed the potential use of decorators for the purpose of parameter validation. We can even use decorators to act on our behalf and implement a set of preconditions and postconditions as well. From these characteristics of how a decorator can be used as a transformation agent, we now come to our main topic of discussion, i.e., to use decorators to change or simply transform the parameters. This means that whenever we are working with parameters in our code, we can implement decorators and gain the ability to manipulate and transform the corresponding parameters.

One might wonder as to what’s even the point of manipulating the parameters during the coding process. One of the most common scenarios where we find ourselves annoyed by parameters is when we are stuck in a situation where we have to generate objects similar to each other and even perform similar transformations. Doing this becomes repetitive and very annoying. We can simply perform these tasks through simple abstractions and to do this, we only need to implement decorators.

Implementing Decorators Effectively and Avoiding Implementation Mistakes

Although decorators offer great functionality and features during the coding and we can benefit greatly from using decorators in our code, however, many people would face some serious issues when trying to implement decorators properly. In coding, decorators are a component that needs to be handled with great care, implementing them in the wrong way, or making silly mistakes when using functions and classes with decorators can lead to serious problems. In this section, we will discuss a bunch of issues that an average person would usually face when implementing decorators in their code.

Retaining the Information of the Initial ‘Wrapped Object’

The most common issue when implementing decorators is that users encounter the problem of losing data about the properties and even attributes of the original functions alongside which we are using the decorators. Once the decorators are implemented, the user loses information regarding the function’s properties and attributes (which were originally present in the code before the implementation of the decorator). Due to this loss of data, we can encounter unnecessary complications and problems when executing our code which could have been otherwise avoided if we carefully implemented the decorators.

Let’s demonstrate this by implementing a decorator that is responsible for logging events just before the function is about to execute so that we can simulate the problem we have mentioned above.

```
> functools.wraps
"""
from log import logger


def trace_decorator(function):
    def wrapped(*args, **kwargs):
        logger.info("running %s", function.__qualname__)
        return function(*args, **kwargs)

    return wrapped


@trace_decorator
def process_account(account_id):
    """Process an account by Id."""
    logger.info("processing account %s", account_id)
    ...
```

Until now, we haven't implemented the decorator yet, we have only seen the function along with its properties and attributes. Now, if we were to add the decorator to this function carelessly, then the definition of the function shown above will actually be changed as shown below.

```
> functools.wraps
"""
from functools import wraps
from log import logger

def trace_decorator(function):
    """Log when a function is being called."""

    @wraps(function)
    def wrapped(*args, **kwargs):
        logger.info("running %s", function.__qualname__)
        return function(*args, **kwargs)

    return wrapped

@trace_decorator
def process_account(account_id):
    """Process an account by Id."""
    logger.info("processing account %s", account_id)
    ...

def decorator(original_function):
    @wraps(original_function)
    def decorated_function(*args, **kwargs):
        # modifications done by the decorator ...
        return original_function(*args, **kwargs)
    return decorated_function
```

On the surface, we might not be able to notice that there are any impacting changes or modifications made to any part of the function’s definition but the fact remains that the decorator is not supposed to make any changes in the first place. If we look at the function’s definition after we implement the decorator, we find out that the decorator ends up renaming the function to ‘**docstring**’ and change some other of its properties as well.

Let’s use the ‘**help**’ command to know more about the function we are dealing with.

```
>>> help(process_account)

Help on function wrapped in module decorator_wraps_1:

wrapped(*args, **kwargs)
```

Now let’s see how this function is actually called.

```
>>> process_account.__qualname__

'trace_decorator.<locals>.wrapped'
```

From this information, we come to the conclusion that the original function being used in the code has been transformed into another function named ‘**wrapped()**’ after the implementation of the decorator. So, when we query to see the function's properties, the data shown corresponds to the ‘wrapped()’ function and not the one we were originally using.

If we implement this decorator across multiple functions (and each function is different from the other), then all of them will be changed to ‘**wrapped**’, thus causing a catastrophic problem for us to later deal with.

Chapter 6: Using Python Generators When Coding

In this chapter, we will focus on a feature that makes Python a somewhat weird programming language compared to the traditional programming languages of the past. We will discuss the use of the ‘Generators’ feature in Python and explore why were they even introduced in the first place and the advantages of using them.

Iteration is one of Python's key features and we see iterative patterns frequently used as well because of how well it can be implemented in this programming language. Furthermore, we will also talk about how other features in Python, such as asynchronous programming and coroutines are somewhat dependent on generators in Python.

One thing to note before moving on to the main discussion is that to be able to recreate the demonstrations shown in this chapter, the reader will need to have Python 3.6 installed on their system. The Operating System doesn’t matter, it will work on macOS, Linux, and Windows without any problems.

The Purpose of Generators

Generators were introduced in Python version PEP-255. This was the first-time iteration was brought to this programming language while keeping significant performance boosts, such as better code execution performance and less memory consumption.

The fundamental concept of a generator is the use of an object which has the property of being iterative. So, once this object is used and enters the iteration cycle, it will create instances of the elements it contains per cycle. However, the object does not produce a copy of every single element in one cycle. Instead, only one element is chosen and an instance of it is created in one iteration. Traditionally, if we were to create instances of different elements through iteration, we would have to create entire lists of such elements and then introduce functions to create their instances and then loop them. This would take up a lot of memory but using generators allows us to skip lists, because the object has all of the information it needs. All we have to do is create the generator, then the corresponding object will be created and it will then begin producing instances of its elements as it completes one iteration cycle.

Understanding the Use of Generators

Now that we know for what purpose we use generators in coding, let's now explore how we can actually implement them in our code.

The best way to do that would be to demonstrate an example. To do this, let's first build a scenario detailing how we would build software and write the appropriate code. Let's say that we are given a huge list containing data of records and we are required to process all of this data along with showing appropriate metrics and indicators for each record in the list. In simpler words, we have a huge list of purchase data and we are asked to process it such that we return three types of data, i.e., lowest sale price, highest sale price, and the average sale price. These three are the metrics according to which we need to process the entire dataset.

Such an example has the potential of easily becoming too complex and difficult to understand, thus we will keep things simple by considering that the CSV fields provided to us consist of a total of 2 fields. Moreover, these fields have a specific format which has been shown below:

```
<purchase_date>, <price>

...
```

As we discussed earlier, a generator works by having an object with all of the elements that need to be generated contained within this object (that's why it is called a 'generator' since it constantly creates the elements contained within it). We now need to create an object and put all of the purchase data inside this object, in this way, we now have the required metrics for the data we are required to process. Generally, if we were to use functions like '**mix()**' and '**max()**', we can have the object produce many of the data values it has carried over from the purchase records file. However, if we do produce the elements in this way, we would essentially create an additional copy of these purchase data values. So, instead of dealing with duplicate data and ruining the final data analysis, we will stick to using a custom object we created that will produce one element from the purchase data per iteration and avoid having to deal with duplicate data instances.

The lines of code that we will be using to implement the object (we previously created) and have it cough up the corresponding data instances are not that sophisticated. On the contrary, the design itself is very simple and straight forward. The entire code is based on a single pair of software components, to be more precise, an object and a method. The method is responsible for calculating the required metrics while observing the data and the result is then stored in the object which in turn, updates the values of the metrics in the list which contains this data. Here’s the implementation of this generator, program, and the code it contains.

```
"""

from _generate_data import PURCHASES_FILE, create_purchases_file

from log import logger



class PurchasesStats:

    def __init__(self, purchases):

        self.purchases = iter(purchases)

        self.min_price: float = None

        self.max_price: float = None

        self._total_purchases_price: float = 0.0

        self._total_purchases = 0

        self._initialize()


    def _initialize(self):

        try:

            first_value = next(self.purchases)

        except StopIteration:

            raise ValueError("no values provided")
        self.min_price = self.max_price = first_value
        self._update_avg(first_value)

    def process(self):
        for purchase_value in self.purchases:
            self._update_min(purchase_value)
            self._update_max(purchase_value)
            self._update_avg(purchase_value)
        return self

    def _update_min(self, new_value: float):
        if new_value < self.min_price:
            self.min_price = new_value

    def _update_max(self, new_value: float):
        if new_value > self.max_price:
            self.max_price = new_value

    @property
    def avg_price(self):
        return self._total_purchases_price / self._total_purchases

    def _update_avg(self, new_value: float):
        self._total_purchases_price += new_value
        self._total_purchases += 1

    def __str__(self):
        return (
            f"{self.__class__.__name__}({self.min_price}, "
            f"{self.max_price}, {self.avg_price})"
        )


def _load_purchases(filename):
    purchases = []
    with open(filename) as f:
        for line in f:
            *_, price_raw = line.partition(",")
            purchases.append(float(price_raw))

    return purchases


def load_purchases(filename):
    with open(filename) as f:
        for line in f:
            *_, price_raw = line.partition(",")
            yield float(price_raw)
```

```
def main():

    create_purchases_file(PURCHASES_FILE)

    purchases = load_purchases(PURCHASES_FILE)

    stats = PurchasesStats(purchases).process()

    logger.info("Results: %s", stats)



if __name__ == "__main__":

    main()
```

The object used in this block of code collects all of the purchase data values that have been totaled. It then processes these values into the metrics we have specified (lowest sales value, highest sales value, and average sales value). With this, we are done with the first requirement, i.e., process the data. Now we have to figure out a way to take this processed data and put it in some container which the object can read and then update the values in its data list accordingly. We will be using a function, which has been implemented in the following lines of code.

```
def _load_purchases(filename):

purchases = []

with open(filename) as f:

for line in f:

*_, price_raw = line.partition(",")

purchases.append(float(price_raw))

return purchases
```

Upon executing this small block of code, it will take the data file which contains the numbers and then puts them all into a list. This list is then handed to the object we previously created, and then this object process the data in the list and returns values according to the metrics we have defined for it (lowest sale price, highest sale price, and average sale price). However, we encounter one problem when using this code implementation to process such data, and this problem is none other than a performance bottleneck. If we introduce a slightly larger list of data than the one we are currently using, the time it would take for the program to process the data completely would actually be considerably longer than it normally should be. In other cases, if the list of data we are dealing with is large enough, the program might even throw the towel in before the event trying to work on it. This would be because the size of the data would be too large for it to even fit inside the system’s memory. Let’s improve the performance of this program so that the performance issue can be resolved.

To fix the problem, we must first understand how the program is actually working in the first place. Upon carefully reviewing the code, we come to know that data contained within the ‘**purchases**’ CSV file is being processed in a step-by-step manner. In other words, only a single data instance of the purchases CSV file is processed at any given moment. If this is how the code is fundamentally working, a question arises. Why is all of the data within the CSV file being stored in the system’s memory when only a single data instance is processed at any given time? Following this logic, one would argue that the main memory should only be storing a single data instance since that’s what the program is currently focused on.

The solution to solving this performance issue is to introduce a generator in the code. Once the program starts using a generator, it will stop throwing all of the data within the list into the system’s memory at once and then processing one data instance at a time, it will load a single data instance, process it, produce a result and then move on to the next data instance on the list. The following lines of code demonstrate the implementation of a generator.

```
def load_purchases(filename):

with open(filename) as f:

for line in f:

*_, price_raw = line.partition(",")
```

```
yield float(price_raw)
```

Using Idioms to Perform Iterations

In the previous sections, we have explored the generator expressions to perform efficient iteration in programs without compromising their performance or causing memory issues. Now, we will move on and explore different blocks of code that use idioms to perform iteration.

Performing Iteration Using Idioms

One of the most common tools used for iteration is a function that is generally known as ‘**enumerate()**’. The working of this function is actually pretty tricky to explain in words but regardless if you are left confused after reading the function’s description, the confusion will be cleared away once you see a practical implementation of this function.

So, when using the ‘enumerate()’ function, we need to provide it with a parameter so that it can properly work. The parameter we provide this function is actually the element that we want to iterate throughout the program. Once the function receives the element that needs to be iterated, it will give back another corresponding element which is actually a ‘tuple’. This tuple essentially has two components or parts. The first part is basically an element that is in reality, an enumerated value of the second element (this is why trying to understand the enumerator function in words is pretty difficult).

Here’s an implementation of the enumerate() function shown below:

```
>>> list(enumerate("abcdef"))

[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e'), (5, 'f')]
```

Now let’s say that we want to use an object which works in a similar fashion to the enumerate() function but is not as flashy or sophisticated as the enumerator function. We simply want this object to generate a series of numbers in a loop such that there is no limit. In other words, we want the object to generate an infinite iteration sequence.

On the surface, this may seem a very complex or difficult task that requires the use of an equally complex or sophisticated technique or method to create such an object but the reality is quite the opposite. We can fulfill this requirement by introducing an object as simple as the one shown below.

```
class NumberSequence:

def __init__(self, start=0):

self.current = start

def next(self):

current = self.current

self.current += 1

return current
```

According to the interface we are provided through this method, we would have to call upon the ‘**next()**’ method to successfully use the object.

```
>>> seq = NumberSequence()

>>> seq.next()

0

>>> seq.next()

1

>>> seq2 = NumberSequence(10)

>>> seq2.next()

10

>>> seq2.next()

11
```

However, this approach does not allow us to use the implementation of the ‘enumerate()’ function that we actually want. To elaborate, the interface given to use through this code has no native support for the implementation of the enumerator function using a ‘**for**’ loop. Since the enumerate() function cannot be iterated using a ‘**for**’ loop in this scenario, this also points to the fact that the enumerator cannot be passed as a parameter to other functions as well. For instance, the following code lines try to implement the enumerate() function we just discussed, but instead of successfully executing, the code just fails.

```
>>> list(zip(NumberSequence(), "abcdef"))

Traceback (most recent call last):

File "...", line 1, in <module>

TypeError: zip argument #1 must support iteration
```

The fundamental reason for this problem to arise is due to the fact the object ‘**NumberSequence()**’ cannot be simply iterated because it has no support for such a feature. In order to resolve this error, we need to somehow convert this object ‘**NumberSequence()**’ from being non-iterable to being iterating or this would be a serious roadblock for us. Luckily, we still have a few tricks up our sleeves as well. We give the ‘**NumerSequence()**’ object complete iteration capabilities by using the method ‘**__iter__()**’. In order to not have any chances of this failing, we also went ahead and made the object an iterator (once again) by changing the ‘**next()**’ method with ‘**__next__()**’. The following lines of code show the implementation of making this object iterable as well as an iterator.

```
class SequenceOfNumbers:

def __init__(self, start=0):

self.current = start

def __next__(self):

current = self.current

self.current += 1

return current

def __iter__(self):
```

```
return self
```

The Purpose of the Function ‘next()’

Basically, the ‘**next()**’ is a built-in function that is included in Python we use an iterator. The object which is iterating moves on to the next iteration cycle by calling this function. Thus, we can say that it is the component that keeps the iterator moving forward. So, without this function, the iterator would not be able to iterate upon the next element, much less return the element defined to be revealed in that cycle. Here’s an example of the use of the ‘**next()**’ function.

```
>>> word = iter("hello")

>>> next(word)

'h'

>>> next(word)

'e' # ...
```

The iterator does not have an infinite number of elements within it. At some point, it will reach the final element during the iteration and then have no element to produce over the next iteration. Once an iterator reaches this stage, there is no point for it to iterate any further as it would be nothing but a waste of computational resources. Thus, to stop the iteration process, an exception is raised which is namely ‘**StopIteration**’. A demonstration of this exception being raised has been shown below:

```
>>> ...

>>> next(word)

'o'

>>> next(word)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration
```

```
>>>
```

Once the program encounters this exception, this means that the process of iteration has been completed and the object has nothing left to give out during the next iteration.

Since exception handling is an important aspect of coding, we cannot ignore this exception and then call it a day or focus on some other aspect. We need to capture this exception flag as soon as it is raised and code an appropriate response. There are many ways to handle exceptions, so let’s set that aside for now (you can find a plethora of ways detailing how to handle exceptions on stackoverflow) and focus on the function. When the exception is raised, we can provide the function responsible for raising it a value and pass it to its second parameter. If the function is specified a value for the second parameter, in addition to raising the ‘**StopIteration’** exception flag, it will also display the value we passed to it. This has been demonstrated in the following lines of code.

```
>>> next(word, "default value")

'default value'
```

Using Generators

We can also implement generators to perform the iteration shown in the previous sections. This would improve the iteration process and the program’s performance and make it easier to deal with objects instead of manually making them iterable since generator objects are iterators by nature. The following lines of code demonstrate the use of a generator that produces the values we need by using the function ‘**yield()**’.

```
def sequence(start=0):

    """

    >>> import inspect

    >>> seq = sequence()

    >>> inspect.getgeneratorstate(seq)
```

```
'GEN_CREATED'

>>> seq = sequence()
>>> next(seq)
0
>>> inspect.getgeneratorstate(seq)
'GEN_SUSPENDED'

>>> seq = sequence()
>>> next(seq)
0
>>> seq.close()
>>> inspect.getgeneratorstate(seq)
'GEN_CLOSED'
>>> next(seq)  # doctest: +ELLIPSIS
Traceback (most recent call last):
  ...
StopIteration

"""
while True:
    yield start
    start += 1
```

We need to keep this thing in our minds that the function ‘**yield**’ is the keyword in this entire example. The element which makes the entire function act as a generator is none other than ‘**yield**’ and since we are working with a generator this time, we can easily create an iteration loop that goes on without any end, i.e., for an infinite number of the iteration cycles. Another fun functionality of generators is that even if we have an infinite number of iterations, the generator function will execute all of the code it encounters until it reaches a point where it meets another ‘**yield**’ statement. At this point, the generator function will simply return its value and then exit from the program.

```
>>> seq = sequence(10)

>>> next(seq)

10

>>> next(seq)

11

>>> list(zip(sequence(), "abcdef"))

[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e'), (5, 'f')]
```

Conclusion

We have finally come to the end of our journey of learning different techniques and conventions to help us effectively make our code cleaner and thus, easier to maintain and reuse in future projects.

At the start of this book, we first went through a comprehensive introduction to the Python Programming Language and learned the importance of clean code and why it is necessary in our projects. Then, we were acquainted with some tools and techniques that will help us format our code more easily and in an efficient manner. We specifically learned about the Mypy tool and the Pylit tool along with some coding design patterns that we absolutely need to uphold if we want our code to work perfectly. We then moved on to explore more of the code formatting concepts in the next chapter, particularly indexes and slices along with the purpose of context managers in Python and the importance of iteration as a tool in programming. We then moved on to develop a clear understanding of what are the common characteristics found in well-built and clean code. Then we stopped by to learn one of the most important concepts in keeping our code clean, i.e., the five core principles of good software design, we also learned about a useful acronym that can help us memorize each of these five principles without any difficulties. Then we learned about decorators that were teased in the beginning chapters and explored how they are intended to be used and how to properly implement them. Finally, we land on the last chapter of this book and build upon the discussion of iteration we briefly touched in the second chapter.

We learned a lot about clean code. We hope that the readers enjoyed this journey through some pretty cool software engineering techniques and became more knowledgeable in matters of programming, problem-solving, and clean coding.

References

Clean Code in Python by Mariano Anaya