Clean Code in Python

Sergio Bugallo

Feb 21, 2020

CONTENTS

CHAPTER

ONE

DOCSTRINGS AND ANNOTATIONS

1.1 1. Docstrings

Docstrings are basically documentation embedded in the source code. A **docstring** is basically a literal string, placed somewhere in the code, with the intention of documenting that part of the logic. This information it's meant to represent explanation, not justification.

Having comments in the code is a bad practice for multiple reasons. First, they represent our failure to express our ideas in the code. Second, it can be misleading. Worst than having to spend some time reading a complicated section is to read a comment on how it is supposed to work and figuring out that the code actually does something different.

Sometimes, we cannot avoid having comments (maybe there is an error on a third-party library). In those cases, placing a small but descriptive comment might be acceptable.

The reason why docstrings are a good thing to have in the code is that Python is dynamically typed. Python will not enforce, nor check, anything like the value for any function's input parameters. Documenting the expected input and output of a function is a good practice that will help the readers of that function understand how it is supposed to work. This information is crucial for someone that hats to learn and understand how a new code works, and how they can take advantage of it.

The docstring is not something separated or isolated from the code. It becomes part of the code, and you can access it. When an object has a docstring defined, this becomes part of it via its `__doc__` attribute:

```
def sample():
    """Sample docstring"""
    return

>>> sample.__doc__
'Sample docstring'
```

There is, unfortunately, one downside to docstrings, and it is that, as it happens with all documentation, it requires manual and constant maintenance. As the code changes, it will have to be updated. Another problem is that for docstrings to be really useful, they have to be detailed, which requires multiple lines.

1.2 2. Annotations

The basic idea is to hint to the readres of the code about what to expect as values of arguments in functions. Annotations enable type hinting.

Annotations let you specify the expected type of some variables that have been defined. It is actually not only about the types, but any kind of metadata that can help you get a better idea of what that variable actually represents.

```
class Point:
    def __init__(self, lat, lon):
        self.lat = lat
```

(continues on next page)

(continued from previous page)

```
        self.lon = lon

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

Here, we use `float` to indicate the expected types of input parameters. This is merely informative for the reader, Python will not check these types nor enforce them. We can also specify the expected type of the returned value of the function. In this case, `Point` is a user-defined class, so it will mean that whatever is returned will be an instance of `Point`.

With the introduction of annotations, a new special attribute is also included, and it is `__annotations__`. This will give us access to a dictionary that maps the n ame of the annotations with their corresponding values, which are those we have defined for them:

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

The idea of type hinting is to have extra tools to check and assess the correct use of types throughout the code and to hint to the user in case any incompatibilities are detected.

Starting with Python 3.5, the new typing module was introduced, and this significantly improved how hwe define the types and the annotations in our Python code. The basic idea is that now the semantics extend to more meaningful concepts. For example, you could have a function that worked with lists of tuples in one of its parameters, and you would have put one of these two types as the annotation, or even a string explaining it. But with this module, it is possible to tell Python that it expects an iterable or a sequence. You can even identify the type or the values on it.

There is one extra improvement made in regards to annotations starting from Python 3.6. It is possible to annotate variables directly, not just function parameters and return types. The idea is that you can declare the types of some variables defined without necessarily assigning a value to them:

```
class Point:
    lat: float
    lon: float

>>> Point.__annotations__
{'lat': <class 'float'>, 'lon': <class 'float'>}
```

1.2.1 2.1. Do annotations replace docstrings?

The short answer is no, and this is because they complement each other. It is true that a part of the information previously contained on the docstring can now be moved to the annotations. But this should only leave more room for a better documentation on the docstring. In particular, for dynamic and nested data types, it is always a good idea to provide examples of the expected data so that we can get a better idea of what we are dealing with.

```
def data_from_response(response: dict) -> dict:
    """
    If the response is OK, return its payload.

    Arguments
    ---------
    response: A dict like::
        {
            "status": 200, # <int>
            "timestamp": "...", # <date time>
            "payload": {...} # <dict>
        }
```

(continues on next page)

(continued from previous page)

```
    Returns
    -------
    result: A dict like::
        {"data": {...}}

    Raises
    ------
    ValueError: if the HTTP status is not 200.
    """
    if response["status"] != 200:
        raise ValueError

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

Now, we have a complete idea of what is expected to be received and returned by this function. The documentation serves as valuable input, not only for understanding and getting an idea of what is being passed around, but also as a valuable source for unit tests. We can derive data like this to use as input, and we know what would be the correct and incorrect values to use on the tests.

The benefit is that now we know what the possible values of the keys are, as well as their types, and we have a more concrete interpretation of what the data looks like. The cost is that, as we mentioned earlier, it takes up a lot of lines and it needs to be verbose and detailed to be effective.

CHAPTER

TWO

PYTHONIC CODE

2.1 1. Indexes and slices

In Python, some data structures or types support accesing it's elements by index. The first element is placed in the index number zero. How would you access the last element of a list?

```
>>> numbers = (1, 2, 3, 4, 5)
>>> numbers[-1]
5
>>> numbers[-3]
3
```

In addition, we can obtain many elements by using `slice`:

```
>>> numbers[2:5]
(3, 4, 5)
```

In this case, the syntax means that we get all of the elements on the tuple, starting from the index of the first number (inclusive), up to the index on the second one (not including it).

You can exclude either one of the intervals, start or stop, and in that case, it will act from the beginning or end of the sequence:

```
>>>  numbers[:3]
(1, 2, 3)
>>> numbers[3:]
(4, 5)
>>> numbers[::]
(1, 2, 3, 4, 5)
>>> numbers[1:5:2]
(2, 4)
```

In the first example, it will everything up to index 3. In the second example, it will get all numbers starting from index 3. In the third example, where both ends are excluded, it is actually creating a copy of the original tuple. The last example includes a third parameter, which is the step.

In all of these cases, when we pass intervals to a sequence, what is actually happening is that we are passing a `slice`. Note that it is a built-in object in Python that you can build yourself and pass directly:

```
>>> interval = slice(1,5,2)
>>>  numbers[interval]
(2, 4)
>>> interval = slice(None, 3)
>>>  numbers[:3] == numbers[interval]
True
```

2.1.1 1.1. Creating your own sequences

The functionality we just discussed works thanks to a magic method called `__getitem__`. This is the method that is called when something like `object[key]` is called, passing the key as a parameter. A sequence is an object that implements both `__getitem__` and `__len__`, and for this reason, it can be iterated over.

In the case that your class is a wrapper around a standard library object, you might as well delegate the behavior as much as possible to the underlying object. This means that if your class is actually a wrapper on the list, call all of the same methods on that list to make sure that it remains compatible. In the following listing, we can see an example of how an object wraps a list, and for the methods we are interested in, we just delegate 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)
```

If you are implementing your own sequence then keep in mind the following points:

- When indexing by a range, the result should be an instance of the same type of the class.
- In the range provided by the slice, respect the semantics that Python uses, excluding the element at the end.

2.2 2. Context managers

Context managers are quite useful since they correctly respond to a pattern. The pattern is actually every situation where we want to run some code, and has preconditions and postconditions, meaning that we want to run things before and after a certain main action.

Most of the time, we see context managers around resource management. For example, on situations when we open files, we want to make sure that they are closed after processing (so we do not leak file descriptors), or if we open a connection to a service (or even a socket), we also want to be sure to close it accordingly, or when removing temporary files, and so on.

In all of these cases, you would normally have to remember to free all of the resources that were allocated and that is just thinking about the best case—but what about exceptions and error handling? Given the fact that handling all possible combinations and execution paths of our program makes it harder to debug, the most common way of addressing this issue is to put the cleanup code on a `finally` block so that we are sure we do not miss it. For example, a very simple case would look like the following:

```
fd = open(filename)
try:
    process_file(fd)
finally:
    fd.close()
```

Nonetheless, there is a much elegant and Pythonic way of achieving the same thing:

```
with open(filename) as fd:
    process_file(fd)
```

The with statement enters the context manager. In this case, the open function implements the context manager protocol, which means that the file will be automatically closed when the block is finished, even if an exception occurred.

Context managers consist of two magic methods: `__enter__` and `__exit__`. On the first line of the context manager, the with statement will call the first method, `__enter__`, and whatever this method returns will be assigned to the variable labeled after as. This is optional—we don't really need to return anything specific on the `__enter__` method, and even if we do, there is still no strict reason to assign it to a variable if it is not required.

After this line is executed, the code enters a new context, where any other Python code can be run. After the last statement on that block is finished, the context will be exited, meaning that Python will call the `__exit__` method of the original context manager object we first invoked.

If there is an exception or error inside the context manager block, the `__exit__` method will still be called, which makes it convenient for safely managing cleaning up conditions. In fact, this method receives the exception that was triggered on the block in case we want to handle it in a custom fashion.

Despite the fact that context managers are very often found when dealing with resources, this is not the sole application they have. We can implement our own context managers in order to handle the particular logic we need.

Context managers are a good way of separating concerns and isolating parts of the code that should be kept independent, because if we mix them, then the logic will become harder to maintain.

As an example, consider a situation where we want to run a backup of our database with a script. The caveat is that the backup is offline, which means that we can only do it while the database is not running, and for this we have to stop it. After running the backup, we want to make sure that we start the process again, regardless of how the process of the backup itself went. Now, the first approach would be to create a huge monolithic function that tries to do everything in the same place, stop the service, perform the backup task, handle exceptions and all possible edge cases, and then try to restart the service again. You can imagine such a function, and for that reason, I will spare you the details, and instead come up directly with a possible way of tackling this issue with context managers:

```python
class DBHandler:

    def stop_database():
        run("systemctl stop postgresql.service")

    def start_database():
        run("systemctl start postgresql.service")

    def __enter__(self):
        self.stop_database()
        return self

    def __exit__(self, exc_type, ex_value, ex_traceback):
        self.start_database()

def db_backup():
    run("pg_dump database")

def main():
    with DBHandler():
        db_backup()
```

As a general rule, it should be good practice (although not mandatory), to always return something on the `__enter__`.

Notice the signature of the `__exit__` method. It receives the values for the exception that was raised on the block. If there was no exception on the block, they are all none.

The return value of `__exit__` is something to consider. Normally, we would want to leave the method as it is, without returning anything in particular. If this method returns True, it means that the exception that was potentially raised will not propagate to the caller and will stop there. Sometimes, this is the desired effect, maybe even depending on the type of exception that was raised, but in general it is not a good idea to swallow the exception. Remember: errors should never pass silently.

Keep in mind not to accidentally return True on the `__exit__`. If you do, make sure that this is exactly what you want, and that there is a good reason for it.

2.2.1 2.1. Implementing context managers

In general, we can implement context managers implementing the `__enter__` and `__exit__` magic methods, and then that object will be able to support the context manager protocol. While this is the most common way for context managers to be implemented, it is not the only one.

The `contextlib` module contains a lot of helper functions and objects to either implement context managers or use some already provided ones that can help us write more compact code.

Let's start by looking at the `contextmanager` decorator. When the `contextlib.contextmanager` decorator is applied to a function, it converts the code on that function into a context manager. The function in question has to be a particular kind of function called a generator function, which will separate the statements into what is going to be on the `__enter__` and `__exit__` magic methods, respectively.

The equivalent code of the previous example can be rewritten with the `contextmanager` decorator like this:

```
import contextlib

@contextlib.contextmanager
def db_handler():
    stop_database()
    yield
    start_database()

with db_handler():
 db_backup()
```

Here, we define the generator function and apply the `@contextlib.contextmanager` decorator to it. The function contains a yield statement, which makes it a generator function. Again, details on generators are not relevant in this case. All we need to know is that when this decorator is applied, everything before the yield statement will be run as if it were part of the `__enter__` method. Then, the yielded value is going to be the result of the context manager evaluation (what `__enter__` would return), and what would be assigned to the variable if we chose to assign it.

At that point, the generator function is suspended, and the context manager is entered, where, again, we run the backup code for our database. After this completes, the execution resumes, so we can consider that every line that comes after the yield statement will be part of the `__exit__` logic.

Another helper we could use is `contextlib.ContextDecorator`. This is a mixin base class that provides the logic for applying a decorator to a function that will make it run inside the context manager, while the logic for the context manager itself has to be provided by implementing the aforementioned magic methods.

In order to use it, we have to extend this class and implement the logic on the required methods:

```
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")
```

There is no with statement. We just have to call the function, `and offline_backup()` will automatically run inside a context manager. This is the logic that the base class provides to use it as a decorator that wraps the original function so that it runs inside a context manager.

The only downside of this approach is that by the way the objects work, they are completely independent (the decorator doesn't know anything about the function that is decorating, and vice versa. This, however good, means that you cannot get an object that you would like to use inside the context manager, so if you really need to use the object returned by the `__exit__` method, one of the previous approaches will have to be the one of choice.

Being a decorator also poses the advantage that the logic is defined only once, and we can reuse it as many times as we want by simply applying the decorators to other functions that require the same invariant logic.

Note that `contextlib.suppress` is a util package that enters a context manager, which, if one of the provided exceptions is raised, doesn't fail. It's similar to running that same code on a try/except block and passing an exception or logging it, but the difference is that calling the suppress method makes it more explicit that those exceptions that are controlled as part of our logic. For example, consider the following code:

```
import contextlib

with contextlib.suppress(DataConversionException):
     parse_data(input_json_or_dict)
```

Here, the presence of the exception means that the input data is already in the expected format, so there is no need for conversion, hence making it safe to ignore it.

2.3 3. Properties, attributes and different types of methods for objects

All of the properties and functions of an object are public in Python, which is different from other languages where properties can be public, private, or protected. That is, there is no point in preventing caller objects from invoking any attributes an object has. This is another difference with respect to other programming languages in which you can mark some attributes as private or protected.

There is no strict enforcement, but there are some conventions. An attribute that starts with an underscore is meant to be private to that object, and we expect that no external agent calls it (but again, there is nothing preventing this).

2.3.1 3.1. Underscores in Python

Consider the following example to illustrate this:

```
class Connector:

    def __init__(self, source, user, password, timeout):
        self.source = source
        self.user = user
        self.__password = password
        self._timeout = timeout
```

```
>>> Connector(...).source
'postgresql://localhost'

>>> Connector(...)._timeout
60

>>> Connector(...).__password
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
AttributeError: 'Connector' object has no attribute '__password'

>>> vars(Connector(...))
{'source': 'postgresql://localhost', '_timeout': 60, 'user': 'root', '_Connector__
↪password': '1234'}
```

(continues on next page)

(continued from previous page)

Here, a Connector object is created with source, and it starts with 4 attributes—the aforementioned `source`, `timeout`, `user` and `password`. `source` and `user` are public, `timeout` is private and `password` is.

However, as we can see from the following lines when we create an object like this, we can actually access `timeout`. The interpretation of this code is that `_timeout` should be accessed only within connector itself and never from a caller. This means that you should organize the code in a way so that you can safely refactor the timeout at all of the times it's needed, relying on the fact that it's not being called from outside the object (only internally), hence preserving the same interface as before. Complying with these rules makes the code easier to maintain and more robust because we don't have to worry about ripple effects when refactoring. The same principle applies to methods as well.

Note: Objects should only expose those attributes and methods that are relevant to an external caller object, namely, entailing its interface. Everything that is not strictly part of an object's interface should be kept prefixed with a single underscore.

This is the Pythonic way of clearly delimiting the interface of an object. There is, however, a common misconception that some attributes and methods can be actually made private. This is, again, a misconception.

`password` is defined with a double underscore instead. Some developers use this method to hide some attributes, thinking, like in this example, that `password` is now private and that no other object can modify it. Now, take a look at the exception that is raised when trying to access it. It's `AttributeError`, saying that it doesn't exist. It doesn't say something like "this is private" or "this can't be accessed" and so on. It says it does not exist. This should give us a clue that, in fact, something different is happening and that this behavior is instead just a side effect, but not the real effect we want.

What's actually happening is that with the double underscores, Python creates a different name for the attribute (this is called **name mangling**). What it does is create the attribute with the following name instead: "_<class-name>__<attribute-name>". In this case, an attribute named '_Connector__password' will be created.

Notice the side effect that we mentioned earlier—the attribute only exists with a different name, and for that reason the AttributeError was raised on our first attempt to access it.

The idea of the double underscore in Python is completely different. It was created as a means to override different methods of a class that is going to be extended several times, without the risk of having collisions with the method names. Even that is a too far-fetched use case as to justify the use of this mechanism.

Double underscores are a non-Pythonic approach. If you need to define attributes as private, use a single underscore, and respect the Pythonic convention that it is a private attribute.

Note: Do not use double underscores.

2.3.2 3.2 Properties

When the object needs to just hold values, we can use regular attributes. Sometimes, we might want to do some computations based on the state of the object and the values of other attributes. Most of the time, properties are a good choice for this.

Properties are to be used when we need to define access control to some attributes in an object, which is another point where Python has its own way of doing things. In other programming languages (like Java), you would create access methods (getters and setters), but idiomatic Python would use properties instead.

Imagine that we have an application where users can register and we want to protect certain information about the user from being incorrect, such as their email, as shown in the following code:

```
import re

EMAIL_FORMAT = re.compile(r"[^@]+@[^@]+\.[^@]+")

def is_valid_email(potentially_valid_email: str):
    return re.match(EMAIL_FORMAT, potentially_valid_email) is not None

class User:
     def __init__(self, username):
         self.username = username
         self._email = None

     @property
     def email(self):
        return self._email

     @email.setter
     def email(self, new_email):
         if not is_valid_email(new_email):
            raise ValueError(f"Can't set {new_email} as it's not a valid email")
         self._email = new_email
```

By putting `email` under a property, we obtain some advantages for free. In this example, the first `@property` method will return the value held by the private attribute `email`. As mentioned earlier, the leading underscore determines that this attribute is intended to be used as private, and therefore should not be accessed from outside this class.

Then, the second method uses `@email.setter`, with the already defined property of the previous method. This is the one that is going to be called when `<user>.email = <new_email>` runs from the caller code, and `<new_email>` will become the parameter of this method. Here, we explicitly defined a validation that will fail if the value that is trying to be set is not an actual email address. If it is, it will then update the attribute with the new value as follows:

```
>>> u1 = User("jsmith")
>>> u1.email = "jsmith@"
Traceback (most recent call last):
...
ValueError: Can't set jsmith@ as it's not a valid email
>>> u1.email = "jsmith@g.co"
>>> u1.email
'jsmith@g.co'
```

This approach is much more compact than having custom methods prefixed with `get_` or `set_`. It's clear what is expected because it's just email.

Note: Don't write custom `get_*` and `set_*` methods for all attributes on your objects. Most of the time, leaving them as regular attributes is just enough. If you need to modify the logic for when an attribute is retrieved or modified, then use properties.

You might find that properties are a good way to achieve command and query separation (CC08). Command and query separation state that a method of an object should either answer to something or do something, but not both. If a method of an object is doing something and at the same time it returns a status answering a question of how that operation went, then it's doing more than one thing, clearly violating the principle that functions should do one thing, and one thing only.

Depending on the name of the method, this can create even more confusion, making it harder for readers to understand what the actual intention of the code is. For example, if a method is called set_email, and we use it as if self.set_email("a@j.com"): . . . , what is that code doing? Is it setting the email to a@j.com? Is it checking if the email is already set to that value? Both (setting and then checking if the status is correct)?

With properties, we can avoid this kind of confusion. The `@property` decorator is the query that will answer to something, and the `@<property_name>.setter` is the command that will do something.

Another piece of good advice derived from this example is as follows: don't do more than one thing on a method. If you want to assign something and then check the value, break that down into two or more sentences.

Note: Methods should do one thing only. If you have to run an action and then check for the status, so that in separate methods that are called by different statements.

2.4 4. Iterable objects

In Python, we have objects that can be iterated by default: lists, tuples, sets and dictionaries. However, the built-in iterable objects are not the only kind that we can have in a for loop. We could also create our own iterable, with the logic we define for iteration.

In order to achieve this, we rely on magic methods. Iteration works in Python by its own protocol (namely the iteration protocol). When you try to iterate an object in the form `for e in myobject:...`, what Python checks at a very high level are the following two things, in order:

- If the object contains one of the iterator methods `__next__` or `__iter__`
- If the object is a sequence and has `__len__` and `__getitem__`

Therefore, as a fallback mechanism, sequences can be iterated, and so there are two ways of customizing our objects to be able to work on for loops.

2.4.1 4.1. Creating iterable objects

When we try to iterate an object, Python will call the `iter()` function over it. One of the first things this function checks for is the presence of the `__iter__` method on that object, which, if present, will be executed.

The following code creates an object that allows iterating over a range of dates, producing one day at a time on every round of the loop:

```python
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
```

This object is designed to be created with a pair of dates, and when iterated, it will produce each day in the interval of specified dates, which is shown in the following code:

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

(continues on next page)

(continued from previous page)

```
2018-01-01
2018-01-02
2018-01-03
2018-01-04
```

Here, the for loop is starting a new iteration over our object. At this point, Python will call the `iter()` function on it, which in turn will call the `__iter__` magic method. On this method, it is defined to return `self`, indicating that the object is an iterable itself, so at that point every step of the loop will call the `next()` function on that object, which delegates to the `__next__` method. In this method, we decide how to produce the elements and return one at a time. When there is nothing else to produce, we have to signal this to Python by raising the StopIteration exception.

This means that what is actually happening is similar to Python calling `next()` every time on our object until there is a StopIteration exception, on which it knows it has to stop the for loop:

This example works, but it has a small problem—once exhausted, the iterable will continue to be empty, hence raising StopIteration. This means that if we use this on two or more consecutive for loops, only the first one will work, while the second one will be empty:

```
>>> r1 = DateRangeIterable(date(2018, 1, 1), date(2018, 1, 5))
>>> ", ".join(map(str, r1))
'2018-01-01, 2018-01-02, 2018-01-03, 2018-01-04'
>>> max(r1)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: max() arg is an empty sequence
```

This is because of the way the iteration protocol works: an iterable constructs an iterator, and this one is the one being iterated over. In our example, `__iter__` just returned self, but we can make it create a new iterator every time it is called. One way of fixing this would be to create new instances of DateRangeIterable, which is not a terrible issue, but we can make `__iter__` use a generator (which are iterator objects), which is being created every time:

```
class DateRangeContainerIterable:

    def __init__(self, start_date, end_date):
        self.start_date = start_date
        self.end_date = end_date

    def __iter__(self):
        current_day = self.start_date
        while current_day < self.end_date:
            yield current_day

        current_day += timedelta(days=1)
```

And this time, it works:

```
>>> r1 = DateRangeContainerIterable(date(2018, 1, 1), date(2018, 1, 5))
>>> ", ".join(map(str, r1))
'2018-01-01, 2018-01-02, 2018-01-03, 2018-01-04'
>>> max(r1)
datetime.date(2018, 1, 4)
```

The difference is that each for loop is calling `__iter__` again, and each one of those is creating the generator again. This is called a container iterable.

..note:: In general, it is a good idea to work with container iterables when dealing with generators.

2.4.2 4.2. Creating sequences

Maybe our object does not define the `__iter__()` method, but we still want to be able to iterate over it. If `__iter__` is not defined on the object, the `iter()` function will look for the presence of `__getitem__`, and if this is not found, it will raise TypeError.

A sequence is an object that implements `__len__` and `__getitem__` and expects to be able to get the elements it contains, one at a time, in order, starting at zero as the first index. This means that you should be careful in the logic so that you correctly implement `__getitem__` to expect this type of index, or the iteration will not work.

The example from the previous section had the advantage that it uses less memory. This means that is only holding one date at a time, and knows how to produce the days one by one. However, it has the drawback that if we want to get the n-th element, we have no way to do so but iterate n-times until we reach it. This is a typical trade-off in computer science between memory and CPU usage.

The implementation with an iterable will use less memory, but it takes up to O(n) to get an element, whereas implementing a sequence will use more memory (because we have to hold everything at once), but supports indexing in constant time, O(1).

This is what the new implementation might look like:

```python
class DateRangeSequence:
     def __init__(self, start_date, end_date):
         self.start_date = start_date
         self.end_date = end_date
         self._range = self._create_range()

     def _create_range(self):
         days = []
         current_day = self.start_date

         while current_day < self.end_date:
             days.append(current_day)
             current_day += timedelta(days=1)

         return days

     def __getitem__(self, day_no):
        return self._range[day_no]

     def __len__(self):
        return len(self._range)
```

Here is how the object behaves:

```
>>> s1 = DateRangeSequence(date(2018, 1, 1), date(2018, 1, 5))
>>> for day in s1:
...     print(day)
2018-01-01
2018-01-02
2018-01-03
2018-01-04
>>> s1[0]
datetime.date(2018, 1, 1)
>>> s1[3]
datetime.date(2018, 1, 4)
>>> s1[-1]
datetime.date(2018, 1, 4)
```

In the preceding code, we can see that negative indices also work. This is because the DateRangeSequence object delegates all of the operations to its wrapped object (a list), which is the best way to maintain compatibility and a consistent behavior.

Evaluate the trade-off between memory and CPU usage when deciding which one of the two possible implementations to use. In general, the iteration is preferable (and generators even more), but keep in mind the requirements of every case.

2.5 5. Container objects

Containers are objects that implement a `__contains__` method (that usually returns a Boolean value). This method is called in the presence of the `in` keyword of Python. Something like `element in container` becomes `container.__contains__(element)`.

You can imagine how much more readable and Pythonic the code can be when this method is properly implemented.

Let's say we have to mark some points on a map of a game that has two-dimensional coordinates. We might expect to find a function like the following:

```python
def mark_coordinate(grid, coord):
    if 0 <= coord.x < grid.width and 0 <= coord.y < grid.height:
        grid[coord] = MARKED
```

Now, the part that checks the condition of the first if statement seems convoluted; it doesn't reveal the intention of the code, it's not expressive, and worst of all it calls for code duplication (every part of the code where we need to check the boundaries before proceeding will have to repeat that if statement).

What if the map itself (called grid on the code) could answer this question? Even better, what if the map could delegate this action to an even smaller (and hence more cohesive) object? Therefore, we can ask the map if it contains a coordinate, and the map itself can have information about its limit, and ask this object the following:

```python
class Boundaries:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def __contains__(self, coord):
        x, y = coord
        return 0 <= x < self.width and 0 <= y < self.height

class Grid:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.limits = Boundaries(width, height)

    def __contains__(self, coord):
        return coord in self.limits
```

This code alone is a much better implementation. First, it is doing a simple composition and it's using delegation to solve the problem. Both objects are really cohesive, having the minimal possible logic; the methods are short, and the logic speaks for itself: `coord in self.limits` is pretty much a declaration of the problem to solve, expressing the intention of the code.

From the outside, we can also see the benefits. It's almost as if Python is solving the problem for us:

```python
def mark_coordinate(grid, coord):
    if coord in grid:
        grid[coord] = MARKED
```

2.6 6. Dynamic attributes for objects

It is possible to control the way attributes are obtained from objects by means of the `__getattr__` magic method. When we call something like `<myobject>.<myattribute>`, Python will look for `<myattribute>` in the dictionary of the object, calling `__getattribute__` on it. If this is not found (namely, the object does not have the attribute we are looking for), then the extra method, `__getattr__`, is called, passing the name of the attribute (myattribute) as a parameter. By receiving this value, we can control the way things should be returned to our objects. We can even create new attributes, and so on.

In the following listing, the `__getattr__` method is demonstrated:

```python
class DynamicAttributes:
    def __init__(self, attribute):
        self.attribute = attribute

    def __getattr__(self, attr):
        if attr.startswith("fallback_"):
            name = attr.replace("fallback_", "")
            return f"[fallback resolved] {name}"
        raise AttributeError(f"{self.__class__.__name__} has no attribute {attr}")
```

Here are some calls to an object of this class:

The first call is straightforward, we just request an attribute that the object has and get its value as a result. The second is where this method takes action because the object does not have anything called `fallback_test`, so the `__getattr__` will run with that value. Inside that method, we placed the code that returns a string, and what we get is the result of that transformation.

The third example is interesting because there a new attribute named fallback_new is created (actually, this call would be the same as running `dyn.fallback_new = "new value"`), so when we request that attribute, notice that the logic we put in `__getattr__` does not apply, simply because that code is never called.

Now, the last example is the most interesting one. There is a subtle detail here that makes a huge difference. Take another look at the code in the `__getattr__` method. Notice the exception it raises when the value is not retrievable AttributeError. This is not only for consistency (as well as the message in the exception) but also required by the builtin `getattr()` function. Had this exception been any other, it would raise, and the default value would not be returned.

Note: Be careful when implementing a method so dynamic as `__getattr__`, and use it with caution. When implementing it, raise AttributeError.

2.7 7. Callable objects

It is possible (and often convenient) to define objects that can act as functions. One of the most common applications for this is to create better decorators, but it's not limited to that.

The magic method `__call__` will be called when we try to execute our object as if it were a regular function. Every argument passed to it will be passed along to the `__call__` method. The main advantage of implementing functions this way, through objects, is that objects have states, so we can save and maintain information across calls.

When we have an object, a statement like this `object(*args, **kwargs)` is translated in Python to `object.__call__(*args, **kwargs)`. This method is useful when we want to create callable objects that will work as parametrized functions, or in some cases functions with memory.

The following listing uses this method to construct an object that when called with a parameter returns the number of times it has been called with the very same value:

```
from collections import defaultdict

class CallCount:
    def __init__(self):
        self._counts = defaultdict(int)

    def __call__(self, argument):
        self._counts[argument] += 1
        return self._counts[argument]
```

Some examples of this class in action are as follows:

```
>>> cc = CallCount()
>>> cc(1)
1
>>> cc(2)
1
>>> cc(1)
2
>>> cc(1)
3
>>> cc("something")
1
```

2.8 8. Caveats in Python

2.8.1 8.1. Mutable default arguments

Simply put, don't use mutable objects as the default arguments of functions. If you use mutable objects as default arguments, you will get results that are not the expected ones. Consider the following erroneous function definition:

```
def wrong_user_display(user_metadata: dict = {"name": "John", "age": 30}):
    name = user_metadata.pop("name")
    age = user_metadata.pop("age")

    return f"{name} ({age})"
```

This has two problems, actually. Besides the default mutable argument, the body of the function is mutating a mutable object, hence creating a side effect. But the main problem is the default argument for `user_medatada`.

This will actually only work the first time it is called without arguments. For the second time, we call it without explicitly passing something to `user_metadata`. It will fail with a KeyError, like so:

```
>>> wrong_user_display()
'John (30)'
>>> wrong_user_display({"name": "Jane", "age": 25})
'Jane (25)'
>>> wrong_user_display()
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File ... in wrong_user_display
 name = user_metadata.pop("name")
KeyError: 'name'
```

The explanation is simple—by assigning the dictionary with the default data to `user_metadata` on the definition of the function, this dictionary is actually created once and the variable `user_metadata` points to it. The body of the function modifies this object, which remains alive in memory so long as the program is running. When we pass a value to it, this will take the place of the default argument we just created. When we don't want this object it is called again, and it has been modified since the previous run; the next time we run it, will not contain the keys since they were removed on the previous call.

The fix is also simple: we need to use None as a default sentinel value and assign the default on the body of the function. Because each function has its own scope and life cycle, `user_metadata` will be assigned to the dictionary every time None appears:

```
def user_display(user_metadata: dict = None):
    user_metadata = user_metadata or {"name": "John", "age": 30}
    name = user_metadata.pop("name")
    age = user_metadata.pop("age")

    return f"{name} ({age})"
```

2.8.2 8.2. Extending built-in types

The correct way of extending built-in types such as lists, strings, and dictionaries is by means of the collections module.

If you create a class that directly extends dict, for example, you will obtain results that are probably not what you are expecting. The reason for this is that in CPython the methods of the class don't call each other (as they should), so if you override one of them, this will not be reflected by the rest, resulting in unexpected outcomes. For example, you might want to override `__getitem__`, and then when you iterate the object with a for loop, you will notice that the logic you have put on that method is not applied.

This is all solved by using collections.UserDict, for example, which provides a transparent interface to actual dictionaries, and is more robust.

Let's say we want a list that was originally created from numbers to convert the values to strings, adding a prefix. The first approach might look like it solves the problem, but it is erroneous:

```
class BadList(list):
     def __getitem__(self, index):
         value = super().__getitem__(index)
         if index % 2 == 0:
            prefix = "even"
         else:
            prefix = "odd"
         return f"[{prefix}] {value}"
```

At first sight, it looks like the object behaves as we want it to. But then, if we try to iterate it (after all, it is a list), we find that we don't get what we wanted:

```
>>> bl = BadList((0, 1, 2, 3, 4, 5))
>>> bl[0]
'[even] 0'
>>> bl[1]
'[odd] 1'
>>> "".join(bl)
Traceback (most recent call last):
...
TypeError: sequence item 0: expected str instance, int found
```

The join function will try to iterate (run a for loop over) the list, but expects values of type string. This should work because it is exactly the type of change we made to the list, but apparently when the list is being iterated, our changed version of the __getitem__ is not being called.

This issue is actually an implementation detail of CPython (a C optimization), and in other platforms such as PyPy it doesn't happen. Regardless of this, we should write code that is portable and compatible in all implementations, so we will fix it by extending not from list but from UserList:

```python
from collections import UserList

class GoodList(UserList):
    def __getitem__(self, index):
        value = super().__getitem__(index)
        if index % 2 == 0:
            prefix = "even"
        else:
            prefix = "odd"
        return f"[{prefix}] {value}"
```

And now things look much better:

```
>>> gl = GoodList((0, 1, 2))
>>> gl[0]
'[even] 0'
>>> gl[1]
'[odd] 1'
>>> "; ".join(gl)
'[even] 0; [odd] 1; [even] 2'
```

Note: Don't extend directly from `dict`, use `collections.UserDict` instead. For lists, use `collections.UserList`, and for strings, use `collections.UserString`.

CHAPTER

THREE

GENERAL TRAITS OF GOOD CODE

3.1 1. Design by contract

Some parts of the software we are working on are not meant to be called directly by users, but instead by other parts of the code. Such is the case when we divide the responsibilities of the application into different components or layers, and we have to think about the interaction between them.

We will have to encapsulate some functionality behind each component, and expose an interface to clients who are going to use that functionality, namely an **Application Programming Interface (API)**. The functions, classes, or methods we write for that component have a particular way of working under certain considerations that, if they are not met, will make our code crash. Conversely, clients calling that code expect a particular response, and any failure of our function to provide this would represent a defect. That is to say that if, for example, we have a function that is expected to work with a series of parameters of type integers, and some other function invokes our passing strings, it is clear that it should not work as expected, but in reality, the function should not run at all because it was called incorrectly (the client made a mistake). This error should not pass silently.

Of course, when designing an API, the expected input, output, and side-effects should be documented. But documentation cannot enforce the behavior of the software at runtime. These rules, what every part of the code expects in order to work properly and what the caller is expecting from them, should be part of the design, and here is where the concept of a contract comes into place.

The idea behind the DbC is that instead of implicitly placing in the code what every party is expecting, both parties agree on a contract that, if violated, will raise an exception, clearly stating why it cannot continue.

In our context, a contract is a construction that enforces some rules that must be honored during the communication of software components. A contract entails mainly preconditions and postconditions, but in some cases, invariants, and side-effects are also described:

- **Preconditions**: We can say that these are all the checks code will do before running. It will check for all the conditions that have to be made before the function can proceed. In general, it's implemented by validating the data set provided in the parameters passed, but nothing should stop us from running all sorts of validations (for example, validating a set in a database, a file, another method that was called before, and so on) if we consider that their side-effects are overshadowed by the importance of such a validation. Notice that this imposes a constraint on the caller.
- **Postconditions**: The opposite of preconditions, here, the validations are done after the function call is returned. Postcondition validations are run to validate what the caller is expecting from this component.
- **Invariants**: Optionally, it would be a good idea to document, in the docstring of a function, the invariants, the things that are kept constant while the code of the function is running, as an expression of the logic of the function to be correct.
- **Side-effects**: Optionally, we can mention any side-effects of our code in the docstring.

While conceptually all of these items form part of the contract for a software component, and this is what should go to the documentation of such piece, only the first two (preconditions and postconditions) are to be enforced at a low level (code).

The reason why we would design by contract is that if errors occur, they must be easy to spot (and by noticing whether it was either the precondition or postcondition that failed, we will find the culprit much easily) so that they can be quickly corrected. More importantly, we want critical parts of the code to avoid being executed under the wrong assumptions. This should help to clearly mark the limits for the responsibilities and errors if they occur, as opposed to something saying—this part of the application is failing… But the caller code provided the wrong arguments, so where should we apply the fix?

The idea is that preconditions bind the client (they have an obligation to meet them if they want to run some part of the code), whereas postconditions bind the component in question to some guarantees that the client can verify and enforce.

This way, we can quickly identify responsibilities. If the precondition fails, we know it is due to a defect on the client. On the other hand, if the postcondition check fails, we know the problem is in the routine or class (supplier) itself.

Specifically regarding preconditions, it is important to highlight that they can be checked at runtime, and if they occur, the code that is being called should not be run at all (it does not make sense to run it because its conditions do not hold, and further more, doing so might end up making things worse).

3.1.1 1.1. Preconditions

Preconditions are all of the guarantees a function or method expects to receive in order to work correctly. In general programming terms, this usually means to provide data that is properly formed, for example, objects that are initialized, non-null values, and many more. For Python, in particular, being dynamically typed, this also means that sometimes we need to check for the exact type of data that is provided. This is not exactly the same as type checking, the kind mypy would do this, but rather verify for exact values that are needed.

Part of these checks can be detected early on by using static analysis tools, such as mypy, but these checks are not enough. A function should have proper validation for the information that it is going to handle.

Now, this poses the question of where to place the validation logic, depending on whether we let the clients validate all the data before calling the function, or allow this one to validate everything that it received prior running its own logic. The former corresponds to a tolerant approach (because the function itself is still allowing any data, potentially malformed data as well), whereas the latter corresponds to a demanding approach.

For the purposes of this analysis, we prefer a demanding approach when it comes to DbC, because it is usually the safest choice in terms of robustness, and usually the most common practice in the industry.

Regardless of the approach we decide to take, we should always keep in mind the non-redundancy principle, which states that the enforcement of each precondition for a function should be done by only one of the two parts of the contract, but not both. This means that we put the validation logic on the client, or we leave it to the function itself, but in no cases should we duplicate it (which also relates to the DRY principle).

3.1.2 1.2. Postconditions

Postconditions are the part of the contract that is responsible for enforcing the state after the method or function has returned.

Assuming that the function or method has been called with the correct properties (that is, with its preconditions met), then the postconditions will guarantee that certain properties are preserved.

The idea is to use postconditions to check and validate for everything that a client might need. If the method executed properly, and the postcondition validations pass, then any client calling that code should be able to work with the returned object without problems, as the contract has been fulfilled.

3.1.3 1.3. Pythonic contracts

Programming by Contract for Python, is deferred. This doesn't mean that we cannot implement it in Python, because, as introduced at the beginningf, this is a general design principle.

Probably the best way to enforce this is by adding control mechanisms to our methods, functions, and classes, and if they fail raise a RuntimeError exception or ValueError . It's hard to devise a general rule for the correct type of exception, as that would pretty much depend on the application in particular. These previously mentioned exceptions are the most common types of exception, but if they don't fit accurately with the problem, creating a custom exception would be the best choice.

We would also like to keep the code as isolated as possible. That is, the code for the preconditions in one part, the one for the postconditions in another, and the core of the function separated. We could achieve this separation by creating smaller functions, but in some cases implementing a decorator would be an interesting alternative.

3.1.4 1.4. Conclusions

The main value of this design principle is to effectively identify where the problem is. By defining a contract, when something fails at runtime it will be clear what part of the code is broken, and what broke the contract.

As a result of following this principle, the code will be more robust. Each component is enforcing its own constraints and maintaining some invariants, and the program can be proven correct as long as these invariants are preserved.

It also serves the purpose of clarifying the structure of the program better. Instead of trying to run ad hoc validations, or trying to surmount all possible failure scenarios, the contracts explicitly specify what each function or method expects to work properly, and what is expected from them.

Of course, following these principles also adds extra work, because we are not just programming the core logic of our main application, but also the contracts. In addition, we might also want to consider adding unit tests for these contracts as well. However, the quality gained by this approach pays off in the long run; hence, it is a good idea to implement this principle for critical components of the application.

Nonetheless, for this method to be effective, we should carefully think about what are we willing to validate, and this has to be a meaningful value. For example, it would not make much sense to define contracts that only check for the correct data types of the parameters provided to a function. Many programmers would argue that this would be like trying to make Python a statically-typed language. Regardless of this, tools such as Mypy, in combination with the use of annotations, would serve this purpose much better and with less effort. With that in mind, design contracts so that there is actually value on them.

3.2 2. Defensive programming

Defensive programming follows a somewhat different approach than DbC; instead of stating all conditions that must be held in a contract, that if unmet will raise an exception and make the program fail, this is more about making all parts of the code (objects, functions, or methods) able to protect themselves against invalid inputs.

Defensive programming is a technique that has several aspects, and it is particularly useful if it is combined with other design principles (this means that the fact that it follows a different philosophy than DbC does not mean that it is a case of either one or the other—it could mean that they might complement each other).

The main ideas on the subject of defensive programming are how to handle errors for scenarios that we might expect to occur, and how to deal with errors that should never occur (when impossible conditions happen). The former will fall into error handling procedures, while the latter will be the case for assertions, both topics we will be exploring in the following sections.

3.2.1 2.1. Error handling

In our programs, we resort to error handling procedures for situations that we anticipate as prone to cause errors. This is usually the case for data input.

The idea behind error handling is to gracefully respond to these expected errors in an attempt to either continue our program execution or decide to fail if the error turns out to be insurmountable.

There are different approaches by which we can handle errors on our programs, but not all of them are always applicable. Some of these approaches are as follows:

- Value substitution
- Error logging
- Exception handling

2.1.1. Value substitution

In some scenarios, when there is an error and there is a risk of the software producing an incorrect value or failing entirely, we might be able to replace the result with another, safer value. We call this value substitution, since we are in fact replacing the actual erroneous result for a value that is to be considered non-disruptive (it could be a default, a well-known constant, a sentinel value, or simply something that does not affect the result at all, like returning zero in a case where the result is intended to be applied to a sum).

Value substitution is not always possible, however. This strategy has to be carefully chosen for cases where the substituted value is actually a safe option. Making this decision is a trade-off between robustness and correctness. A software program is robust when it does not fail, even in the presence of an erroneous scenario. But this is not correct either. This might not be acceptable for some kinds of software. If the application is critical, or the data being handled is too sensitive, this is not an option, since we cannot afford to provide users (or other parts of the application) with erroneous results. In these cases, we opt for correctness, rather than let the program explode when yielding the wrong results.

A slightly different, and safer, version of this decision is to use default values for data that is not provided. This can be the case for parts of the code that can work with a default behavior, for example, default values for environment variables that are not set, for missing entries in configuration files, or for parameters of functions. We can find examples of Python supporting this throughout different methods of its API, for example, dictionaries have a get method, whose (optional) second parameter allows you to indicate a default value:

```
>>> configuration = {"dbport": 5432}
>>> configuration.get("dbhost", "localhost")
'localhost'
>>> configuration.get("dbport")
5432
```

Environment variables have a similar API:

```
>>> import os
>>> os.getenv("DBHOST")
'localhost'
>>> os.getenv("DPORT", 5432)
5432
```

In both previous examples, if the second parameter is not provided, None will be returned, because it's the default value those functions are defined with. We can also define default values for the parameters of our own functions:

```
>>> def connect_database(host="localhost", port=5432):
...
        logger.info("connecting to database server at %s:%i", host, port)
```

In general, replacing missing parameters with default values is acceptable, but substituting erroneous data with legal close values is more dangerous and can mask some errors. Take this criterion into consideration when deciding on this approach.

2.1.2. Exception handling

In the presence of incorrect or missing input data, sometimes it is possible to correct the situation with some examples such as the ones mentioned in the previous section. In other cases, however, it is better to stop the program from continuing to run with the wrong data than to leave it computing under erroneous assumptions. In those cases, failing and notifying the caller that something is wrong is a good approach, and this is the case for a precondition that was violated, as we saw in DbC.

Nonetheless, erroneous input data is not the only possible way in which a function can go wrong. After all, functions are not just about passing data around; they also have side-effects and connect to external components.

It could be possible that a fault in a function call is due to a problem on one of these external components, and not in our function itself. If that is the case, our function should communicate this properly. This will make it easier to debug. The function should clearly and unambiguously notify the rest of the application about errors that cannot be ignored so that they can be addressed accordingly.

The mechanism for accomplishing this is an exception. It is important to emphasize that this is what exceptions should be used for—clearly announcing an exceptional situation, not altering the flow of the program according to business logic.

If the code tries to use exceptions to handle expected scenarios or business logic, the flow of the program will become harder to read. This will lead to a situation where exceptions are used as a sort of go-to statement, that (to make things worse) could span multiple levels on the call stack (up to caller functions), violating the encapsulation of the logic into its correct level of abstraction. The case could get even worse if these except blocks are mixing business logic with truly exceptional cases that the code is trying to defend against; in that case, it will be harder to distinguish between the core logic we have to maintain and errors to be handled.

Note: Do not use exceptions as a go-to mechanism for business logic. Raise exceptions when there is actually something wrong with the code that callers need to be aware of.

This last concept is an important one; exceptions are usually about notifying the caller about something that is amiss. This means that exceptions should be used carefully because they weaken encapsulation. The more exceptions a function has, the more the caller function will have to anticipate, therefore knowing about the function it is calling. And if a function raises too many exceptions, this means that is not so context-free, because every time we want to invoke it, we will have to keep all of its possible side-effects in mind.

This can be used as a heuristic to tell when a function is not cohesive enough and has too many responsibilities. If it raises too many exceptions, it could be a sign that it has to be broken down into multiple, smaller ones.

2.1.2.1. Handle exceptions at the right level of abstraction

Exceptions are also part of the principal functions that do one thing, and one thing only. The exception the function is handling (or raising) has to be consistent with the logic encapsulated on it.

In this example, we can see what we mean by mixing different levels of abstractions. Imagine an object that acts as a transport for some data in our application. It connects to an external component where the data is going to be sent upon decoding. In the following listing, we will focus on the deliver_event method:

```python
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:
```

(continues on next page)

(continued from previous page)

```
            self.connect()
            data = event.decode()
            self.send(data)
        except ConnectionError as e:
            logger.info(f"connection error detected: {e}")
            raise
        except ValueError as e:
            logger.error(f"{event} contains incorrect data: {e}")
            raise

    def connect(self):
        for _ in range(self.retry_n_times):
            try:
                self.connection = self._connector.connect()
            except ConnectionError as e:
                logger.info(f"{e}: attempting new connection in {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)
```

For our analysis, let's zoom in and focus on how the deliver_event() method handles exceptions.

What does `ValueError` have to do with `ConnectionError`? Not much. By looking at these two highly different types of error, we can get an idea of how responsibilities should be divided. The `ConnectionError` should be handled inside the connect method. This will allow a clear separation of behavior. For example, if this method needs to support retries, that would be a way of doing it. Conversely, `ValueError` belongs to the decode method of the event. With this new implementation, this method does not need to catch any exception: the exceptions it was worrying about before are either handled by internal methods or deliberately left to be raised.

We should separate these fragments into different methods or functions. For connection management, a small function should be enough. This function will be in charge of trying to establish the connection, catching exceptions (should they occur), and logging them accordingly:

```
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(f"{e}: attempting new connection in {retry_threshold}")
        time.sleep(retry_threshold)
    exc = ConnectionError(f"Couldn't connect after {retry_n_times} times")
    logger.exception(exc)
    raise exc
```

Then, we will call this function in our method. As for the `ValueError` exception on the event, we could separate it with a new object and do composition, but for this limited case it would be overkill, so just moving the logic to a separate method would be enough. With these two considerations in place, the new version of the method looks much more compact and easier to read:

```
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(f"{event} contains incorrect data: {e}")
            raise
```

2.1.2.2 Do not expose tracebacks

This is a security consideration. When dealing with exceptions, it might be acceptable to let them propagate if the error is too important, and maybe even let the program fail if this is the decision for that particular scenario and correctness was favored over robustness.

When there is an exception that denotes a problem, it's important to log in with as much detail as possible (including the traceback information, message, and all we can gather) so that the issue can be corrected efficiently. At the same time, we want to include as much detail as possible for ourselves: we definitely don't want any of this becoming visible to users.

In Python, tracebacks of exceptions contain very rich and useful debugging information. Unfortunately, this information is also very useful for attackers or malicious users who want to try and harm the application, not to mention that the leak would represent an important information disclosure, jeopardizing the intellectual property of your organization (parts of the code will be exposed).

If you choose to let exceptions propagate, make sure not to disclose any sensitive information. Also, if you have to notify users about a problem, choose generic messages (such as Something went wrong, or Page not found). This is a common technique used in web applications that display generic informative messages when an HTTP error occurs.

2.1.2.3 Avoid empty except blocks

This was even referred to as the most diabolical Python anti-pattern. While it is good to anticipate and defend our programs against some errors, being too defensive might lead to even worse problems. In particular, the only problem with being too defensive is that there is an empty except block that silently passes without doing anything.

Python is so flexible that it allows us to write code that can be faulty and yet, will not raise an error, like this:

```
try:
    process_data()
except:
    pass
```

The problem with this is that it will not fail, ever. Even when it should. It is also non-Pythonic if you remember from the zen of Python that errors should never pass silently.

If there is a true exception, this block of code will not fail, which might be what we wanted in the first place. But what if there is a defect? We need to know if there is an error in our logic to be able to correct it. Writing blocks such as this one will mask problems, making things harder to maintain.

There are two alternatives:

- Catch a more specific exception (not too broad, such as an Exception). In fact, some linting tools and IDEs will warn you in some cases when the code is handling too broad an exception.
- Do some actual error handling on the except block.

The best thing to do would be to apply both items simultaneously.

Handling a more specific exception (for example, AttributeError or KeyError) will make the program more maintainable because the reader will know what to expect, and can get an idea of the why of it. It will also leave other exceptions free to be raised, and if that happens, this probably means a bug, only this time it can be discovered.

Handling the exception itself can mean multiple things. In its simplest form, it could be just about logging the exception (make sure to use logger.exception or logger.error to provide the full context of what happened). Other alternatives could be to return a default value (substitution, only that in this case after detecting an error, not prior to causing it), or raising a different exception.

Note: If you choose to raise a different exception, to include the original exception that caused the problem, which leads us to the next point.

2.1.2.4. Include the original exception

As part of our error handling logic, we might decide to raise a different one, and maybe even change its message. If that is the case, it is recommended to include the original exception that led to that.

In Python 3, we can now use the raise <e> from <original_exception> syntax. When using this construction, the original traceback will be embedded into the new exception, and the original exception will be set in the `__cause__` attribute of the resulting one.

For example, if we desire to wrap default exceptions with custom ones internally to our project, we could still do that while including information about the root exception:

```python
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
```

Note: Always use the raise <e> from <o> syntax when changing the type of the exception.

3.2.2 2.2. Using assertions in Python

Assertions are to be used for situations that should never happen, so the expression on the assert statement has to mean an impossible condition. Should this condition happen, it means there is a defect in the software.

In contrast with the error handling approach, here there is (or should not be) a possibility of continuing the program. If such an error occurs, the program must stop. It makes sense to stop the program because, as commented before, we are in the presence of a defect, so there is no way to move forward by releasing a new version of the software that corrects this defect.

The idea of using assertions is to prevent the program from causing further damage if such an invalid scenario is presented. Sometimes, it is better to stop and let the program crash, rather than let it continue processing under the wrong assumptions.

For this reason, assertions should not be mixed with the business logic, or used as control flow mechanisms for the software. The following example is a bad idea:

```
try:
    assert condition.holds(), "Condition is not satisfied"
except AssertionError:
    alternative_procedure()
```

Note: Do not catch the AssertionError exception.

Make sure that the program terminates when an assertion fails.

Include a descriptive error message in the assertion statement and log the errors to make sure that you can properly debug and correct the problem later on.

Another important reason why the previous code is a bad idea is that besides catching AssertionError, the statement in the assertion is a function call. Function calls can have side-effects, and they aren't always repeatable (we don't know if calling condition.holds() again will yield the same result). Moreover, if we stop the debugger at that line, we might not be able to conveniently see the result that causes the error, and, again, even if we call that function again, we don't know if that was the offending value.

A better alternative requires fewer lines of code and provides more useful information:

```
result = condition.holds()
assert result > 0, "Error with {0}".format(result)
```

3.3 3. Separation of concerns

This is a design principle that is applied at multiple levels. It is not just about the low-level design (code), but it is also relevant at a higher level of abstraction, so it will come up later when we talk about architecture.

Different responsibilities should go into different components, layers, or modules of the application. Each part of the program should only be responsible for a part of the functionality (what we call its concerns) and should know nothing about the rest.

The goal of separating concerns in software is to enhance maintainability by minimizing ripple effects. A ripple effect means the propagation of a change in the software from a starting point. This could be the case of an error or exception triggering a chain of other exceptions, causing failures that will result in a defect on a remote part of the application. It can also be that we have to change a lot of code scattered through multiple parts of the code base, as a result of a simple change in a function definition.

Clearly, we do not want these scenarios to happen. The software has to be easy to change. If we have to modify or refactor some part of the code that has to have a minimal impact on the rest of the application, the way to achieve this is through proper encapsulation.

In a similar way, we want any potential errors to be contained so that they don't cause major damage.

This concept is related to the DbC principle in the sense that each concern can be enforced by a contract. When a contract is violated, and an exception is raised as a result of such a violation, we know what part of the program has the failure, and what responsibilities failed to be met.

Despite this similarity, separation of concerns goes further. We normally think of contracts between functions, methods, or classes, and while this also applies to responsibilities that have to be separated, the idea of separation of concerns also applies to Python modules, packages, and basically any software component.

3.3.1 3.1. Cohesion and coupling

These are important concepts for good software design.

On the one hand, cohesion means that objects should have a small and well-defined purpose, and they should do as little as possible. It follows a similar philosophy as Unix commands that do only one thing and do it well. The more cohesive our objects are, the more useful and reusable they become, making our design better.

On the other hand, coupling refers to the idea of how two or more objects depend on each other. This dependency poses a limitation. If two parts of the code (objects or methods) are too dependent on each other, they bring with them some undesired consequences:

- **No code reuse**: If one function depends too much on a particular object, or takes too many parameters, it's coupled with this object, which means that it will be really difficult to use that function in a different context (in order to do so, we will have to find a suitable parameter that complies with a very restrictive interface).
- **Ripple effects**: Changes in one of the two parts will certainly impact the other, as they are too close
- **Low level of abstraction**: When two functions are so closely related, it is hard to see them as different concerns resolving problems at different levels of abstraction

Note: Rule of thumb: Well-defined software will achieve high cohesion and low coupling.

3.4 4. Acronyms to live by

In this section, we will review some principles that yield some good design ideas. The point is to quickly relate to good software practices by acronyms that are easy to remember, working as a sort of mnemonic rule. If you keep these words in mind, you will be able to associate them with good practices more easily, and finding the right idea behind a particular line of code that you are looking at will be faster.

These are by no means formal or academic definitions, but more like empirical ideas that emerged from years of working in the software industry. Some of them do appear in books, as they were coined by important authors, and others have their roots probably in blog posts, papers, or conference talks.

3.4.1 4.1. DRY/OAOO

The ideas of **Don't Repeat Yourself (DRY) and Once and Only Once (OAOO)** are closely related, so they were included together here. They are self-explanatory, you should avoid duplication at all costs.

Things in the code, knowledge, have to be defined only once and in a single place. When you have to make a change in the code, there should be only one rightful location to modify. Failure to do so is a sign of a poorly designed system.

Code duplication is a problem that directly impacts maintainability. It is very undesirable to have code duplication because of its many negative consequences:

- **It's error prone**: When some logic is repeated multiple times throughout the code, and this needs to change, it means we depend on efficiently correcting all the instances with this logic, without forgetting of any of them, because in that case there will be a bug.

- **It's expensive**: Linked to the previous point, making a change in multiple places takes much more time (development and testing effort) than if it was defined only once. This will slow the team down.
- **It's unreliable**: Also linked to the first point, when multiple places need to be changed for a single change in the context, you rely on the person who wrote the code to remember all the instances where the modification has to be made. There is no single source of truth.

Duplication is often caused by ignoring (or forgetting) that code represents knowledge. By giving meaning to certain parts of the code, we are identifying and labeling that knowledge.

Let's see what this means with an example. Imagine that, in a study center, students are ranked by the following criteria: 11 points per exam passed, minus five points per exam failed, and minus two per year in the institution. The following is not actual code, but just a representation of how this might be scattered in a real code base:

```
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(f"Name: {student.name}, Score: {student.passed * 11 - student.failed␣
↪* 5 - student.years * 2}")
```

Notice how the lambda which is in the key of the sorted function represents some valid knowledge from the domain problem, yet it doesn't reflect it (it doesn't have a name, a proper and rightful location, there is no meaning assigned to that code, nothing). This lack of meaning in the code leads to the duplication we find when the score is printed out while listing the raking.

We should reflect our knowledge of our domain problem in our code, and our code will then be less likely to suffer from duplication and will be easier to understand:

```
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(f"Name: {student.name}, Score: {core_for_student(student)}")
```

A fair disclaimer: this is just an analysis of one of the traits of code duplication. In reality, there are more cases, types, and taxonomies of code duplication, entire chapters could be dedicated to this topic, but here we focus on one particular aspect to make the idea behind the acronym clear.

In this example, we have taken what is probably the simplest approach to eliminating duplication: creating a function. Depending on the case, the best solution would be different. In some cases, there might be an entirely new object that has to be created (maybe an entire abstraction was missing). In other cases, we can eliminate duplication with a context manager. Iterators or generators could also help to avoid repetition in the code, and decorators will also help.

Unfortunately, there is no general rule or pattern to tell you which of the features of Python are the most suitable to address code duplication, but hopefully, after seeing the examples, and how the elements of Python are used, you will be able to develop your own intuition.

3.4.2 4.2. YAGNI

YAGNI (short for You Ain't Gonna Need It) is an idea you might want to keep in mind very often when writing a solution if you do not want to over-engineer it.

We want to be able to easily modify our programs, so we want to make them future-proof. In line with that, many developers think that they have to anticipate all future requirements and create solutions that are very complex, and so create abstractions that are hard to read, maintain, and understand. Sometime later, it turns out that those anticipated requirements do not show up, or they do but in a different way (surprise!), and the original code that was supposed to handle precisely that does not work. The problem is that now it is even harder to refactor and extend our programs. What happened was that the original solution did not handle the original requirements correctly, and neither do the current ones, simply because it is the wrong abstraction.

Having maintainable software is not about anticipating future requirements. It is about writing software that only addresses current requirements in such a way that it will be possible (and easy) to change later on. In other words, when designing, make sure that your decisions don't tie you down, and that you will be able to keep on building, but do not build more than what's necessary.

3.4.3 4.3. KIS

KIS (stands for Keep It Simple) relates very much to the previous point. When you are designing a software component, avoid over-engineering it; ask yourself if your solution is the minimal one that fits the problem.

Implement minimal functionality that correctly solves the problem and does not complicate your solution more than is necessary. Remember: the simpler the design, the more maintainable it will be.

This design principle is an idea we will want to keep in mind at all levels of abstraction, whether we are thinking of a high-level design, or addressing a particular line of code.

At a high-level, think on the components we are creating. Do we really need all of them? Does this module a ctually require being utterly extensible right now? Emphasize the last part—maybe we want to make that component extensible, but now is not the right time, or it is not appropriate to do so because we still do not have enough information to create the proper abstractions, and trying to come up with generic interfaces at this point will only lead to even worse problems.

In terms of code, keeping it simple usually means using the smallest data structure that fits the problem. You will most likely find it in the standard library.

Sometimes, we might over-complicate code, creating more functions or methods than what's necessary. The following class creates a namespace from a set of keyword arguments that have been provided, but it has a rather complicated code interface:

```python
class ComplicatedNamespace:
    """An convoluted example of initializing an object with some
    properties."""

    ACCEPTED_VALUES = ("id_", "user", "location")

    @classmethod
    def init_with_data(cls, **data):
        instance = cls()
        for key, value in data.items():
            if key in cls.ACCEPTED_VALUES:
                setattr(instance, key, value)
        return instance
```

Having an extra class method for initializing the object doesn't seem really necessary. Then, the iteration, and the call to setattr inside it, make things even more strange, and the interface that is presented to the user is not very clear:

```
>>> cn = ComplicatedNamespace.init_with_data(
...
id_=42, user="root", location="127.0.0.1", extra="excluded"
... )
>>> cn.id_, cn.user, cn.location
(42, 'root', '127.0.0.1')
>>> hasattr(cn, "extra")
False
```

The user has to know of the existence of this other method, which is not convenient. It would be better to keep it simple, and just initialize the object as we initialize any other object in Python (after all, there is a method for that) with the `__init__` method:

```
class Namespace:
    """Create an object from keyword arguments."""

    ACCEPTED_VALUES = ("id_", "user", "location")

    def __init__(self, **data):
        accepted_data = {k: v for k, v in data.items() if k in self.ACCEPTED_
↪VALUES}
        self.__dict__.update(accepted_data)
```

Remember the zen of Python: simple is better than complex.

3.4.4 4.4. EAFP/LBYL

EAFP (stands for Easier to Ask Forgiveness than Permission), while LBYL (stands for Look Before You Leap).

The idea of EAFP is that we write our code so that it performs an action directly, and then we take care of the consequences later in case it doesn't work. Typically, this means try running some code, expecting it to work, but catching an exception if it doesn't, and then handling the corrective code on the except block.

This is the opposite of LBYL. As its name says, in the look before you leap approach, we first check what we are about to use. For example, we might want to check if a file is available before trying to operate with it:

```
if os.path.exists(filename):
    with open(filename) as f:
    ...
```

This might be good for other programming languages, but it is not the Pythonic way of writing code. Python was built with ideas such as EAFP, and it encourages you to follow them (remember, explicit is better than implicit). This code would instead be rewritten like this:

```
try:
with open(filename) as f:
    ...
except FileNotFoundError as e:
    logger.error(e)
```

Note: Prefer EAFP over LBYL.

3.5 5. Composition and inheritance

In object-oriented software design, there are often discussions as to how to address some problems by using the main ideas of the paradigm (polymorphism, inheritance, and encapsulation).

Probably the most commonly used of these ideas is inheritance—developers often start by creating a class hierarchy with the classes they are going to need and decide the methods each one should implement.

While inheritance is a powerful concept, it does come with its perils. The main one is that every time we extend a base class, we are creating a new one that is tightly coupled with the parent. As we have already discussed, coupling is one of the things we want to reduce to a minimum when designing software.

One of the main uses developers relate inheritance with is code reuse. While we should always embrace code reuse, it is not a good idea to force our design to use inheritance to reuse code just because we get the methods from the parent class for free. The proper way to reuse code is to have highly cohesive objects that can be easily composed and that could work on multiple contexts.

3.5.1 5.1. When inheritance is a good decision

We have to be careful when creating a derived class, because this is a double-edged sword—on the one hand, it has the advantage that we get all the code of the methods from the parent class for free, but on the other hand, we are carrying all of them to a new class, meaning that we might be placing too much functionality in a new definition.

When creating a new subclass, we have to think if it is actually going to use all of the methods it has just inherited, as a heuristic to see if the class is correctly defined. If instead, we find out that we do not need most of the methods, and have to override or replace them, this is a design mistake that could be caused by several reasons:

- The superclass is vaguely defined and contains too much responsibility, instead of a well-defined interface.
- The subclass is not a proper specialization of the superclass it is trying to extend.

A good case for using inheritance is the type of situation when you have a class that defines certain components with its behavior that are defined by the interface of this class (its public methods and attributes), and then you need to specialize this class in order to create objects that do the same but with something else added, or with some particular parts of its behavior changed.

You can find examples of good uses of inheritance in the Python standard library itself. For example, in the `http.server` package, we can find a base class such as `BaseHTTPRequestHandler`, and subclasses such as `SimpleHTTPRequestHandler` that extend this one by adding or changing part of its base interface.

Speaking of interface definition, this is another good use for inheritance. When we want to enforce the interface of some objects, we can create an abstract base class that does not implement the behavior itself, but instead just defines the interface—every class that extends this one will have to implement these to be a proper subtype.

Finally, another good case for inheritance is exceptions. We can see that the standard exception in Python derives from `Exception`. This is what allows you to have a generic clause such as `except Exception:`, which will catch every possible error. The important point is the conceptual one, they are classes derived from Exception because they are more specific exceptions. This also works in well-known libraries such as `requests`, for instance, in which an `HTTPError` is `RequestException`, which in turn is an `IOError`.

3.5.2 5.2. Anti-patterns for inheritance

If the previous section had to be summarized into a single word, it would be specialization. The correct use for inheritance is to specialize objects and create more detailed abstractions starting from base ones.

The parent (or base) class is part of the public definition of the new derived class. This is because the methods that are inherited will be part of the interface of this new class. For this reason, when we read the public methods of a class, they have to be consistent with what the parent class defines.

For example, if we see that a class derived from `BaseHTTPRequestHandler` implements a method named `handle()`, it would make sense because it is overriding one of the parents. If it had any other method whose name relates to an action that has to do with an HTTP request, then we could also think that is correctly placed (but we would not think that if we found something called process_purchase() on that class).

The previous illustration might seem obvious, but it is something that happens very often, especially when developers try to use inheritance with the sole goal of reusing code. In the next example, we will see a typical situation that represents a common anti-pattern in Python: there is a domain problem that has to be represented, and a suitable data structure is devised for that problem, but instead of creating an object that uses such a data structure, the object becomes the data structure itself.

Let's see these problems more concretely through an example. Imagine we have a system for managing insurance, with a module in charge of applying policies to different clients. We need to keep in memory a set of customers that are being processed at the time in order to apply those changes before further processing or persistence. The basic operations we need are to store a new customer with its records as satellite data, apply a change on a policy, or edit some of the data, just to name a few. We also need to support a batch operation, that is, when something on the policy itself changes (the one this module is currently processing), we have to apply these changes overall to customers on the current transaction.

Thinking in terms of the data structure we need, we realize that accessing the record for a particular customer in constant time is a nice trait. Therefore, something like `policy_transaction[customer_id]` looks like a nice interface. From this, we might think that a subscriptable object is a good idea, and further on, we might get carried away into thinking that the object we need is a dictionary:

```
class TransactionalPolicy(collections.UserDict):
    """Example of an incorrect use of inheritance."""

    def change_in_policy(self, customer_id, **new_policy_data):
        self[customer_id].update(**new_policy_data)
```

With this code, we can get information about a policy for a customer by its identifier:

```
>>> policy = TransactionalPolicy({
...
"client001": {
...
"fee": 1000.0,
...
"expiration_date": datetime(2020, 1, 3),
...
}
... })

>>> policy["client001"]
{'fee': 1000.0, 'expiration_date': datetime.datetime(2020, 1, 3, 0, 0)}

>>> policy.change_in_policy("client001", expiration_date=datetime(2020, 1,
4))

>>> policy["client001"]
{'fee': 1000.0, 'expiration_date': datetime.datetime(2020, 1, 4, 0, 0)}
```

Sure, we achieved the interface we wanted in the first place, but at what cost? Now, this class has a lot of extra behavior from carrying out methods that weren't necessary:

```
>>> dir(policy)
[ # all magic and special method have been omitted for brevity...
'change_in_policy', 'clear', 'copy', 'data', 'fromkeys', 'get', 'items',
'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

There are (at least) two major problems with this design. On the one hand, the hierarchy is wrong. Creating a new class from a base one conceptually means that it's a more specific version of the class it's extending (hence the name). How is it that a `TransactionalPolicy` is a dictionary? Does this make sense? Remember, this is part of the public interface of the object, so users will see this class, their hierarchy, and will notice such an odd specialization, as well as its public methods.

This leads us to the second problem—coupling. The interface of the transactional policy now includes all methods from a dictionary. Does a transactional policy really need methods such as `pop()` or `items()`? However, there they are. They are also public, so any user of this interface is entitled to call them, with whatever undesired side-effect they may carry. More on this point: we don't really gain much by extending a dictionary. The only method it actually needs to update for all customers affected by a change in the current policy (`change_in_policy()`) is not on the base class, so we will have to define it ourselves either way.

This is a problem of mixing implementation objects with domain objects. A dictionary is an implementation object, a data structure, suitable for certain kinds of operation, and with a trade-off like all data structures. A transactional policy should represent something in the domain problem, an entity that is part of the problem we are trying to solve.

Hierarchies like this one are incorrect, and just because we get a few magic methods from a base class (to make the object subscriptable by extending a dictionary) is not reason enough to create such an extension. Implementation classes should be extending solely when creating other, more specific, implementation classes. In other words, extend a dictionary if you want to create another (more specific, or slightly modified) dictionary. The same rule applies to classes of the domain problem.

The correct solution here is to use composition. `TransactionalPolicy` is not a dictionary: it uses a dictionary. It should store a dictionary in a private attribute, and implement __getitem__() by proxying from that dictionary and then only implementing the rest of the public method it requires:

```python
class TransactionalPolicy:
    """Example refactored to use composition."""

    def __init__(self, policy_data, **extra_data):
        self._data = {**policy_data, **extra_data}

    def change_in_policy(self, customer_id, **new_policy_data):
        self._data[customer_id].update(**new_policy_data)

    def __getitem__(self, customer_id):
        return self._data[customer_id]

    def __len__(self):
        return len(self._data)
```

This way is not only conceptually correct, but also more extensible. If the underlying data structure (which, for now, is a dictionary) is changed in the future, callers of this object will not be affected, so long as the interface is maintained. This reduces coupling, minimizes ripple effects, allows for better refactoring (unit tests ought not to be changed), and makes the code more maintainable.

3.5.3 5.3. Multiple inheritance in Python

Python supports multiple inheritance. As inheritance, when improperly used, leads to design problems, you could also expect that multiple inheritance will also yield even bigger problems when it's not correctly implemented.

Multiple inheritance is, therefore, a double-edged sword. It can also be very beneficial in some cases. Just to be clear, there is nothing wrong with multiple inheritance, the only problem it has is that when it's not implemented correctly, it will multiply the problems.

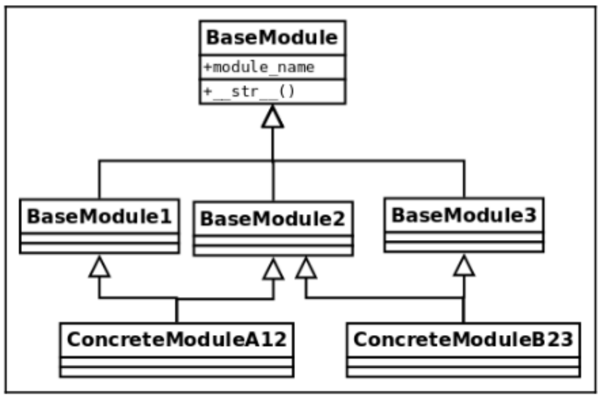
Multiple inheritance is a perfectly valid solution when used correctly, and this opens up new patterns (such as the adapter pattern) and mixins.

One of the most powerful applications of multiple inheritance is perhaps that which enables the creation of mixins. Before exploring mixins, we need to understand how multiple inheritance works, and how methods are resolved in a complex hierarchy.

5.3.1. Method Resolution Order (MRO)

Some people don't like multiple inheritance because of the constraints it has in other programming languages, for instance, the so-called diamond problem. When a class extends from two or more, and all of those classes also extend from other base classes, the bottom ones will have multiple ways to resolve the methods coming from the top-level classes. The question is, which of these implementations is used?

Consider the following diagram, which has a structure with multiple inheritance.



The top-level class has a class attribute and implements the `__str__` method. Think of any of the concrete classes, for example, `ConcreteModuleA12`: it extends from `BaseModule1` and `BaseModule2`, and each one of them will take the implementation of `__str__` from `BaseModule`. Which of these two methods is going to be the one for `ConcreteModuleA12`?

With the value of the class attribute, this will become evident:

```python
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(BaseModule2, BaseModule3):
    """Extend 2 & 3"""
```

Now, let's test this to see what method is being called:

```
>>> str(ConcreteModuleA12("test"))
'module-1:test'
```

There is no collision. Python resolves this by using an algorithm called C3 linearization or MRO, which defines a deterministic way in which methods are going to be called.

In fact, we can specifically ask the class for its resolution order:

```
>>> [cls.__name__ for cls in ConcreteModuleA12.mro()]
['ConcreteModuleA12', 'BaseModule1', 'BaseModule2', 'BaseModule', 'object']
```

Knowing about how the method is going to be resolved in a hierarchy can be used to our advantage when designing classes because we can make use of mixins.

5.3.2. Mixins

A mixin is a base class that encapsulates some common behavior with the goal of reusing code. Typically, a mixin class is not useful on its own, and extending this class alone will certainly not work, because most of the time it depends on methods and properties that are defined in other classes. The idea is to use mixin classes along with other ones, through multiple inheritance, so that the methods or properties used on the mixin will be available.

Imagine we have a simple parser that takes a string and provides iteration over it by its values separated by hyphens (-):

```
class BaseTokenizer:
    def __init__(self, str_token):
        self.str_token = str_token
    def __iter__(self):
    yield from self.str_token.split("-")
```

This is quite straightforward:

```
>>> tk = BaseTokenizer("28a2320b-fd3f-4627-9792-a2b38e3c46b0")
>>> list(tk)
['28a2320b', 'fd3f', '4627', '9792', 'a2b38e3c46b0']
```

But now we want the values to be sent in upper-case, without altering the base class. For this simple example, we could just create a new class, but imagine that a lot of classes are already extending from `BaseTokenizer`, and we don't want to replace all of them. We can mix a new class into the hierarchy that handles this transformation:

```
class UpperIterableMixin:
    def __iter__(self):
        return map(str.upper, super().__iter__())

class Tokenizer(UpperIterableMixin, BaseTokenizer):
    pass
```

The new `Tokenizer` class is really simple. It doesn't need any code because it takes advantage of the mixin. This type of mixing acts as a sort of decorator. Based on what we just saw, `Tokenizer` will take `__iter__` from the mixin, and this one, in turn, delegates to the next class on the line (by calling `super()`), which is the `BaseTokenizer`, but it converts its values to uppercase, creating the desired effect.

3.6 6. Arguments in functions and methods

In Python, functions can be defined to receive arguments in several different ways, and these arguments can also be provided by callers in multiple ways.

There is also an industry-wide set of practices for defining interfaces in software engineering that closely relates to the definition of arguments in functions.

3.6.1 6.1. How function arguments work in Python

First, we will explore the particularities of how arguments are passed to functions in Python, and then we will review the general theory of good software engineering practices that relate to these concepts.

By first understanding the possibilities that Python offers for handling parameters, we will be able to assimilate general rules more easily, and the idea is that after having done so, we can easily draw conclusions on what good patterns or idioms are when handling arguments. Then, we can identify in which scenarios the Pythonic approach is the correct one, and in which cases we might be abusing the features of the language.

6.1.1. How arguments are copied to functions

The first rule in Python is that all arguments are passed by a value. Always. This means that when passing values to functions, they are assigned to the variables on the signature definition of the function to be later used on it. You will notice that a function changing arguments might depend on the type arguments: if we are passing mutable objects, and the body of the function modifies this, then, of course, we have side-effect that they will have been changed by the time the function returns.

In the following we can see the difference:

```
>>> def function(argument):
...     argument += " in function"
...     print(argument)
...
>>> immutable = "hello"
>>> function(immutable)
hello in function
>>> mutable = list("hello")
>>> immutable
'hello'
>>> function(mutable)
['h', 'e', 'l', 'l', 'o', ' ', 'i', 'n', ' ', 'f', 'u', 'n', 'c', 't', 'i', 'o', 'n
↪']
>>> mutable
['h', 'e', 'l', 'l', 'o', ' ', 'i', 'n', ' ', 'f', 'u', 'n', 'c', 't', 'i', 'o', 'n
↪']
```

This might look like an inconsistency, but it's not. When we pass the first argument, a string, this is assigned to the argument on the function. Since string objects are immutable, a statement like `argument += <expression>` will in fact create the new object, `argument + <expression>` , and assign that back to the argument. At that point, an argument is just a local variable inside the scope of the function and has nothing to do with the original one in the caller.

On the other hand, when we pass list, which is a mutable object, then that statement has a different meaning (it's actually equivalent to calling `.extend()` on that list). This operator acts by modifying the list in-place over a variable that holds a reference to the original list object, hence modifying it.

We have to be careful when dealing with these types of parameter because it can lead to unexpected side-effects. Unless you are absolutely sure that it is correct to manipulate mutable arguments in this way, we would recommend avoiding it and going for alternatives without these problems.

Note: Don't mutate function arguments. In general, try to avoid side-effects in functions as much as possible.

Arguments in Python can be passed by position, as in many other programming languages, but also by keyword. This means that we can explicitly tell the function which values we want for which of its parameters. The only caveat is that after a parameter is passed by keyword, the rest that follow must also be passed this way, otherwise, SyntaxError will be raised.

6.1.2. Variable number of arguments

Python, as well as other languages, has built-in functions and constructions that can take a variable number of arguments. Consider for example string interpolation functions (whether it be by using the % operator or the format method for strings), which follow a similar structure to the printf function in C, a first positional parameter with the string format, followed by any number of arguments that will be placed on the markers of that formatting string.

Besides taking advantage of these functions that are available in Python, we can also create our own, which will work in a similar fashion. In this section, we will cover the basic principles of functions with a variable number of arguments, along with some recommendations, so that in the next section, we can explore how to use these features to our advantage when dealing with common problems, issues, and constraints that functions might have if they have too many arguments.

For a variable number of positional arguments, the star symbol (`*`) is used, preceding the name of the variable that is packing those arguments. This works through the packing mechanism of Python.

Let's say there is a function that takes three positional arguments. In one part of the code, we conveniently happen to have the arguments we want to pass to the function inside a list, in the same order as they are expected by the function. Instead of passing them one by one by the position (that is, `list[0]` to the first element, `list[1]` to the second, and so on), which would be really un-Pythonic, we can use the packing mechanism and pass them all together in a single instruction:

```
>>> def f(first, second, third):
...     print(first)
...     print(second)
...     print(third)
...
>>> l = [1, 2, 3]
>>> f(*l)
1
2
3
```

The nice thing about the packing mechanism is that it also works the other way around. If we want to extract the values of a list to variables, by their respective position, we can assign them like this:

```
>>> a, b, c = [1, 2, 3]
>>> a
1
>>> b
2
>>> c
3
```

Partial unpacking is also possible. Let's say we are just interested in the first values of a sequence (this can be a list, tuple, or something else), and after some point we just want the rest to be kept together. We can assign the variables we need and leave the rest under a packaged list. The order in which we unpack is not limited. If there is nothing to place in one of the unpacked subsections, the result will be an empty list:

```
>>> def show(e, rest):
...     print("Element: {0} - Rest: {1}".format(e, rest))
...
>>> first, *rest = [1, 2, 3, 4, 5]
>>> show(first, rest)
Element: 1 - Rest: [2, 3, 4, 5]
>>> *rest, last = range(6)
>>> show(last, rest)
Element: 5 - Rest: [0, 1, 2, 3, 4]
>>> first, *middle, last = range(6)
>>> first
0
```

(continues on next page)

(continued from previous page)

```
>>> middle
[1, 2, 3, 4]
>>> last
5
>>> first, last, *empty = (1, 2)
>>> first
1
>>> last
2
>>> empty
[]
```

One of the best uses for unpacking variables can be found in iteration. When we have to iterate over a sequence of elements, and each element is, in turn, a sequence, it is a good idea to unpack at the same time each element is being iterated over. To see an example of this in action, we are going to pretend that we have a function that receives a list of database rows, and that it is in charge of creating users out of that data. The first implementation takes the values to construct the user with from the position of each column in the row, which is not idiomatic at all. The second implementation uses unpacking while iterating:

```
USERS = [(i, f"first_name_{i}", "last_name_{i}") for i in range(1_000)]

class User:
    def __init__(self, user_id, first_name, last_name):
        self.user_id = user_id
        self.first_name = first_name
        self.last_name = last_name

    def bad_users_from_rows(dbrows) -> list:
        """A bad case (non-pythonic) of creating ``User``s from DB rows."""
        return [User(row[0], row[1], row[2]) for row in dbrows]

    def users_from_rows(dbrows) -> list:
    """Create ``User``s from DB rows."""
    return [User(user_id, first_name, last_name) for (user_id, first_name, last_
↪name) in dbrows]
```

Notice that the second version is much easier to read. In the first version of the function (`bad_users_from_rows`), we have data expressed in the form `row[0]` , `row[1]` , and `row[2]`, which doesn't tell us anything about what they are. On the other hand, variables such as `user_id`, `first_name` , and `last_name` speak for themselves.

We can leverage this kind of functionality to our advantage when designing our own functions.

An example of this that we can find in the standard library lies in the max function, which is defined as follows:

```
max(...)
max(iterable, *[, default=obj, key=func]) -> value
max(arg1, arg2, *args, *[, key=func]) -> value
```

With a single iterable argument, return its biggest item. The default keyword-only argument specifies an object to return if the provided iterable is empty.

With two or more arguments, return the largest argument.

There is a similar notation, with two stars (**) for keyword arguments. If we have a dictionary and we pass it with a double star to a function, what it will do is pick the keys as the name for the parameter, and pass the value for that key as the value for that parameter in that function.

For instance, check this out:

```
function(**{"key": "value"})
```

It is the same as the following:

```
function(key="value")
```

Conversely, if we define a function with a parameter starting with two-star symbols, the opposite will happen: keyword-provided parameters will be packed into a dictionary:

```
>>> def function(**kwargs):
...     print(kwargs)
...
>>> function(key="value")
{'key': 'value'}
```

3.6.2 6.2. The number of arguments in functions

Having functions or methods that take too many arguments is a sign of bad design (a code smell). Then, we propose ways of dealing with this issue.

The first alternative is a more general principle of software design: **reification** (creating a new object for all of those arguments that we are passing, which is probably the abstraction we are missing). Compacting multiple arguments into a new object is not a solution specific to Python, but rather something that we can apply in any programming language.

Another option would be to use the Python-specific features we saw in the previous section, making use of variable positional and keyword arguments to create functions that have a dynamic signature. While this might be a Pythonic way of proceeding, we have to be careful not to abuse the feature, because we might be creating something that is so dynamic that it is hard to maintain. In this case, we should take a look at the body of the function. Regardless of the signature, and whether the parameters seem to be correct, if the function is doing too many different things responding to the values of the parameters, then it is a sign that it has to be broken down into multiple smaller functions (remember, functions should do one thing, and one thing only!).

6.2.1. Function arguments and coupling

The more arguments a function signature has, the more likely this one is going to be tightly coupled with the caller function.

Let's say we have two functions, `f1`, and `f2`, and the latter takes five parameters. The more parameters `f2` takes, the more difficult it would be for anyone trying to call that function to gather all that information and pass it along so that it can work properly.

Now, `f1` seems to have all of this information because it can call it correctly. From this, we can derive two conclusions: first, `f2` is probably a leaky abstraction, which means that since `f1` knows everything that `f2` requires, it can pretty much figure out what it is doing internally and will be able to do it by itself. So, all in all, `f2` is not abstracting that much. Second, it looks like `f2` is only useful to `f1`, and it is hard to imagine using this function in a different context, making it harder to reuse.

When functions have a more general interface and are able to work with higher-level abstractions, they become more reusable.

This applies to all sort of functions and object methods, including the `__init__` method for classes. The presence of a method like this could generally (but not always) mean that a new higher-level abstraction should be passed instead, or that there is a missing object.

Note: If a function needs too many parameters to work properly, consider it a code smell.

In fact, this is such a design problem that static analysis tools will, by default, raise a warning about when they encounter such a case. When this happens, don't suppress the warning, refactor it instead.

6.2.2. Compact function signatures that take too many arguments

Suppose we find a function that requires too many parameters. We know that we cannot leave the code base like that, and a refactor is imperative. But, what are the options? Depending on the case, some of the following rules might apply. This is by no means extensive, but it does provide an idea of how to solve some scenarios that occur quite often.

Sometimes, there is an easy way to change parameters if we can see that most of them belong to a common object. For example, consider a function call like this one:

```
track_request(request.headers, request.ip_addr, request.request_id)
```

Now, the function might or might not take additional arguments, but something is really obvious here: all of the parameters depend upon `request`, so why not pass the request object instead? This is a simple change, but it significantly improves the code. The correct function call should be `track_request(request)`: not to mention that, semantically, it also makes much more sense.

While passing around parameters like this is encouraged, in all cases where we pass mutable objects to functions, we must be really careful about side-effects. The function we are calling should not make any modifications to the object we are passing because that will mutate the object, creating an undesired side-effect. Unless this is actually the desired effect (in which case, it must be made explicit), this kind of behavior is discouraged. Even when we actually want to change something on the object we are dealing with, a better alternative would be to copy it and return a (new) modified version of it.

Note: Work with immutable objects, and avoid side-effects as much as possible.

This brings us to a similar topic: grouping parameters. In the previous example, the parameters were already grouped, but the group (in this case, the request object) was not being used. But other cases are not as obvious as that one, and we might want to group all the data in the parameters in a single object that acts as a container. Needless to say, this grouping has to make sense. The idea here is to reify: create the abstraction that was missing from our design.

If the previous strategies don't work, as a last resort we can change the signature of the function to accept a variable number of arguments. If the number of arguments is too big, using `*args` or `**kwargs` will make things harder to follow, so we have to make sure that the interface is properly documented and correctly used, but in some cases this is worth doing.

It's true that a function defined with `*args` and `**kwargs` is really flexible and adaptable, but the disadvantage is that it loses its signature, and with that, part of its meaning, and almost all of its legibility. We have seen examples of how names for variables (including function arguments) make the code much easier to read. If a function will take any number of arguments (positional or keyword), we might find out that when we want to take a look at that function in the future, we probably won't know exactly what it was supposed to do with its parameters, unless it has a very good docstring.

3.7 7. Final remarks on good practices for software design

A good software design involves a combination of following good practices of software engineering and taking advantage of most of the features of the language. There is a great value in using everything that Python has to offer, but there is also a great risk of abusing this and trying to fit complex features into simple designs.

In addition to this general principle, it would be good to add some final recommendations.

3.7.1 7.1. Orthogonality in software

This word is very general and can have multiple meanings or interpretations. In math, orthogonal means that two elements are independent. If two vectors are orthogonal, their scalar product is zero. It also means they are not related at all: a change in one of them doesn't affect the other one at all. That's the way we should think about our software.

Changing a module, class, or function should have no impact on the outside world to that component that is being modified. This is of course highly desirable, but not always possible. But even for cases where it's not possible, a good design will try to minimize the impact as much as possible. We have seen ideas such as separation of concerns, cohesion, and isolation of components.

In terms of the runtime structure of software, orthogonality can be interpreted as the fact that makes changes (or side-effects) local. This means, for instance, that calling a method on an object should not alter the internal state of other (unrelated) objects. We have already (and will continue to do so) emphasized in this book the importance of minimizing side-effects in our code.

In the example with the mixin class, we created a tokenizer object that returned an iterable. The fact that the `__iter__` method returned a new generator increases the chances that all three classes (the base, the mixing, and the concrete class) are orthogonal. If this had returned something in concrete (a list, let's say), this would have created a dependency on the rest of the classes, because when we changed the list to something else, we might have needed to update other parts of the code, revealing that the classes were not as independent as they should be.

Let's show you a quick example. Python allows passing functions by parameter because they are just regular objects. We can use this feature to achieve some orthogonality. We have a function that calculates a price, including taxes and discounts, but afterward we want to format the final price that's obtained:

```
def calculate_price(base_price: float, tax: float, discount: float) ->
    return (base_price * (1 + tax)) * (1 - discount)

def show_price(price: float) -> str:
    return "$ {0:,.2f}".format(price)

def str_final_price(base_price: float, tax: float, discount: float, fmt_
↪function=str) -> str:
    return fmt_function(calculate_price(base_price, tax, discount))
```

Notice that the top-level function is composing two orthogonal functions. One thing to notice is how we calculate the price, which is how the other one is going to be represented. Changing one does not change the other. If we don't pass anything in particular, it will use string conversion as the default representation function, and if we choose to pass a custom function, the resulting string will change. However, changes in `show_price` do not affect `calculate_price` . We can make changes to either function, knowing that the other one will remain as it was:

```
>>> str_final_price(10, 0.2, 0.5)
'6.0'
>>> str_final_price(1000, 0.2, 0)
'1200.0'
>>> str_final_price(1000, 0.2, 0.1, fmt_function=show_price)
'$ 1,080.00'
```

There is an interesting quality aspect that relates to orthogonality. If two parts of the code are orthogonal, it means one can change without affecting the other. This implies that the part that changed has unit tests that are also orthogonal to the unit tests of the rest of the application. Under this assumption, if those tests pass, we can assume (up to a certain degree) that the application is correct without needing full regression testing.

More broadly, orthogonality can be thought of in terms of features. Two functionalities of the application can be totally independent so that they can be tested and released without having to worry that one might break the other (or the rest of the code, for that matter).

Imagine that the project requires a new authentication mechanism (`oauth2`, let's say, but just for the sake of the example), and at the same time another team is also working on a new report. Unless there is something fundamentally wrong in that system, neither of those features should impact the other. Regardless of which one of those gets merged first, the other one should not be affected at all.

3.7.2 7.2. Structuring the code

The way code is organized also impacts the performance of the team and its maintainability.

In particular, having large files with lots of definitions (classes, functions, constants, and so on) is a bad practice and should be discouraged. This doesn't mean going to the extreme of placing one definition per file, but a good code base will structure and arrange components by similarity.

Luckily, most of the time, changing a large file into smaller ones is not a hard task in Python. Even if multiple other parts of the code depend on definitions made on that file, this can be broken down into a package, and will maintain total compatibility. The idea would be to create a new directory with a `__init__.py` file on it (this will make it a Python package). Alongside this file, we will have multiple files with all the particular definitions each one requires (fewer functions and classes grouped by a certain criterion). Then, the `__init__.py` file will import from all the other files the definitions it previously had (which is what guarantees its compatibility). Additionally, these definitions can be mentioned in the `__all__` variable of the module to make them exportable.

There are many advantages of this. Other than the fact that each file will be easier to navigate, and things will be easier to find, we could argue that it will be more efficient because of the following reasons:

- It contains fewer objects to parse and load into memory when the module is imported
- The module itself will probably be importing fewer modules because it needs fewer dependencies, like before

It also helps to have a convention for the project. For example, instead of placing constants in all of the files, we can create a file specific to the constant values to be used in the project, and import it from there: `from myproject.constants import CONNECTION_TIMEOUT`. Centralizing information like this makes it easier to reuse code and helps to avoid inadvertent duplication.

More details about separating modules and creating Python packages will be discussed in Chapter 10 , Clean Architecture, when we explore this in the context of software architecture.

CHAPTER

FOUR

THE SOLID PRINCIPLES

In case some of us aren't aware of what SOLID stands for, here it is:

- **S**: Single responsibility principle
- **O**: Open/closed principle
- **L**: Liskov's substitution principle
- **I**: Interface segregation principle
- **D**: Dependency inversion principle

4.1 1. Single responsibility principle

The **single responsibility principle (SRP)** states that a software component (in general, a class) must have only one responsibility. The fact that the class has a sole responsibility means that it is in charge of doing just one concrete thing, and as a consequence of that, we can conclude that it must have only one reason to change.

Only if one thing on the domain problem changes will the class have to be updated. If we have to make modifications to a class, for different reasons, it means the abstraction is incorrect, and that the class has too many responsibilities.T

This design principle helps us build more cohesive abstractions; objects that do one thing, and just one thing, well, following the Unix philosophy. What we want to avoid in all cases is having objects with multiple responsibilities (often called **god-objects**, because they know too much, or more than they should). These objects group different (mostly unrelated) behaviors, thus making them harder to maintain.

Again, the smaller the class, the better.

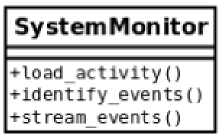
The SRP is closely related to the idea of cohesion in software design, which we already explored, when we discussed separation of concerns in software. What we strive to achieve here is that classes are designed in such a way that most of their properties and their attributes are used by its methods, most of the time. When this happens, we know they are related concepts, and therefore it makes sense to group them under the same abstraction.

In a way, this idea is somehow similar to the concept of normalization on relational database design. When we detect that there are partitions on the attributes or methods of the interface of an object, they might as well be moved somewhere else—it is a sign that they are two or more different abstractions mixed into one.

There is another way of looking at this principle. If, when looking at a class, we find methods that are mutually exclusive and do not relate to each other, they are the different responsibilities that have to be broken down into smaller classes.

4.1.1 1.1. A class with too many responsibilities

In this example, we are going to create the case for an application that is in charge of reading information about events from a source (this could be log files, a database, or many more sources), and identifying the actions corresponding to each particular log. A design that fails to conform to the SRP would look like this:



Without considering the implementation, the code for the class might look in the following listing:

```python
class SystemMonitor:

    def load_activity(self):
    """Get the events from a source, to be processed."""

    def identify_events(self):
    """Parse the source raw data into events (domain objects)."""

    def stream_events(self):
    """Send the parsed events to an external agent."""
```

The problem with this class is that it defines an interface with a set of methods that correspond to actions that are orthogonal: each one can be done independently of the rest.

This design flaw makes the class rigid, inflexible, and error-prone because it is hard to maintain. In this example, each method represents a responsibility of the class. Each responsibility entails a reason why the class might need to be modified. In this case, each method represents one of the various reasons why the class will have to be modified.

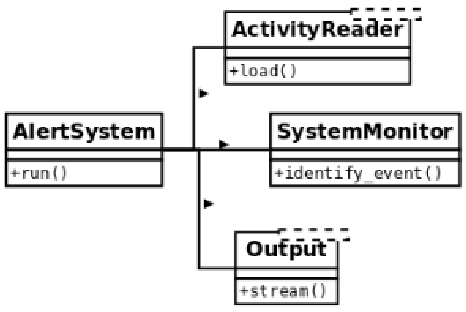
Consider the loader method, which retrieves the information from a particular source. Regardless of how this is done (we can abstract the implementation details here), it is clear that it will have its own sequence of steps, for instance connecting to the data source, loading the data, parsing it into the expected format, and so on. If any of this changes (for example, we want to change the data structure used for holding the data), the SystemMonitor class will need to change. Ask yourself whether this makes sense. Does a system monitor object have to change because we changed the representation of the data? No.

The same reasoning applies to the other two methods. If we change how we fingerprint events, or how we deliver them to another data source, we will end up making changes to the same class.

It should be clear by now that this class is rather fragile, and not very maintainable. There are lots of different reasons that will impact on changes in this class. Instead, we want external factors to impact our code as little as possible. The solution, again, is to create smaller and more cohesive abstractions.

4.1.2 1.2. Distributing responsibilities

To make the solution more maintainable, we separate every method into a different class. This way, each class will have a single responsibility:



The same behavior is achieved by using an object that will interact with instances of these new classes, using those objects as collaborators, but the idea remains that each class encapsulates a specific set of methods that are independent of the rest. The idea now is that changes on any of these classes do not impact the rest, and all of them have a clear and specific meaning. If we need to change something on how we load events from the data sources, the alert system is not even aware of these changes, so we do not have to modify anything on the system monitor (as long as the contract is still preserved), and the data target is also unmodified.

Changes are now local, the impact is minimal, and each class is easier to maintain.

The new classes define interfaces that are not only more maintainable but also reusable. Imagine that now, in another part of the application, we also need to read the activity from the logs, but for different purposes. With this design, we can simply use objects of type `ActivityReader` (which would actually be an interface, but for the purposes of this section, that detail is not relevant and will be explained later for the next principles). This would make sense, whereas it would not have made sense in the previous design, because attempts to reuse the only class we had defined would have also carried extra methods (such as `identify_events()`, or `stream_events()`) that were not needed at all.

One important clarification is that the principle does not mean at all that each class must have a single method. Any of the new classes might have extra methods, as long as they correspond to the same logic that that class is in charge of handling.

4.2 2. The open/closed principle

The **open/closed principle (OCP)** states that a module should be both open and closed (but with respect to different aspects).

When designing a class, for instance, we should carefully encapsulate the logic so that it has good maintenance, meaning that we will want it to be **open to extension but closed for modification**.

What this means in simple terms is that, of course, we want our code to be extensible, to adapt to new requirements, or changes in the domain problem. This means that, when something new appears on the domain problem, we only want to add new things to our model, not change anything existing that is closed to modification.

If, for some reason, when something new has to be added, we found ourselves modifying the code, then that logic is probably poorly designed. Ideally, when requirements change, we want to just have to extend the module with the new required behavior in order to comply with the new requirements, but without having to modify the code.

This principle applies to several software abstractions. It could be a class or even a module. In the following two subsections, we will see examples of each one, respectively.

4.2.1 2.1. Example of maintainability perils for not following the open/closed principle

Let's begin with an example of a system that is designed in such a way that does not follow the open/closed principle, in order to see the maintainability problems this carries, and the inflexibility of such a design.

The idea is that we have a part of the system that is in charge of identifying events as they occur in another system, which is being monitored. At each point, we want this component to identify the type of event, correctly, according to the values of the data that was previously gathered (for simplicity, we will assume it is packaged into a dictionary, and was previously retrieved through another means such as logs, queries, and many more). We have a class that, based on this data, will retrieve the event, which is another type with its own hierarchy.

A first attempt to solve this problem might look like this:

```python
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)
```

The following is the expected behavior of the preceding code:

```python
>>> 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'
```

We can clearly notice the hierarchy of event types, and some business logic to construct them. For instance, when there was no previous flag for a session, but there is now, we identify that record as a login event. Conversely, when the opposite happens, it means that it was a logout event. If it was not possible to identify an event, an event of type unknown is returned. This is to preserve polymorphism by following the null object pattern (instead of returning `None`, it retrieves an object of the corresponding type with some default logic).

This design has some problems. The first issue is that the logic for determining the types of events is centralized inside a monolithic method. As the number of events we want to support grows, this method will as well, and it could end up being a very long method, which is bad because, as we have already discussed, it will not be doing just one thing and one thing well.

On the same line, we can see that this method is not closed for modification. Every time we want to add a new type of event to the system, we will have to change something in this method (not to mention, that the chain of `elif` statements will be a nightmare to read!).

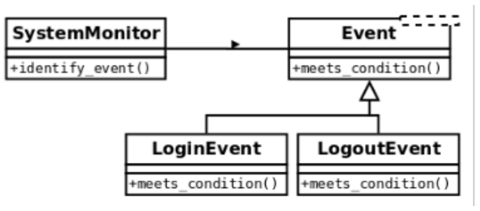
We want to be able to add new types of event without having to change this method (closed for modification). We also want to be able to support new types of event (open for extension) so that when a new event is added, we only have to add code, not change the code that already exists.

4.2.2 2.2. Refactoring the events system for extensibility

The problem with the previous example was that the SystemMonitor class was interacting directly with the concrete classes it was going to retrieve.

In order to achieve a design that honors the open/closed principle, we have to design toward abstractions.

A possible alternative would be to think of this class as it collaborates with the events, and then we delegate the logic for each particular type of event to its corresponding class:



Then we have to add a new (polymorphic) method to each type of event with the single responsibility of determining if it corresponds to the data being passed or not, and we also have to change the logic to go through all events, finding the right one.

The new code should look like this:

```
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:
```

(continues on next page)

(continued from previous page)

```
    """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)
```

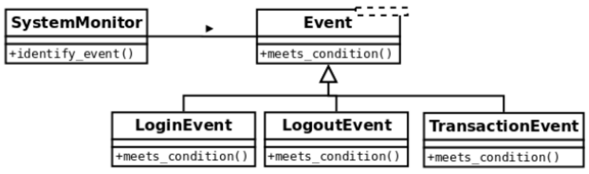
Notice how the interaction is now oriented toward an abstraction (in this case, it would be the generic base class `Event`, which might even be an abstract base class or an interface, but for the purposes of this example it is enough to have a concrete base class). The method no longer works with specific types of event, but just with generic events that follow a common interface—they are all polymorphic with respect to the `meets_condition` method.

Notice how events are discovered through the `__subclasses__()` method. Supporting new types of event is now just about creating a new class for that event that has to inherit from `Event` and implement its own `meets_condition()` method, according to its specific business logic.

4.2.3 2.3. Extending the events system

Now, let's prove that this design is actually as extensible as we wanted it to be. Imagine that a new requirement arises, and we have to also support events that correspond to transactions that the user executed on the monitored system.

The class diagram for the design has to include such a new event type, as in the following:



Only by adding the code to this new class does the logic keep working as expected:

```
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)
```

(continues on next page)

(continued from previous page)

```
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)
```

We can verify that the previous cases work as before and that the new event is also correctly identified:

```
>>> 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'
```

Notice that the `SystemMonitor.identify_event()` method did not change at all when we added the new event type. We, therefore, say that this method is closed with respect to new types of event.

Conversely, the `Event` class allowed us to add a new type of event when we were required to do so. We then say that events are open for an extension with respect to new types.

This is the true essence of this principle—when something new appears on the domain problem, we only want to add new code, not modify existing code.

4.2.4 2.4. Final thoughts about the OCP

As you might have noticed, this principle is closely related to effective use of polymorphism. We want to design toward abstractions that respect a polymorphic contract that the client can use, to a structure that is generic enough that extending the model is possible, as long as the polymorphic relationship is preserved.

This principle tackles an important problem in software engineering: maintainability. The perils of not following the OCP are ripple effects and problems in the software where a single change triggers changes all over the code base, or risks breaking other parts of the code.

One important final note is that, in order to achieve this design in which we do not change the code to extend behavior, we need to be able to create proper closure against the abstractions we want to protect (in this example, new types of event). This is not always possible in all programs, as some abstractions might collide (for example, we might have a proper abstraction that provides closure against a requirement, but does not work for other types of requirements). In these cases, we need to be selective and apply a strategy that provides the best closure for the types of requirement that require to be the most extensible.

4.3 3. Liskov's substitution principle

Liskov's substitution principle (LSP) states that there is a series of properties that an object type must hold to preserve reliability on its design.

The main idea behind LSP is that, for any class, a client should be able to use any of its subtypes indistinguishably, without even noticing, and therefore without compromising the expected behavior at runtime. This means that clients are completely isolated and unaware of changes in the class hierarchy.

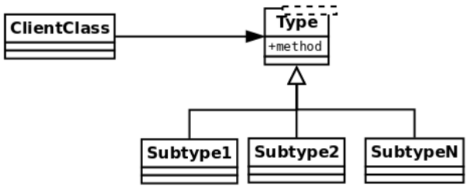
More formally, this is the original definition (LISKOV 01) of Liskov's substitution principle:

```
if S is a subtype of T, then objects of type T may be replaced by objects of type␣
↪S, without breaking the program.
```

This can be understood with the help of a generic diagram such as the following one.

Imagine that there is some client class that requires (includes) objects of another type. Generally speaking, we will want this client to interact with objects of some type, namely, it will work through an interface.

Now, this type might as well be just a generic interface definition, an abstract class or an interface, not a class with the behavior itself. There may be several subclasses extending this type (described in the diagram with the name Subtype, up to N). The idea behind this principle is that, if the hierarchy is correctly implemented, the client class has to be able to work with instances of any of the subclasses without even noticing. These objects should be interchangeable, as shown here:



This is related to other design principles we have already visited, like designing to interfaces. A good class must define a clear and concise interface, and as long as subclasses honor that interface, the program will remain correct.

As a consequence of this, the principle also relates to the ideas behind designing by contract. There is a contract between a given type and a client. By following the rules of LSP, the design will make sure that subclasses respect the contracts as they are defined by parent classes.

There are some scenarios so notoriously wrong with respect to the LSP that they can be easily identified.

4.3.1 3.1. Detecting incorrect datatypes in method signatures

By using type annotations throughout our code, we can quickly detect some basic errors early and check basic compliance with LSP.

One common code smell is that one of the subclasses of the parent class were to override a method in an incompatible fashion:

```
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)
```

The violation to LSP is clear—since the derived class is using a type for the `event_data` parameter which is different from the one defined on the base class, we cannot expect them to work equally. Remember that, according to this principle, any caller of this hierarchy has to be able to work with `Event` or `LoginEvent` transparently, without noticing any difference. Interchanging objects of these two types should not make the application fail. Failure to do so would break the polymorphism on the hierarchy.

The same error would have occurred if the return type was changed for something other than a Boolean value. The rationale is that clients of this code are expecting a Boolean value to work with. If one of the derived classes changes this return type, it would be breaking the contract, and again, we cannot expect the program to continue working normally.

A quick note about types that are not the same but share a common interface: even though this is just a simple example to demonstrate the error, it is still true that both dictionaries and lists have something in common; they are both iterables. This means that in some cases, it might be valid to have a method that expects a dictionary and another one expecting to receive a list, as long as both treat the parameters through the iterable interface. In this case, the problem would not lie in the logic itself (LSP might still apply), but in the definition of the types of the signature, which should read neither `list` nor `dict`, but a union of both. Regardless of the case, something has to be modified, whether it is the code of the method, the entire design, or just the type annotations.

Another strong violation of LSP is when, instead of varying the types of the parameters on the hierarchy, the signatures of the methods differ completely. This might seem like quite a blunder, but detecting it would not always be so easy to remember; Python is interpreted, so there is no compiler to detect these type of error early on, and therefore they will not be caught until runtime.

In the presence of a class that breaks the compatibility defined by the hierarchy (for example, by changing the signature of the method, adding an extra parameter, and so on) shown as follows:

```
class LogoutEvent(Event):
    def meets_condition(self, event_data: dict, override: bool) -> bool:
        if override:
            return True
```

4.3.2 3.2. More subtle cases of LSP violations

Cases where contracts are modified are particularly harder to detect. Given that the entire idea of LSP is that subclasses can be used by clients just like their parent class, it must also be true that contracts are correctly preserved on the hierarchy.

Remember that, when designing by contract, the contract between the client and supplier sets some rules: the client must provide the preconditions to the method, which the supplier might validate, and it returns some result to the client that it will check in the form of postconditions.

The parent class defines a contract with its clients. Subclasses of this one must respect such a contract. This means that, for example:

- A subclass can never make preconditions stricter than they are defined on the parent class
- A subclass can never make postconditions weaker than they are defined on the parent class

Consider the example of the events hierarchy defined in the previous section, but now with a change to illustrate the relationship between LSP and DbC.

This time, we are going to assume a precondition for the method that checks the criteria based on the data, that the provided parameter must be a dictionary that contains both keys "before" and "after" , and that their values are also nested dictionaries. This allows us to encapsulate even further, because now the client does not need to catch the `KeyError` exception, but instead just calls the precondition method (assuming that is acceptable to fail if the system is operating under the wrong assumptions). As a side note, it is good that we can remove this from the client, as now, `SystemMonitor` does not require to know which types of exceptions the methods of the collaborator class might raise (remember that exception weaken encapsulation, as they require the caller to know something extra about the object they are calling).

Such a design might be represented with the following changes in the code:

```
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)
```

And now the code that tries to detect the correct event type just checks the precondition once, and proceeds to find the right type of event:

```
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)
```

The contract only states that the top-level keys "before" and "after" are mandatory and that their values should also be dictionaries. Any attempt in the subclasses to demand a more restrictive parameter will fail.

The class for the transaction event was originally correctly designed. Look at how the code does not impose a restriction on the internal key named "transaction"; it only uses its value if it is there, but this is not mandatory:

```
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
```

However, the original two methods are not correct, because they demand the presence of a key named "session", which is not part of the original contract. This breaks the contract, and now the client cannot use these classes in the same way it uses the rest of them because it will raise `KeyError`.

After fixing this (changing the square brackets for the .get() method), the order on the LSP has been reestablished, and polymorphism prevails:

```
>>> 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'
```

We have to be careful when designing classes that we do not accidentally change the input or output of the methods in a way that would be incompatible with what the clients are originally expecting.

4.3.3 3.3. Remarks on the LSP

The LSP is fundamental to a good object-oriented software design because it emphasizes one of its core traits—polymorphism. It is about creating correct hierarchies so that classes derived from a base one are polymorphic along the parent one, with respect to the methods on their interface.

It is also interesting to notice how this principle relates to the previous one—if we attempt to extend a class with a new one that is incompatible, it will fail, the contract with the client will be broken, and as a result such an extension will not be possible (or, to make it possible, we would have to break the other end of the principle and modify code in the client that should be closed for modification, which is completely undesirable and unacceptable).

Carefully thinking about new classes in the way that LSP suggests helps us to extend the hierarchy correctly. We could then say that LSP contributes to the OCP.

4.4 4. Interface segregation

The **interface segregation principle (ISP)** provides some guidelines over an idea that we have revisited quite repeatedly already: that interfaces should be small.

In object-oriented terms, an interface is represented by the set of methods an object exposes. This is to say that all the messages that an object is able to receive or interpret constitute its interface, and this is what other clients can request. The interface separates the definition of the exposed behavior for a class from its implementation.

In Python, interfaces are implicitly defined by a class according to its methods. This is because Python follows the so-called **duck typing** principle.

Traditionally, the idea behind duck typing was that any object is really represented by the methods it has, and by what it is capable of doing. This means that, regardless of the type of the class, its name, its docstring, class attributes, or instance attributes, what ultimately defines the essence of the object are the methods it has. The methods defined on a class (what it knows how to do) are what determines what that object will actually be. It was called duck typing because of the idea that "If it walks like a duck, and quacks like a duck, it must be a duck."

For a long time, duck typing was the sole way interfaces were defined in Python. Later on, Python 3 (PEP-3119) introduced the concept of abstract base classes as a way to define interfaces in a different way. The basic idea of abstract base classes is that they define a basic behavior or interface that some derived classes are responsible for implementing. This is useful in situations where we want to make sure that certain critical methods are actually overridden, and it also works as a mechanism for overriding or extending the functionality of methods such as `isinstance()`.

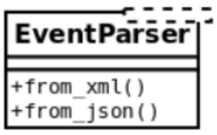
This module also contains a way of registering some types as part of a hierarchy, in what is called a **virtual subclass**. The idea is that this extends the concept of duck typing a little bit further by adding a new criterion—walks like a duck, quacks like a duck, or. . . it says it is a duck.

These notions of how Python interprets interfaces are important for understanding this principle and the next one.

In abstract terms, this means that the ISP states that, when we define an interface that provides multiple methods, it is better to instead break it down into multiple ones, each one containing fewer methods (preferably just one), with a very specific and accurate scope. By separating interfaces into the smallest possible units, to favor code reusability, each class that wants to implement one of these interfaces will most likely be highly cohesive given that it has a quite definite behavior and set of responsibilities.

4.4.1 4.1. An interface that provides too much

Now, we want to be able to parse an event from several data sources, in different formats (XML and JSON, for instance). Following good practice, we decide to target an interface as our dependency instead of a concrete class, and something like the following is devised:

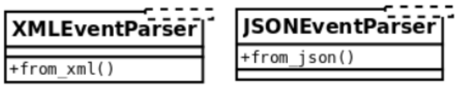


In order to create this as an interface in Python, we would use an abstract base class and define the methods (`from_xml()` and `from_json()`) as abstract, to force derived classes to implement them. Events that derive from this abstract base class and implement these methods would be able to work with their corresponding types.

But what if a particular class does not need the XML method, and can only be constructed from a JSON? It would still carry the from_xml() method from the interface, and since it does not need it, it will have to pass. This is not very flexible as it creates coupling and forces clients of the interface to work with methods that they do not need.

4.4.2 4.2. The smaller the interface, the better

It would be better to separate this into two different interfaces, one for each method:



With this design, objects that derive from `XMLEventParser` and implement the `from_xml()` method will know how to be constructed from an XML, and the same for a JSON file, but most importantly, we maintain the orthogonality of two independent functions, and preserve the flexibility of the system without losing any functionality that can still be achieved by composing new smaller objects.

There is some resemblance to the SRP, but the main difference is that here we are talking about interfaces, so it is an abstract definition of behavior. There is no reason to change because there is nothing there until the interface is actually implemented. However, failure to comply with this principle will create an interface that will be coupled with orthogonal functionality, and this derived class will also fail to comply with the SRP (it will have more than one reason to change).

4.4.3 4.3. How small should an interface be?

The point made in the previous section is valid, but it also needs a warning: avoid a dangerous path if it's misunderstood or taken to the extreme.

A base class (abstract or not) defines an interface for all the other classes to extend it. The fact that this should be as small as possible has to be understood in terms of cohesion: it should do one thing. That doesn't mean it must necessarily have one method. In the previous example, it was by coincidence that both methods were doing totally disjoint things, hence it made sense to separate them into different classes.

But it could be the case that more than one method rightfully belongs to the same class. Imagine that you want to provide a mixin class that abstracts certain logic in a context manager so that all classes derived from that mixin gain that context manager logic for free. As we already know, a context manager entails two methods: `__enter__` and `__exit__`. They must go together, or the outcome will not be a valid context manager at all!

Failure to place both methods in the same class will result in a broken component that is not only useless, but also misleadingly dangerous. Hopefully, this exaggerated example works as a counter-balance to the one in the previous section, and together the reader can get a more accurate picture about designing interfaces.

4.5 5. Dependency inversion

The **dependency inversion principle (DIP)** proposes an interesting design principle by which we protect our code by making it independent of things that are fragile, volatile, or out of our control. The idea of inverting dependencies is that our code should not adapt to details or concrete implementations, but rather the other way around: we want to force whatever implementation or detail to adapt to our code via a sort of API.

Abstractions have to be organized in such a way that they do not depend on details, but rather the other way around: the details (concrete implementations) should depend on abstractions.

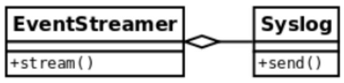
Imagine that two objects in our design need to collaborate, A and B. A works with an instance of B, but as it turns out, our module doesn't control B directly (it might be an external library, or a module maintained by another team, and so on). If our code heavily depends on B, when this changes the code will break. To prevent this, we have to invert the dependency: make B have to adapt to A. This is done by presenting an interface and forcing our code not to depend on the concrete implementation of B, but rather on the interface we have defined. It is then B's responsibility to comply with that interface.

In line with the concepts explored in previous sections, abstractions also come in the form of interfaces (or abstract base classes in Python).

In general, we could expect concrete implementations to change much more frequently than abstract components. It is for this reason that we place abstractions (interfaces) as flexibility points where we expect our system to change, be modified, or extended without the abstraction itself having to be changed.

4.5.1 5.1. A case of rigid dependencies

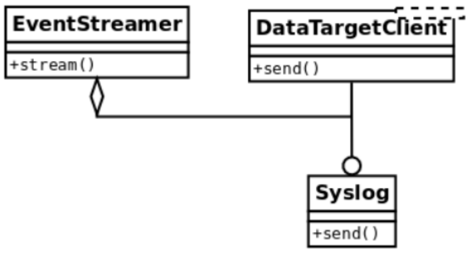
The last part of our event's monitoring system is to deliver the identified events to a data collector to be further analyzed. A naive implementation of such an idea would consist of having an event streamer class that interacts with a data destination, for example, `Syslog`:



However, this design is not very good, because we have a high-level class (`EventStreamer`) depending on a low-level one (`Syslog` is an implementation detail). If something changes in the way we want to send data to `Syslog`, `EventStreamer` will have to be modified. If we want to change the data destination for a different one or add new ones at runtime, we are also in trouble because we will find ourselves constantly modifying the `stream()` method to adapt it to these requirements.

4.5.2 5.2. Inverting the dependencies

The solution to these problems is to make `EventStreamer` work with an interface, rather than a concrete class. This way, implementing this interface is up to the low-level classes that contain the implementation details:



Now there is an interface that represents a generic data target where data is going to be sent to. Notice how the dependencies have now been inverted since `EventStreamer` does not depend on a concrete implementation of a particular data target, it does not have to change in line with changes on this one, and it is up to every particular data target; to implement the interface correctly and adapt to changes if necessary.

In other words, the original `EventStreamer` of the first implementation only worked with objects of type Syslog , which was not very flexible. Then we realized that it could work with any object that could respond to a .send() message, and identified this method as the interface that it needed to comply with. Now, in this version, Syslog is actually extending the abstract base class named `DataTargetClient`, which defines the `send()` method.

From now on, it is up to every new type of data target (email, for instance) to extend this abstract base class and implement the `send()` method.

We can even modify this property at runtime for any other object that implements a `send()` method, and it will still work. This is the reason why it is often called dependency injection: because the dependency can be provided dynamically.

The avid reader might be wondering why this is actually necessary. Python is flexible enough (sometimes too flexible), and will allow us to provide an object like `EventStreamer` with any particular data target object, without this one having to comply with any interface because it is dynamically typed. The question is this: why do we need to define the abstract base class (interface) at all when we can simply pass an object with a `send()` method to it?

In all fairness, this is true; there is actually no need to do that, and the program will work just the same. After all, polymorphism does not mean (or require) inheritance to work. However, defining the abstract base class is a good practice that comes with some advantages, the first one being duck typing. Together with as duck typing, we can mention the fact that the models become more readable: remember that inheritance follows the rule of is a, so by declaring the abstract base class and extending from it, we are saying that, for instance, `Syslog` is `DataTargetClient`, which is something users of your code can read and understand (again, this is duck typing).

All in all, it is not mandatory to define the abstract base class, but it is desirable in order to achieve a cleaner design.

CHAPTER

FIVE

USING DECORATORS TO IMPROVE OUR CODE

5.1 1. What are decorators?

5.1.1 1.1. What are decorators in Python?

Decorators were introduced in Python a long time ago as a mechanism to simplify the way functions and methods are defined when they have to be modified after their original definition.

One of the original motivations for this was because functions such as `classmethod` and `staticmethod` were used to transform the original definition of the method, but they required an extra line, modifying the original definition of the function.

More generally speaking, every time we had to apply a transformation to a function, we had to call it with the modifier function, and then reassign it to the same name the function was originally defined with.

For instance, if we have a function called original , and then we have a function that changes the behavior of original on top of it, called modifier , we have to write something like the following:

```
def original(...):
...

original = modifier(original)
```

Notice how we change the function and reassign it to the same name. This is confusing, error-prone (imagine that someone forgets to reassign the function, or does reassign that but not in the line immediately after the function definition, but much farther away), and cumbersome. For this reason, some syntax support was added to the language.

The previous example could be rewritten like so:

```
@modifier
def original(...):
...
```

This means that decorators are just syntax sugar for calling whatever is after the decorator as a first parameter of the decorator itself, and the result would be whatever the decorator returns.

In line with the Python terminology, and our example, modifier is what we call the decorator, and original is the decorated function, often also called a **wrapped object**.

While the functionality was originally thought for methods and functions, the actual syntax allows any kind of object to be decorated, so we are going to explore decorators applied to functions, methods, generators, and classes.

One final note is that, while the name of a decorator is correct (after all, the decorator is in fact, making changes, extending, or working on top of the wrapped function), it is not to be confused with the decorator design pattern.

5.1.2 1.2. Decorate functions

Functions are probably the simplest representation of a Python object that can be decorated. We can use decorators on functions to apply all sorts of logic to them—we can validate parameters, check preconditions, change the behavior entirely, modify its signature, cache results (create a memorized version of the original function), and more.

As an example, we will create a basic decorator that implements a retry mechanism, controlling a particular domain-level exception and retrying a certain number of times:

```
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
```

The use of `@wraps` can be ignored for now. The use of _ in the for loop, means that the number is assigned to a variable we are not interested in at the moment, because it's not used inside the for loop (it's a common idiom in Python to name _ values that are ignored).

The `retry` decorator doesn't take any parameters, so it can be easily applied to any function, as follows:

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

As explained at the beginning, the definition of `@retry` on top of `run_operation` is just syntactic sugar that Python provides to actually execute `run_operation = retry(run_operation)`.

In this limited example, we can see how decorators can be used to create a generic retry operation that, under certain conditions (in this case, represented as exceptions that could be related to timeouts, for example), will allow calling the decorated code multiple times.

5.1.3 1.2. Decorate classes

Classes can also be decorated with the same as can be applied to syntax functions. The only difference is that when writing the code for this decorator, we have to take into consideration that we are receiving a class, not a function.

Some practitioners might argue that decorating a class is something rather convoluted and that such a scenario might jeopardize readability because we would be declaring some attributes and methods in the class, but behind the scenes, the decorator might be applying changes that would render a completely different class.

This assessment is true, but only if this technique is heavily abused. Objectively, this is no different from decorating functions; after all, classes are just another type of object in the Python ecosystem, as functions are. For now, we'll explore the benefits of decorators that apply particularly to classes:

- All the benefits of reusing code and the DRY principle. A valid case of a class decorator would be to enforce that multiple classes conform to a certain interface or criteria (by making this checks only once in the decorator that is going to be applied to those many classes).
- We could create smaller or simpler classes that will be enhanced later on by decorators
- The transformation logic we need to apply to a certain class will be much easier to maintain if we use a decorator, as opposed to more complicated (and often rightfully discouraged) approaches such as metaclasses

Among all possible applications of decorators, we will explore a simple example to give an idea of the sorts of things they can be useful for. Keep in mind that this is not the only application type for class decorators, but also that the code we show you could have many other multiple solutions as well, all with their pros and cons, but we chose decorators with the purpose of illustrating their usefulness.

Recalling our event systems for the monitoring platform, we now need to transform the data for each event and send it to an external system. However, each type of event might have its own particularities when selecting how to send its data.

In particular, the `event` for a login might contain sensitive information such as credentials that we want to hide. Other fields such as `timestamp` might also require some transformations since we want to show them in a particular format. A first attempt at complying with these requirements would be as simple as having a class that maps to each particular `event` and knows how to serialize it:

```python
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()
```

Here, we declare a class that is going to map directly with the login event, containing the logic for it: hide the password field, and format the timestamp as required.

While this works and might look like a good option to start with, as time passes and we want to extend our system, we will find some issues:

- **Too many classes**: As the number of events grows, the number of serialization classes will grow in the same order of magnitude, because they are mapped one to one.
- **The solution is not flexible enough**: If we need to reuse parts of the components (for example, we need to hide the password in another type of event that also has it), we will have to extract this into a function, but also call it repeatedly from multiple classes, meaning that we are not reusing that much code after all.
- **Boilerplate**: The `serialize()` method will have to be present in all event classes, calling the same code. Although we can extract this into another class (creating a mixin), it does not seem like a good use of inheritance.

An alternative solution is to be able to dynamically construct an object that, given a set of filters (transformation functions) and an event instance, is able to serialize it by applying the filters to its fields. We then only need to define the functions to transform each type of field, and the serializer is created by composing many of these functions.

Once we have this object, we can decorate the class in order to add the `serialize()` method, which will just call these Serialization objects with itself:

```python
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:
    def __init__(self, serialization_fields: dict) -> None:
        self.serialization_fields = serialization_fields

    def serialize(self, event) -> dict:
        return {
            field: transformation(getattr(event, field))
            for field, transformation in
            self.serialization_fields.items()
        }
class Serialization:
    def __init__(self, **transformations):
        self.serializer = EventSerializer(transformations)

    def __call__(self, event_class):
        def serialize_method(event_instance):
            return self.serializer.serialize(event_instance)

        event_class.serialize = serialize_method
    return event_class

@Serialization(
    username=show_original,
    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
```

Notice how the decorator makes it easier for the user to know how each field is going to be treated without having to look into the code of another class. Just by reading the arguments passed to the class decorator, we know that the username and IP address will be left unmodified, the password will be hidden, and the timestamp will be formatted.

Now, the code of the class does not need the `serialize()` method defined, nor does it need to extend from a mixin that implements it, since the decorator will add it. In fact, this is probably the only part that justifies the creation of the class decorator, because otherwise, the `Serialization` object could have been a class attribute of `LoginEvent`, but the fact that it is altering the class by adding a new method to it makes it impossible.

Moreover, we could have another class decorator that, just by defining the attributes of the class, implements the logic of the init method, but this is beyond the scope of this example. This is what libraries such as `attrs` do, and a similar functionality is proposed in for the Standard library.

By using this class decorator, the previous example could be rewritten in a more compact way, without the boilerplate code of the init, as shown here:

```python
from dataclasses import dataclass
from datetime import datetime

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

Note that `@dataclass` is a decorator that is used to add generated special methods to classes. It examines the class to find fields. A field is defined as class variable that has a type annotation. Nothing in `dataclass()` examines the type specified in the variable annotation.

5.1.4 1.3. Other types of decorator

Now that we know what the `@` syntax for decorators actually means, we can conclude that it isn't just functions, methods, or classes that can be decorated; actually, anything that can be defined, such as generators, coroutines, and even objects that have already been decorated, can be decorated, meaning that decorators can be stacked.

The previous example showed how decorators can be chained. We first defined the class, and then applied `@dataclass` to it, which converted it into a data class, acting as a container for those attributes. After that, the `@Serialization` will apply the logic to that class, resulting in a new class with the new `serialize()` method added to it. Another good use of decorators is for generators that are supposed to be used as coroutines. The main idea is that, before sending any data to a newly created generator, the latter has to be advanced up to their next `yield` statement by calling `next()` on it. This is a manual process that every user will have to remember and hence is error-prone. We could easily create a decorator that takes a generator as a parameter, calls `next()` to it, and then returns the generator.

5.1.5 1.4. Passing arguments to decorators

At this point, we already regard decorators as a powerful tool in Python. However, they could be even more powerful if we could just pass parameters to them so that their logic is abstracted even more.

There are several ways of implementing decorators that can take arguments, but we will go over the most common ones. The first one is to create decorators as nested functions with a new level of indirection, making everything in the decorator fall one level deeper. The second approach is to use a class for the decorator.

In general, the second approach favors readability more, because it is easier to think in terms of an object than three or more nested functions working with closures. However, for completeness, we will explore both, and the reader can decide what is best for the problem at hand.

1.4.1. Decorators with nested functions

Roughly speaking, the general idea of a decorator is to create a function that returns a function (often called a higher-order function). The internal function defined in the body of the decorator is going to be the one actually being called.

Now, if we wish to pass parameters to it, we then need another level of indirection. The first one will take the parameters, and inside that function, we will define a new function, which will be the decorator, which in turn will define yet another new function, namely the one to be returned as a result of the decoration process. This means that we will have at least three levels of nested functions.

Don't worry if this didn't seem clear so far. After reviewing the examples that are about to come, everything will become clear.

One of the first examples we saw of decorators implemented the retry functionality over some functions. This is a good idea, except it has a problem; our implementation did not allow us to specify the numbers of retries, and instead, this was a fixed number inside the decorator.

Now, we want to be able to indicate how many retries each instance is going to have, and perhaps we could even add a default value to this parameter. In order to do this, we need another level of nested functions—first for the parameters, and then for the decorator itself. This is because we are now going to have something in the form of the following: `@retry(arg1, arg2,... )`. And that has to return a decorator because the `@` syntax will apply the result of that computation to the object to be decorated. Semantically, it would translate to something like the following: `<original_function> = retry(arg1, arg2, ....)(<original_function>)`

Besides the number of desired retries, we can also indicate the types of exception we wish to control. The new version of the code supporting the new requirements might look like this:

```
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
```

Here are some examples of how this decorator can be applied to functions, showing the different options it accepts:

```
@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)
```

(continues on next page)

(continued from previous page)

```
)
def run_with_custom_parameters(task):
    return task.run()
```

1.4.2. Decorator objects

The previous example requires three levels of nested functions. The first it is going to be a function that receives the parameters of the decorator we want to use. Inside this function, the rest of the functions are closures that use these parameters along with the logic of the decorator.

A cleaner implementation of this would be to use a class to define the decorator. In this case, we can pass the parameters in the `__init__` method, and then implement the logic of the decorator on the magic method named `__call__`.

The code for the decorator will look like it does in the following example:

```
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, e)
                    last_raised = e

            raise last_raised

        return wrapped
```

And this decorator can be applied pretty much like the previous one, like so:

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

It is important to note how the Python syntax takes effect here. First, we create the object, so before the `@` operation is applied, the object is created with its parameters passed to it. This will create a new object and initialize it with these parameters, as defined in the `init` method. After this, the `@` operation is invoked, so this object will wrap the function named `run_with_custom_retries_limit`, meaning that it will be passed to the call magic method.

Inside this `call` magic method, we defined the logic of the decorator as we normally do: we wrap the original function, returning a new one with the logic we want instead.

5.1.6 1.5. Good uses for decorators

In this section, we will take a look at some common patterns that make good use of decorators. These are common situations for when decorators are a good choice.

From all the countless applications decorators can be used for, we will enumerate a few, the most common or relevant:

- **Transforming parameters**: Changing the signature of a function to expose a nicer API, while encapsulating details on how the parameters are treated and transformed underneath.
- **Tracing code**: Logging the execution of a function with its parameters.
- **Validate parameters**.
- **Implement retry operations**.
- **Simplify classes by moving some (repetitive) logic into decorators**.

1.5.1. Transforming parameters

We have mentioned before that decorators can be used to validate parameters (and even enforce some preconditions or postconditions under the idea of DbC), so from this you probably have got the idea that it is somehow common to use decorators when dealing with or manipulating parameters.

In particular, there are some cases on which we find ourselves repeatedly creating similar objects, or applying similar transformations that we would wish to abstract away. Most of the time, we can achieve this by simply using a decorator.

1.5.2. Tracing code

When talking about **tracing** in this section, we will refer to something more general that has to do with dealing with the execution of a function that we wish to monitor. This could refer to scenarios in which we want to:

- Actually trace the execution of a function (for example, by logging the lines it executes)
- Monitor some metrics over a function (such as CPU usage or memory footprint)
- Measure the running time of a function
- Log when a function was called, and the parameters that were passed to it

5.2 2. Effective decorators: avoid common mistakes

While decorators are a great feature of Python, they are not exempt from issues if used incorrectly. In this section, we will see some common issues to avoid in order to create effective decorators.

5.2.1 2.1. Preserving data about the original wrapped object

One of the most common problems when applying a decorator to a function is that some of the properties or attributes of the original function are not maintained, leading to undesired, and hard-to-track, side-effects.

To illustrate this we show a decorator that is in charge of logging when the function is about to run:

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

Now, let's imagine we have a function with this decorator applied to it. We might initially think that nothing of that function is modified with respect to its original definition:

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

But maybe there are changes.

The decorator is not supposed to alter anything from the original function, but, as it turns out since it contains a flaw it's actually modifying its name and docstring, among other properties.

Let's try to get help for this function:

```
>>> help(process_account)
Help on function wrapped in module decorator_wraps_1:
wrapped(*args, **kwargs)
```

And let's check how it's called: .. code-block:: python

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

We can see that, since the decorator is actually changing the original function for a new one (called `wrapped`), what we actually see are the properties of this function instead of those from the original function.

If we apply a decorator like this one to multiple functions, all with different names, they will all end up being called wrapped, which is a major concern (for example, if we want to log or trace the function, this will make debugging even harder).

Another problem is that, in case we placed docstrings with tests on these functions, they will be overridden by those of the decorator. As a result, the docstrings with the test we want will not run when we call our code with the `doctest` module.

The fix is simple, though. We just have to apply the wraps decorator in the internal function (`wrapped`), telling it that it is actually wrapping function :

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

    return wrapped
```

Now, if we check the properties, we will obtain what we expected in the first place. Check help for the function, like so:

```
>>> Help on function process_account in module decorator_wraps_2:
process_account(account_id)
Process an account by Id.
```

And verify that its qualified name is correct, like so:

```
>>> process_account.__qualname__
'process_account'
```

Most importantly, we recovered the unit tests we might have had on the docstrings! By using the wraps decorator, we can also access the original, unmodified function under the `__wrapped__` attribute. Although it should not be used in production, it might come in handy in some unit tests when we want to check the unmodified version of the function.

In general, for simple decorators, the way we would use `functools.wraps` would typically follow the general formula or structure:

```
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
```

Note: Always use `functools.wraps` applied over the wrapped function, when creating a decorator, as shown in the preceding formula.

5.2.2 2.2. Dealing with side-effects in decorators

In this section, we will learn that it is advisable to avoid side-effects in the body of the decorator. There are cases where this might be acceptable, but the bottom line is that, if in case of doubt, decide against it, for the reasons that are explained ahead. Everything that the decorator needs to do aside from the function that it's decorating should be placed in the innermost function definition, or there will be problems when it comes to importing.

Nonetheless, sometimes these side-effects are required (or even desired) to run at import time, and the obverse applies.

We will see examples of both, and where each one applies. If in doubt, err on the side of caution, and delay all side-effects until the very latest, right after the `wrapped` function is going to be called.

Next, we will see when it's not a good idea to place extra logic outside the `wrapped` function.

2.2.1. Incorrect handling of side-effects in a decorator

Let's imagine the case of a decorator that was created with the goal of logging when a function started running and then logging its running time:

```
def traced_function_wrong(function):
    logger.info("started execution of %s", function)
    start_time = time.time()

    @functools.wraps(function)
    def wrapped(*args, **kwargs):
        result = function(*args, **kwargs)
        logger.info(
            "function %s took %.2fs",
            function,
            time.time() - start_time
        )
        return result
    return wrapped
```

Now we will apply the decorator to a regular function, thinking that it will work just fine:

```
@traced_function_wrong
def process_with_delay(callback, delay=0):
    time.sleep(delay)
    return callback()
```

This decorator has a subtle, yet critical bug in it. First, let's import the function, call it several times, and see what happens:

```
>>> from decorator_side_effects_1 import process_with_delay
INFO:started execution of <function process_with_delay at 0x...>
```

Just by importing the function, we will notice that something's amiss. The logging line should not be there, because the function was not invoked.

Now, what happens if we run the function, and see how long it takes to run? Actually, we would expect that calling the same function multiple times will give similar results:

```
>>> main()
...
INFO:function <function process_with_delay at 0x> took 8.67s
>>> main()
...
INFO:function <function process_with_delay at 0x> took 13.39s
>>> main()
...
INFO:function <function process_with_delay at 0x> took 17.01s
```

Every time we run the same function, it takes longer! At this point, you have probably already noticed the (now obvious) error.

Remember the syntax for decorators. `@traced_function_wrong` actually means the following: `process_with_delay = traced_function_wrong(process_with_delay)`. And this will run when the module is imported. Therefore, the time that is set in the function will be the one at the time the module was imported. Successive calls will compute the time difference from the running time until that original starting time. It will also log at the wrong moment, and not when the function is actually called.

Luckily, the fix is also very simple: we just have to move the code inside the wrapped function in order to delay its execution:

```
def traced_function(function):
    @functools.wraps(function)
    def wrapped(*args, **kwargs):
        logger.info("started execution of %s", function.__qualname__)
        start_time = time.time()
        result = function(*args, **kwargs)
        logger.info(
            "function %s took %.2fs",
            function.__qualname__,
            time.time() - start_time
        )
        return result
    return wrapped
```

With this new version, the previous problems are resolved.

If the actions of the decorator had been different, the results could have been much more disastrous. For instance, if it requires that you log events and send them to an external service, it will certainly fail unless the configuration has been run right before this has been imported, which we cannot guarantee. Even if we could, it would be bad practice. The same applies if the decorator has any other sort of side-effect, such as reading from a file, parsing a configuration, and many more.

2.2.2. Requiring decorators with side-effects

Sometimes, side-effects on decorators are necessary, and we should not delay their execution until the very last possible time, because that's part of the mechanism which is required for them to work.

One common scenario for when we don't want to delay the side-effect of decorators is when we need to register objects to a public registry that will be available in the module.

For instance, going back to our previous event system example, we now want to only make some events available in the module, but not all of them. In the hierarchy of events, we might want to have some intermediate classes that are not actual events we want to process on the system, but some of their derivative classes instead.

Instead of flagging each class based on whether it's going to be processed or not, we could explicitly register each class through a decorator.

In this case, we have a class for all events that relate to the activities of a user. However, this is just an intermediate table for the types of event we actually want, namely `UserLoginEvent` and `UserLogoutEvent`:

```
EVENTS_REGISTRY = {}

def register_event(event_cls):
    """Place the class for the event into the registry to make it
    accessible in
    the module.
    """
    EVENTS_REGISTRY[event_cls.__name__] = event_cls
    return event_cls

class Event:
    """A base event object"""

class UserEvent:
    TYPE = "user"

@register_event
class UserLoginEvent(UserEvent):
    """Represents the event of a user when it has just accessed the
    system."""

@register_event
class UserLogoutEvent(UserEvent):
    """Event triggered right after a user abandoned the system."""
```

When we look at the preceding code, it seems that `EVENTS_REGISTRY` is empty, but after importing something from this module, it will get populated with all of the classes that are under the register_event decorator:

```
>>> from decorator_side_effects_2 import EVENTS_REGISTRY
>>> EVENTS_REGISTRY
{'UserLoginEvent': decorator_side_effects_2.UserLoginEvent,
'UserLogoutEvent': decorator_side_effects_2.UserLogoutEvent}
```

This might seem like it's hard to read, or even misleading, because `EVENTS_REGISTRY` will have its final value at runtime, right after the module was imported, and we cannot easily predict its value by just looking at the code.

While that is true, in some cases this pattern is justified. In fact, many web frameworks or well-known libraries use this to work and expose objects or make them available.

It is also true that in this case, the decorator is not changing the wrapped object, nor altering the way it works in any way. However, the important note here is that, if we were to do some modifications and define an internal function that modifies the wrapped object, we would still probably want the code that registers the resulting object outside it.

Notice the use of the word outside. It does not necessarily mean before, it's just not part of the same closure; but it's in the outer scope, so it's not delayed until runtime.

5.2.3 2.3. Creating decorators that will always work

There are several different scenarios to which decorators might apply. It can also be the case that we need to use the same decorator for objects that fall into these different multiple scenarios, for instance, if we want to reuse our decorator and apply it to a function, a class, a method, or a static method.

If we create the decorator, just thinking about supporting only the first type of object we want to decorate, we might notice that the same decorator does not work equally well on a different type of object. The typical example is where we create a decorator to be used on a function, and then we want to apply it to a method of a class, only to realize that it does not work. A similar scenario might occur if we designed our decorator for a method, and then we want it to also apply for static methods or class methods.

When designing decorators, we typically think about reusing code, so we will want to use that decorator for functions and methods as well.

Defining our decorators with the signature `*args` , and `**kwargs` , will make them work in all cases, because it's the most generic kind of signature that we can have. However, sometimes we might want not to use this, and instead define the decorator wrapping function according to the signature of the original function, mainly because of two reasons:

- It will be more readable since it resembles the original function.
- It actually needs to do something with the arguments, so receiving `*args` and `**kwargs` wouldn't be convenient.

Consider the case on which we have many functions in our code base that require a particular object to be created from a parameter. For instance, we pass a string, and initialize a driver object with it, repeatedly. Then we think we can remove the duplication by using a decorator that will take care of converting this parameter accordingly.

In the next example, we pretend that `DBDriver` is an object that knows how to connect and run operations on a database, but it needs a connection string. The methods we have in our code, are designed to receive a string with the information of the database and require to create an instance of `DBDriver` always. The idea of the decorator is that it's going to take place of this conversion automatically: the function will continue to receive a string, but the decorator will create a `DBDriver` and pass it to the function, so internally we can assume that we receive the object we need directly.

An example of using this in a function is shown in the next listing:

```python
import logging
from functools import wraps

logger = logging.getLogger(__name__)

class DBDriver:
    def __init__(self, dbstring):
        self.dbstring = dbstring

    def execute(self, query):
        return f"query {query} at {self.dbstring}"

def inject_db_driver(function):
    """This decorator converts the parameter by creating a ``DBDriver``
    instance from the database dsn string.
    """

    @wraps(function)
    def wrapped(dbstring):
        return function(DBDriver(dbstring))

    return wrapped

@inject_db_driver
def run_query(driver):
    return driver.execute("test_function")
```

It's easy to verify that if we pass a string to the function, we get the result done by an instance of `DBDriver`, so the decorator works as expected:

```
>>> run_query("test_OK")
'query test_function at test_OK'
```

But now, we want to reuse this same decorator in a class method, where we find the same problem:

```
class DataHandler:
    @inject_db_driver
    def run_query(self, driver):
        return driver.execute(self.__class__.__name__)
```

We try to use this decorator, only to realize that it doesn't work:

```
>>> DataHandler().run_query("test_fails")
Traceback (most recent call last):
...
TypeError: wrapped() takes 1 positional argument but 2 were given
```

What is the problem? The method in the class is defined with an extra argument: `self`. Methods are just a particular kind of function that receives self (the object they're defined upon) as the first parameter.

Therefore, in this case, the decorator (designed to work with only one parameter, named `dbstring`), will interpret that self is said parameter, and call the method passing the string in the place of self, and nothing in the place for the second parameter, namely the string we are passing.

To fix this issue, we need to create a decorator that will work equally for methods and functions, and we do so by defining this as a decorator object, that also implements the protocol descriptor.

The solution is to implement the decorator as a class object and make this object a description, by implementing the `__get__` method.

```
from functools import wraps
from types import MethodType

class inject_db_driver:
    """Convert a string to a DBDriver instance and pass this to the
    wrapped function."""
    def __init__(self, function):
        self.function = function
        wraps(self.function)(self)

    def __call__(self, dbstring):
        return self.function(DBDriver(dbstring))

    def __get__(self, instance, owner):
        if instance is None:
            return self

        return self.__class__(MethodType(self.function, instance))
```

For now, we can say that what this decorator does is actually rebinding the callable it's decorating to a method, meaning that it will bind the function to the object, and then recreate the decorator with this new callable.

For functions, it still works, because it won't call the `__get__` method at all.

5.3 3. The DRY principle with decorators

We have seen how decorators allow us to abstract away certain logic into a separate component. The main advantage of this is that we can then apply the decorator multiple times into different objects in order to reuse code. This follows the **Don't Repeat Yourself (DRY)** principle since we define certain knowledge once and only once.

The retry mechanism implemented in the previous sections is a good example of a decorator that can be applied multiple times to reuse code. Instead of making each particular function include its retry logic, we create a decorator and apply it several times. This makes sense once we have made sure that the decorator can work with methods and functions equally.

The class decorator that defined how events are to be represented also complies with the DRY principle in the sense that it defines one specific place for the logic for serializing an event, without needing to duplicate code scattered among different classes. Since we expect to reuse this decorator and apply it to many classes, its development (and complexity) pay off.

This last remark is important to bear in mind when trying to use decorators in order to reuse code: we have to be absolutely sure that we will actually be saving code.

Any decorator (especially if it is not carefully designed) adds another level of indirection to the code, and hence more complexity. Readers of the code might want to follow the path of the decorator to fully understand the logic of the function (although these considerations are addressed in the following section), so keep in mind that this complexity has to pay off. If there is not going to be too much reuse, then do not go for a decorator and opt for a simpler option (maybe just a separate function or another small class is enough).

But how do we know what too much reuse is? Is there a rule to determine when to refactor existing code into a decorator? There is nothing specific to decorators in Python, but we could apply a general rule of thumb in software engineering that states that a component should be tried out at least three times before considering creating a generic abstraction in the sort of a reusable component.

The bottom line is that reusing code through decorators is acceptable, but only when you take into account the following considerations:

- Do not create the decorator in the first place from scratch. Wait until the pattern emerges and the abstraction for the decorator becomes clear, and then refactor.
- Consider that the decorator has to be applied several times (at least three times) before implementing it.
- Keep the code in the decorators to a minimum.

5.4 4. Decorators and separation of concerns

The last point on the previous list is so important that it deserves a section of its own. We have already explored the idea of reusing code and noticed that a key element of reusing code is having components that are cohesive. This means that they should have the minimum level of responsibility: do one thing, one thing only, and do it well. The smaller our components, the more reusable, and the more they can be applied in a different context without carrying extra behavior that will cause coupling and dependencies, which will make the software rigid.

To show you what this means, let's reprise one of the decorators that we used in a previous example. We created a decorator that traced the execution of certain functions with code similar to the following:

```python
def traced_function(function):

    @functools.wraps(function)
    def wrapped(*args, **kwargs):
        logger.info("started execution of %s", function.__qualname__)
        start_time = time.time()
        result = function(*args, **kwargs)
        logger.info(
            "function %s took %.2fs",
            function.__qualname__,
```

(continues on next page)

(continued from previous page)

```
            time.time() - start_time
        )
        return result

    return wrapped
```

Now, this decorator, while it works, has a problem: it is doing more than one thing. It logs that a particular function was just invoked, and also logs how much time it took to run. Every time we use this decorator, we are carrying these two responsibilities, even if we only wanted one of them.

This should be broken down into smaller decorators, each one with a more specific and limited responsibility:

```
def log_execution(function):
    @wraps(function)
    def wrapped(*args, **kwargs):
        logger.info("started execution of %s", function.__qualname__)
        return function(*kwargs, **kwargs)
    return wrapped

def measure_time(function):
    @wraps(function)
    def wrapped(*args, **kwargs):
        start_time = time.time()
        result = function(*args, **kwargs)
        logger.info("function %s took %.2f", function.__qualname__,
        time.time() - start_time)
        return result
    return wrapped
```

Notice that the same functionality that we had previously can be achieved by simply combining both of them:

```
@measure_time
@log_execution
def operation():
    ....
```

Notice how the order in which the decorators are applied is also important.

Note: Do not place more than one responsibility in a decorator. The SRP applies to decorators as well.

5.5 5. Analyzing good decorators

As a closing note for this chapter, let's review some examples of good decorators and how they are used both in Python itself, as well as in popular libraries. The idea is to get guidelines on how good decorators are created.

Before jumping into examples, let's first identify traits that good decorators should have:

- **Encapsulation, or separation of concerns**: A good decorator should effectively separate different responsibilities between what it does and what it is decorating. It cannot be a leaky abstraction, meaning that a client of the decorator should only invoke it in black box mode, without knowing how it is actually implementing its logic.
- **Orthogonality**: What the decorator does should be independent, and as decoupled as possible from the object it is decorating.
- **Reusability**: It is desirable that the decorator can be applied to multiple types, and not that it just appears on one instance of one function, because that means that it could just have been a function instead. It has to be generic enough.

A nice example of decorators can be found in the Celery project, where a task is defined by applying the decorator of the task from the application to a function:

```
@app.task
def mytask():
    ....
```

One of the reasons why this is a good decorator is because it is very good at something: encapsulation. The user of the library only needs to define the function body and the decorator will convert that into a task automatically. The `@app.task` decorator surely wraps a lot of logic and code, but none of that is relevant to the body of `mytask()` . It is complete encapsulation and separation of concerns—nobody will have to take a look at what that decorator does, so it is a correct abstraction that does not leak any details.

Another common use of decorators is in web frameworks (Pyramid, Flask, and Sanic, just to name a few), on which the handlers for views are registered to the URLs through decorators:

```
@route("/", method=["GET"])
def view_handler(request):
    ...
```

These sorts of decorator have the same considerations as before; they also provide total encapsulation because a user of the web framework rarely (if ever) needs to know what the `@route` decorator is doing. In this case, we know that the decorator is doing something more, such as registering these functions to a mapper to the URL, and also that it is changing the signature of the original function to provide us with a nicer interface that receives a request object with all the information already set.

The previous two examples are enough to make us notice something else about this use of decorators. They conform to an API. These libraries of frameworks are exposing their functionality to users through decorators, and it turns out that decorators are an excellent way of defining a clean programming interface.

This is probably the best way we should think about to decorators. Much like in the example of the class decorator that tells us how the attributes of the event are going to be handled, a good decorator should provide a clean interface so that users of the code know what to expect from the decorator, without needing to know how it works, or any of its details for that matter.

CHAPTER

SIX

GETTING MORE OUT OF OUR OBJECTS WITH DESCRIPTORS

Descriptors are another distinctive feature of Python that takes object-oriented programming to another level, and their potential allows users to build more powerful and reusable abstractions. Most of the time, the full potential of descriptors is observed in libraries or frameworks.

6.1 1. A first look at descriptors

First, we will explore the main idea behind descriptors to understand their mechanics and internal workings. Once this is clear, it will be easier to assimilate how the different types of descriptors work, which we will explore in the next section.

Once we have a first understanding of the idea behind descriptors, we will look at an example where their use gives us a cleaner and more Pythonic implementation.

6.1.1 1.1. The machinery behind descriptors

The way descriptors work is not all that complicated, but the problem with them is that there are a lot of caveats to take into consideration, so the implementation details are of the utmost importance here.

In order to implement descriptors, we need at least two classes. For the purposes of this generic example, we are going to call the client class to the one that is going to take advantage of the functionality we want to implement in the descriptor (this class is generally just a domain model one, a regular abstraction we create for our solution), and we are going to call the descriptor class to the one that implements the logic of the descriptor.

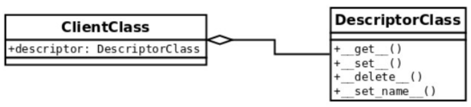
A descriptor is, therefore, just an object that is an instance of a class that implements the descriptor protocol. This means that this class must have its interface containing at least one of the following magic methods (part of the descriptor protocol as of Python 3.6+):

- `__get__`
- `__set__`
- `__delete__`
- `__set_name__`

For the purposes of this initial high-level introduction, the following naming convention will be used:

- *ClientClass*: The domain-level abstraction that will take advantage of the functionality to be implemented by the descriptor. This class is said to be a client of the descriptor. This class contains a class attribute (named descriptor by this convention), which is an instance of *DescriptorClass*.
- *DescriptorClass*: The class that implements the descriptor itself. This class should implement some of the aforementioned magic methods that entail the descriptor protocol.
- *client*: An instance of *ClientClass*. `client = ClientClass()`
- *descriptor*: An instance of *DescriptorClass*. `descriptor = DescriptorClass()`. This object is a class attribute that is placed in *ClientClass*.

This relationship is illustrated in the following diagram:



A very important observation to keep in mind is that for this protocol to work, the *descriptor* object has to be defined as a class attribute. Creating this object as an instance attribute will not work, so it must be in the body of the class, and not in the `init` method.

Note: Always place the *descriptor* object as a class attribute!

On a slightly critical note, readers can also note that it is possible to implement the descriptor protocol partially: not all methods must always be defined; instead, we can implement only those we need, as we will see shortly.

So, now we have the structure in place: we know what elements are set and how they interact. We need a class for the *descriptor*, another class that will consume the logic of the *descriptor*, which, in turn, will have a *descriptor* object (an instance of the *DescriptorClass*) as a class attribute, and instances of *ClientClass* that will follow the descriptor protocol when we call for the attribute named `descriptor`. But now what? How does all of this fit into place at runtime?

Normally, when we have a regular class and we access its attributes, we simply obtain the objects as we expect them, and even their properties, as in the following example:

```
>>> class Attribute:
...     value = 42
...
>>> class Client:
...     attribute = Attribute()
...
>>> Client().attribute
<__main__.Attribute object at 0x7ff37ea90940>
>>> Client().attribute.value
42
```

But, in the case of descriptors, something different happens. When an object is defined as a class attribute (and this one is a *descriptor*), when a client requests this attribute, instead of getting the object itself (as we would expect from the previous example), we get the result of having called the `__get__` magic method.

Let's start with some simple code that only logs information about the context, and returns the same *client* object:

```
class DescriptorClass:

    def __get__(self, instance, owner):
        if instance is None:
            return self

        logger.info("Call: %s.__get__(%r, %r)",
        self.__class__.__name__,instance, owner)
        return instance

class ClientClass:
    descriptor = DescriptorClass()
```

When running this code, and requesting the descriptor attribute of an instance of *ClientClass*, we will discover that we are, in fact, not getting an instance of *DescriptorClass*, but whatever its __get__() method returns instead:

```
>>> client = ClientClass()
>>> client.descriptor
```

(continues on next page)

(continued from previous page)

```
INFO:Call: DescriptorClass.__get__(<ClientClass object at 0x...>, <class
↪'ClientClass'>)
<ClientClass object at 0x...>
>>> client.descriptor is client
INFO:Call: DescriptorClass.__get__(ClientClass object at 0x...>, <class
↪'ClientClass'>)
True
```

Notice how the logging line, placed under the `__get__` method, was called instead of just returning the object we created. In this case, we made that method return the *client* itself, hence making true a comparison of the last statement. The parameters of this method are explained in more detail in the following subsections when we explore each method in more detail.

Starting from this simple, yet demonstrative example, we can start creating more complex abstractions and better decorators, because the important note here is that we have a new (powerful) tool to work with. Notice how this changes the control flow of the program in a completely different way. With this tool, we can abstract all sorts of logic behind the `__get__` method, and make the *descriptor* transparently run all sorts of transformations without clients even noticing. This takes encapsulation to a new level.

6.1.2 1.2. Exploring each method of the descriptor protocol

Up until now, we have seen quite a few examples of descriptors in action, and we got the idea of how they work. These examples gave us a first glimpse of the power of descriptors, but you might be wondering about some implementation details and idioms whose explanation we failed to address.

Since descriptors are just objects, these methods take `self` as the first parameter. For all of them, this just means the descriptor object itself.

In this section, we will explore each method of the descriptor protocol, in full detail, explaining what each parameter signifies, and how they are intended to be used.

1.2.1. __get__(self, instance, owner)

The first parameter, `instance`, refers to the object from which the *descriptor* is being called. In our first example, this would mean the *client* object.

The `owner` parameter is a reference to the class of that object, which following our example would be *ClientClass*.

From the previous paragraph we conclude that the parameter named `instance` in the signature of `__get__` is the object over which the *descriptor* is taking action, and `owner` is the class of `instance`. The avid reader might be wondering why is the signature define like this, after all the class can be taken from `instance` directly (`owner = instance.__class__`). There is an edge case: when the *descriptor* is called from the class (*ClientClass*), not from the instance (*client*), then the value of `instance` is None, but we might still want to do some processing in that case.

With the following simple code we can demonstrate the difference of when a descriptor is being called from the class, or from an instance. In this case, the `__get__` method is doing two separate things for each case.

```python
class DescriptorClass:
    def __get__(self, instance, owner):
        if instance is None:
            return f"{self.__class__.__name__}.{owner.__name__}"
        return f"value for {instance}"

class ClientClass:
    descriptor = DescriptorClass()
```

When we call it from *ClientClass* directly it will do one thing, which is composing a namespace with the names of the classes:

```
>>> ClientClass.descriptor
'DescriptorClass.ClientClass'
```

And then if we call it from an object we have created, it will return the other message instead:

```
>>> ClientClass().descriptor
'value for <descriptors_methods_1.ClientClass object at 0x...>'
```

In general, unless we really need to do something with the `owner` parameter, the most common idiom, is to just return the descriptor itself, when instance is None.

1.2.2. __set__(self, instance, value)

This method is called when we try to assign something to a *descriptor*. It is activated with statements such as the following, in which a *descriptor* is an object that implements `__set__()`. The `instance` parameter, in this case, would be *client*, and the value would be the "value" string: `client.descriptor = "value"`

If `client.descriptor` doesn't implement `__set__()`, then "value" will override the descriptor entirely.

Note: Be careful when assigning a value to an attribute that is a descriptor. Make sure it implements the `__set__` method, and that we are not causing an undesired side effect.

By default, the most common use of this method is just to store data in an object. Nevertheless, we have seen how powerful descriptors are so far, and that we can take advantage of them, for example, if we were to create generic validation objects that can be applied multiple times (again, this is something that if we don't abstract, we might end up repeating multiple times in setter methods of properties).

The following listing illustrates how we can take advantage of this method in order to create generic validation objects for attributes, which can be created dynamically with functions to validate on the values before assigning them to the object:

```python
class Validation:

    def __init__(self, validation_function, error_msg: str):
        self.validation_function = validation_function
        self.error_msg = error_msg

    def __call__(self, value):
        if not self.validation_function(value):
            raise ValueError(f"{value!r} {self.error_msg}")

class Field:

    def __init__(self, *validations):
        self._name = None
        self.validations = validations

    def __set_name__(self, owner, name):
        self._name = name

    def __get__(self, instance, owner):
        if instance is None:
            return self

        return instance.__dict__[self._name]

    def validate(self, value):
        for validation in self.validations:
            validation(value)
```

(continues on next page)

(continued from previous page)

```
    def __set__(self, instance, value):
        self.validate(value)
        instance.__dict__[self._name] = value

class ClientClass:
    descriptor = Field(
        Validation(lambda x: isinstance(x, (int, float)), "is not a number"),
        Validation(lambda x: x >= 0, "is not >= 0")
    )
```

We can see this object in action in the following listing:

```
>>> client = ClientClass()
>>> client.descriptor = 42
>>> client.descriptor
42
>>> client.descriptor = -42
Traceback (most recent call last):
...
ValueError: -42 is not >= 0
>>> client.descriptor = "invalid value"
...
ValueError: 'invalid value' is not a number
```

The idea is that something that we would normally place in a property can be abstracted away into a *descriptor*, and reuse it multiple times. In this case, the `__set__()` method would be doing what the `@property.setter` would have been doing.

1.2.3. __delete__(self, instance)

This method is called upon with the following statement, in which `self` would be the *descriptor* attribute, and `instance` would be the client object in this example:

```
>>> del client.descriptor
```

In the following example, we use this method to create a *descriptor* with the goal of preventing you from removing attributes from an object without the required administrative privileges. Notice how, in this case, that the *descriptor* has logic that is used to predicate with the values of the object that is using it, instead of different related objects:

```
class ProtectedAttribute:

    def __init__(self, requires_role=None) -> None:
        self.permission_required = requires_role
        self._name = None

    def __set_name__(self, owner, name):
        self._name = name

    def __set__(self, user, value):
        if value is None:
            raise ValueError(f"{self._name} can't be set to None")

        user.__dict__[self._name] = value

    def __delete__(self, user):
        if self.permission_required in user.permissions:
            user.__dict__[self._name] = None
        else:
```

(continues on next page)

(continued from previous page)

```
            raise ValueError(f"User {user!s} doesn't have {self.permission_
↪required} permission")

class User:
    """Only users with "admin" privileges can remove their email
    address."""

    email = ProtectedAttribute(requires_role="admin")

    def __init__(self, username: str, email: str, permission_list: list = None) ->␣
↪None:
        self.username = username
        self.email = email
        self.permissions = permission_list or []

    def __str__(self):
        return self.username
```

Before seeing examples of how this object works, it's important to remark some of the criteria of this descriptor. Notice the `User` class requires the `username` and `email` as mandatory parameters. According to its `__init__` method, it cannot be a user if it doesn't have an email attribute. If we were to delete that attribute, and extract it from the object entirely we would be creating an inconsistent object, with some invalid intermediate state that does not correspond to the interface defined by the class `User`. Details like this one are really important, in order to avoid issues. Some other object is expecting to work with this `User`, and it also expects that it has an email attribute.

For this reason, it was decided that the "deletion" of an email will just simply set it to None. For the same reason, we must forbid someone trying to set a None value to it, because that would bypass the mechanism we placed in the `__delete__ method`.

Here, we can see it in action, assuming a case where only users with "admin" privileges can remove their email address:

```
>>> admin = User("root", "root@d.com", ["admin"])
>>> user = User("user", "user1@d.com", ["email", "helpdesk"])
>>> admin.email
'root@d.com'
>>> del admin.email
>>> admin.email is None
True
>>> user.email
'user1@d.com'
>>> user.email = None
ValueError: email can't be set to None
>>> del user.email
ValueError: User user doesn't have admin permission
```

Here, in this simple *descriptor*, we see that we can delete the email from users that contain the "admin" permission only. As for the rest, when we try to call `del` on that attribute, we will get a `ValueError` exception.

In general, this method of the *descriptor* is not as commonly used as the two previous ones, but it is worth showing it for completeness.

1.2.4. __set_name__(self, owner, name)

When we create the *descriptor* object in the class that is going to use it, we generally need the *descriptor* to know the name of the attribute it is going to be handling.

This attribute name is the one we use to read from and write to `__dict__` in the `__get__` and `__set__` methods, respectively.

Before Python 3.6, the descriptor couldn't take this name automatically, so the most general approach was to just pass it explicitly when initializing the object. This works fine, but it has an issue in that it requires that we duplicate the name every time we want to use the descriptor for a new attribute.

This is what a typical *descriptor* would look like if we didn't have this method:

```python
class DescriptorWithName:

    def __init__(self, name):
        self.name = name

    def __get__(self, instance, value):
        if instance is None:
            return self

        logger.info("getting %r attribute from %r", self.name, instance)
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value

class ClientClass:
    descriptor = DescriptorWithName("descriptor")
```

We can see how the descriptor uses this value:

```
>>> client = ClientClass()
>>> client.descriptor = "value"
>>> client.descriptor
INFO:getting 'descriptor' attribute from <ClientClass object at 0x...>
'value'
```

Now, if we wanted to avoid writing the name of the attribute twice (once for the variable assigned inside the class, and once again as the name of the first parameter of the descriptor), we have to resort to a few tricks, like using a class decorator, or (even worse) using a metaclass.

In Python 3.6, the new method `__set_name__` was added, and it receives the class where that descriptor is being created, and the name that is being given to that descriptor. The most common idiom is to use this method for the descriptor so that it can store the required name in this method.

For compatibility, it is generally a good idea to keep a default value in the `__init__` method but still take advantage of `__set_name__`.

With this method, we can rewrite the previous descriptors as follows:

```python
class DescriptorWithName:
    def __init__(self, name=None):
        self.name = name

    def __set_name__(self, owner, name):
        self.name = name

    ...
```

6.2 2. Types of descriptors

Based on the methods we have just explored, we can make an important distinction among descriptors in terms of how they work. Understanding this distinction plays an important role in working effectively with descriptors, and will also help to avoid caveats or common errors at runtime.

If a descriptor implements the `__set__` or `__delete__` methods, it is called a **data descriptor**. Otherwise, a descriptor that solely implements `__get__` is a **non-data descriptor**. Notice that `__set_name__` does not affect this classification at all.

When trying to resolve an attribute of an object, a data descriptor will always take precedence over the dictionary of the object, whereas a non-data descriptor will not. This means that in a non-data descriptor if the object has a key on its dictionary with the same name as the descriptor, this one will always be called, and the descriptor itself will never run. Conversely, in a data descriptor, even if there is a key in the dictionary with the same name as the descriptor, this one will never be used since the descriptor itself will always end up being called.

The following two sections explain this in more detail, with examples, in order to get a deeper idea of what to expect from each type of descriptor.

6.2.1 2.1. Non-data descriptors

We will start with a descriptor that only implements the `__get__` method, and see how it is used:

```
class NonDataDescriptor:
    def __get__(self, instance, owner):
        if instance is None:
            return self
        return 42

class ClientClass:
    descriptor = NonDataDescriptor()
```

As usual, if we ask for the descriptor, we get the result of its `__get__` method:

```
>>> client = ClientClass()
>>> client.descriptor
42
```

But if we change the descriptor attribute to something else, we lose access to this value, and get what was assigned to it instead:

```
>>> client.descriptor = 43
>>> client.descriptor
43
```

Now, if we delete the descriptor, and ask for it again, let’s see what we get:

```
>>> del client.descriptor
>>> client.descriptor
42
```

Let’s rewind what just happened. When we first created the client object, the descriptor attribute lay in the class, not the instance, so if we ask for the dictionary of the client object, it will be empty:

```
>>> vars(client)
{}
```

And then, when we request the `.descriptor` attribute, it doesn’t find any key in `client.__dict__` named “descriptor”, so it goes to the class, where it will find it, but only as a descriptor, hence why it returns the result of the `__get__` method.

But then, we change the value of the `.descriptor` attribute to something else, and what this does is set this into the dictionary of the instance, meaning that this time it won't be empty:

```
>>> client.descriptor = 99
>>> vars(client)
{'descriptor': 99}
```

So, when we ask for the `.descriptor` attribute here, it will look for it in the object (and this time it will find it, because there is a key named descriptor in the `__dict__` attribute of the object, as the vars result is showing us), and return it without having to look for it in the class. For this reason, the descriptor protocol is never invoked, and the next time we ask for this attribute, it will instead return the value we have overridden it with (99).

Afterward, we delete this attribute by calling `del`, and what this does is remove the key "descriptor" from the dictionary of the object, leaving us back in the first scenario, where it's going to default to the class where the descriptor protocol will be activated:

```
>>> del client.descriptor
>>> vars(client)
{}
>>> client.descriptor
42
```

This means that if we set the attribute of the descriptor to something else, we might be accidentally breaking it. Why? Because the descriptor doesn't handle the delete action (some of them don't need to).

This is called a non-data descriptor because it doesn't implement the `__set__` magic method, as we will see in the next example.

6.2.2 2.2. Data descriptors

Now, let's look at the difference of using a data descriptor. For this, we are going to create another simple descriptor that implements the `__set__` method:

```
class DataDescriptor:
    def __get__(self, instance, owner):
        if instance is None:
            return self
        return 42

    def __set__(self, instance, value):
        logger.debug("setting %s.descriptor to %s", instance, value)
        instance.__dict__["descriptor"] = value

class ClientClass:
    descriptor = DataDescriptor()
```

Let's see what the value of the descriptor returns:

```
>>> client = ClientClass()
>>> client.descriptor
42
```

Now, let's try to change this value to something else, and see what it returns instead:

```
>>> client.descriptor = 99
>>> client.descriptor
42
```

The value returned by the descriptor didn't change. But when we assign a different value to it, it must be set to the dictionary of the object (as it was previously):

```
>>> vars(client)
{'descriptor': 99}
>>> client.__dict__["descriptor"]
99
```

So, the `__set__()` method was called, and indeed it did set the value to the dictionary of the object, only this time, when we request this attribute, instead of using the `__dict__` attribute of the dictionary, the descriptor takes precedence (because it's an overriding descriptor).

One more thing: deleting the attribute will not work anymore:

```
>>> del client.descriptor
Traceback (most recent call last):
...
AttributeError: __delete__
```

The reason is as follows: given that now, the descriptor always takes place, calling `del` on an object doesn't try to delete the attribute from the dictionary (`__dict__`) of the object, but instead it tries to call the `__delete__()` method of the descriptor (which is not implemented in this example, hence the attribute error).

This is the difference between data and non-data descriptors. If the descriptor implements `__set__()`, then it will always take precedence, no matter what attributes are present in the dictionary of the object. If this method is not implemented, then the dictionary will be looked up first, and then the descriptor will run.

An interesting observation you might have noticed is this line on the set method: `instance.__dict__["descriptor"] = value`. There are a lot of things to question about that line, but let's break it down into parts.

First, why is it altering just the name of a "descriptor" attribute? This is just a simplification for this example, but, as it transpires when working with descriptors, it doesn't know at this point the name of the parameter it was assigned to, so we just used the one from the example, knowing that it was going to be "descriptor".

In a real example, you would do one of two things: either receive the name as a parameter and store it internally in the `init` method, so that this one will just use the internal attribute, or, even better, use the `__set_name__` method.

Why is it accessing the `__dict__` attribute of the instance directly? Another good question, which also has at least two explanations. First, you might be thinking why not just do the following: `setattr(instance, "descriptor", value)`. Remember that this method (`__set__`) is called when we try to assign something to the attribute that is a descriptor. So, using `setattr()` will call this descriptor again, which, in turn, will call it again, and so on and so forth. This will end up in an infinite recursion.

Note: Do not use `setattr()` or the assignment expression directly on the descriptor inside the `__set__` method because that will trigger an infinite recursion.

Why, then, is the descriptor not able to book-keep the values of the properties for all of its objects?

The client class already has a reference to the descriptor. If we add a reference from the descriptor to the client object, we are creating circular dependencies, and these objects will never be garbage-collected. Since they are pointing at each other, their reference counts will never drop below the threshold for removal.

A possible alternative here is to use weak references, with the `weakref` module, and create a weak reference key dictionary if we want to do that. This implementation is explained later on, but we prefer to use this idiom, since it is fairly common and accepted when writing descriptors.

6.3 3. Descriptors in action

Now that we have seen what descriptors are, how they work, and what the main ideas behind them are, we can see them in action. In this section, we will be exploring some situations that can be elegantly addressed through descriptors.

Here, we will look at some examples of working with descriptors, and we will also cover implementation considerations for them (different ways of creating them, with their pros and cons), and finally we will discuss what are the most suitable scenarios for descriptors.

6.3.1 3.1. An application of descriptors

We will start with a simple example that works, but that will lead to some code duplication. It is not very clear how this issue will be addressed. Later on, we will devise a way of abstracting the repeated logic into a descriptor, which will address the duplication problem, and we will notice that the code on our client classes will be reduced drastically.

3.1.1. A first attempt without using descriptors

The problem we want to solve now is that we have a regular class with some attributes, but we wish to track all of the different values a particular attribute has over time, for example, in a list. The first solution that comes to our mind is to use a property, and every time a value is changed for that attribute in the setter method of the property, we add it to an internal list that will keep this trace as we want it.

Imagine that our class represents a traveler in our application that has a current city, and we want to keep track of all the cities that user has visited throughout the running of the program. The following code is a possible implementation that addresses these requirements:

```python
class Traveller:
    def __init__(self, name, current_city):
        self.name = name
        self._current_city = current_city
        self._cities_visited = [current_city]

    @property
    def current_city(self):
        return self._current_city

    @current_city.setter
    def current_city(self, new_city):
        if new_city != self._current_city:
            self._cities_visited.append(new_city)
        self._current_city = new_city

    @property
    def cities_visited(self):
        return self._cities_visited
```

We can easily check that this code works according to our requirements:

```python
>>> alice = Traveller("Alice", "Barcelona")
>>> alice.current_city = "Paris"
>>> alice.current_city = "Brussels"
>>> alice.current_city = "Amsterdam"
>>> alice.cities_visited
['Barcelona', 'Paris', 'Brussels', 'Amsterdam']
```

So far, this is all we need and nothing else has to be implemented. For the purposes of this problem, the property would be more than enough. What happens if we need the exact same logic in multiple places of the application?

This would mean that this is actually an instance of a more generic problem: tracing all the values of an attribute in another one. What would happen if we want to do the same with other attributes, such as keeping track of all tickets Alice bought, or all the countries she has been in? We would have to repeat the logic in all of these places.

Moreover, what would happen if we need this same behavior in different classes? We would have to repeat the code or come up with a generic solution (maybe a decorator, a property builder, or a descriptor).

3.1.2. The idiomatic implementation

We will now look at how to address the questions of the previous section by using a descriptor that is generic enough as to be applied in any class. Again, this example is not really needed because the requirements do not specify such generic behavior (we haven't even followed the rule of three instances of the similar pattern previously creating the abstraction), but it is shown with the goal of portraying descriptors in action.

Note: Do not implement a descriptor unless there is actual evidence of the repetition we are trying to solve, and the complexity is proven to have paid off.

Now, we will create a generic descriptor that, given a name for the attribute to hold the traces of another one, will store the different values of the attribute in a list.

As we mentioned previously, the code is more than what we need for the problem, but its intention is just to show how a descriptor would help us in this case. Given the generic nature of descriptors, the reader will notice that the logic on it (the name of their method, and attributes) does not relate to the domain problem at hand (a traveler object). This is because the idea of the descriptor is to be able to use it in any type of class, probably on different projects, with the same outcomes.

In order to address this gap, some parts of the code are annotated, and the respective explanation for each section (what it does, and how it relates to the original problem) is described in the following code:

```python
class HistoryTracedAttribute:
    def __init__(self, trace_attribute_name) -> None:
        self.trace_attribute_name = trace_attribute_name # [1]
        self._name = None

    def __set_name__(self, owner, name):
        self._name = name

    def __get__(self, instance, owner):
        if instance is None:
            return self

        return instance.__dict__[self._name]

    def __set__(self, instance, value):
        self._track_change_in_value_for_instance(instance, value)
        instance.__dict__[self._name] = value

    def _track_change_in_value_for_instance(self, instance, value):
        self._set_default(instance) # [2]
        if self._needs_to_track_change(instance, value):
            instance.__dict__[self.trace_attribute_name].append(value)

    def _needs_to_track_change(self, instance, value) -> bool:
        try:
            current_value = instance.__dict__[self._name]
        except KeyError: # [3]
            return True

        return value != current_value # [4]
```

(continues on next page)

(continued from previous page)

```
    def _set_default(self, instance):
        instance.__dict__.setdefault(self.trace_attribute_name, []) # [6]

class Traveller:
    current_city = HistoryTracedAttribute("cities_visited") # [1]

    def __init__(self, name, current_city):
        self.name = name
        self.current_city = current_city # [5]
```

Some annotations and comments on the code are as follows (numbers in the list correspond to the number annotations in the previous listing):

1. The name of the attribute is one of the variables assigned to the descriptor, in this case, `current_city`. We pass to the descriptor the name of the variable in which it will store the trace for the variable of the descriptor. In this example, we are telling our object to keep track of all the values that `current_city` has had in the attribute named `cities_visited`.
2. The first time we call the descriptor, in the `init`, the attribute for tracing values will not exist, in which case we initialize it to an empty list to later append values to it.
3. In the `init` method, the name of the attribute `current_city` will not exist either, so we want to keep track of this change as well. This is the equivalent of initializing the list with the first value in the previous example.
4. Only track changes when the new value is different from the one that is currently set.
5. In the `init method`, the descriptor already exists, and this assignment instruction triggers the actions from step 2 (create the empty list to start tracking values for it), and step 3 (append the value to this list, and set it to the key in the object for retrieval later).
6. The `setdefault` method in a dictionary is used to avoid a `KeyError`. In this case an empty list will be returned for those attributes that aren't still available.

It is true that the code in the descriptor is rather complex. On the other hand, the code in the client class is considerably simpler. Of course, this balance only pays off if we are going to use this descriptor multiple times, which is a concern we have already covered.

What might not be so clear at this point is that the descriptor is indeed completely independent from the client class. Nothing in it suggests anything about the business logic. This makes it perfectly suitable to apply it in any other class; even if it does something completely different, the descriptor will take the same effect.

This is the true Pythonic nature of descriptors. They are more appropriate for defining libraries, frameworks, or internal APIs, and not that much for business logic.

6.3.2 3.2. Different forms of implementing descriptors

We have to first understand a common issue that's specific to the nature of descriptors before thinking of ways of implementing them. First, we will discuss the problem of a global shared state, and afterward we will move on and look at different ways descriptors can be implemented while taking this into consideration.

6.3.3 3.2.1. The issue of global shared state

As we have already mentioned, descriptors need to be set as class attributes to work. This should not be a problem most of the time, but it does come with some warnings that need to be taken into consideration.

The problem with class attributes is that they are shared across all instances of that class. Descriptors are not an exception here, so if we try to keep data in a descriptor object, keep in mind that all of them will have access to the same value.

Let's see what happens when we incorrectly define a descriptor that keeps the data itself, instead of storing it in each object:

```
class SharedDataDescriptor:
    def __init__(self, initial_value):
        self.value = initial_value

    def __get__(self, instance, owner):
        if instance is None:
            return self
        return self.value

    def __set__(self, instance, value):
        self.value = value

    class ClientClass:
        descriptor = SharedDataDescriptor("first value")
```

In this example, the descriptor object stores the data itself. This carries with it the inconvenience that when we modify the value for an instance all other instances of the same classes are also modified with this value as well. The following code listing puts that theory in action:

```
>>> client1 = ClientClass()
>>> client1.descriptor
'first value'
>>> client2 = ClientClass()
>>> client2.descriptor
'first value'
>>> client2.descriptor = "value for client 2"
>>> client2.descriptor
'value for client 2'
>>> client1.descriptor
'value for client 2'
```

Notice how we change one object, and suddenly all of them are from the same class, and we can see that this value is reflected. This is because `ClientClass.descriptor` is unique; it's the same object for all of them.

In some cases, this might be what we actually want (for instance, if we were to create a sort of Borg pattern implementation, on which we want to share state across all objects from a class), but in general, that is not the case, and we need to differentiate between objects.

To achieve this, the descriptor needs to know the value for each instance and return it accordingly. That is the reason we have been operating with the dictionary (`__dict__`) of each instance and setting and retrieving the values from there.

This is the most common approach. We have already covered why we cannot use `getattr()` and `setattr()` on those methods, so modifying the `__dict__` attribute is the last standing option, and, in this case, is acceptable.

3.2.2. Accessing the dictionary of the object

The way we implement descriptors throughout this book is making the descriptor object store the values in the dictionary of the object, `__dict__`, and retrieve the parameters from there as well.

Note: Always store and return the data from the `__dict__` attribute of the instance.

3.2.3. Using weak references

Another alternative (if we don't want to use `__dict__`) is to make the descriptor object keep track of the values for each instance itself, in an internal mapping, and return values from this mapping as well.

There is a caveat, though. This mapping cannot just be any dictionary. Since the client class has a reference to the descriptor, and now the descriptor will keep references to the objects that use it, this will create circular dependencies, and, as a result, these objects will never be garbage-collected because they are pointing at each other.

In order to address this, the dictionary has to be a weak key one, as defined in the `weakref` module.

In this case, the code for the descriptor might look like the following:

```python
from weakref import WeakKeyDictionary

class DescriptorClass:
    def __init__(self, initial_value):
        self.value = initial_value
        self.mapping = WeakKeyDictionary()

    def __get__(self, instance, owner):
        if instance is None:
            return self
        return self.mapping.get(instance, self.value)

    def __set__(self, instance, value):
        self.mapping[instance] = value
```

This addresses the issues, but it does come with some considerations:

- The objects no longer hold their attributes: the descriptor does instead. This is somewhat controversial, and it might not be entirely accurate from a conceptual point of view. If we forget this detail, we might be asking the object by inspecting its dictionary, trying to find things that just aren't there (calling `vars(client)` will not return the complete data, for example).
- It poses the requirement over the objects that they need to be hashable. If they aren't, they can't be part of the mapping. This might be too demanding a requirement for some applications.

For these reasons, we prefer the implementation that has been shown so far in this book, which uses the dictionary of each instance. However, for completeness, we have shown this alternative as well.

6.3.4 3.3. More considerations about descriptors

Here, we will discuss general considerations about descriptors in terms of what we can do with them when it is a good idea to use them, and also how things that we might have initially conceived as having been resolved by means of another approach can be improved through descriptors. We will then analyze the pros and cons of the original implementation versus the one after descriptors have been used.

3.3.1. Reusing code

Descriptors are a generic tool and a powerful abstraction that we can use to avoid code duplication. The best way to decide when to use descriptors is to identify cases where we would be using a property (whether for its get logic, set logic, or both), but repeating its structure many times.

Properties are just a particular case of descriptors (the @property decorator is a descriptor that implements the full descriptor protocol to define their get , set , and delete actions), which means that we can use descriptors for far more complex tasks.

Another powerful type we have seen for reusing code was decorators. Descriptors can help us create to better decorators by making sure that they will be able to work correctly for class methods as well.

When it comes to decorators, we could say that it is safe to always implement the `__get__()` method on them, and also make it a descriptor. When trying to decide whether the decorator is worth creating, consider the problems but note that there are no extra considerations toward descriptors.

As for generic descriptors, besides the aforementioned three instances rule that applies to decorators (and, in general, any reusable component), it is advisable to also keep in mind that you should use descriptors for cases when we want to define an internal API, which is some code that will have clients consuming it. This is a feature-oriented more toward designing libraries and frameworks, rather than one-time solutions.

Unless there is a very good reason to, or that the code will look significantly better, we should avoid putting business logic in a descriptor. Instead, the code of a descriptor will contain more implementational code rather than business code. It is more similar to defining a new data structure or object that another part of our business logic will use as a tool.

Note: In general, descriptors will contain implementation logic, and not so much business logic.

3.3.2. Avoiding class decorators

If we recall the class decorator we used previously to determine how an event object is going to be serialized, we ended up with an implementation that (for Python 3.7+) relied on two class decorators:

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

The first one takes the attributes from the annotations to declare the variables, whereas the second one defines how to treat each file. Let's see whether we can change these two decorators for descriptors instead.

The idea is to create a descriptor that will apply the transformation over the values of each attribute, returning the modified version according to our requirements (for example, hiding sensitive information, and formatting dates correctly):

```python
from functools import partial
from typing import Callable


class BaseFieldTransformation:
    def __init__(self, transformation: Callable[[], str]) -> None:
        self._name = None
        self.transformation = transformation

    def __get__(self, instance, owner):
        if instance is None:
            return self

        raw_value = instance.__dict__[self._name]
        return self.transformation(raw_value)

    def __set_name__(self, owner, name):
        self._name = name

    def __set__(self, instance, value):
        instance.__dict__[self._name] = value
        ShowOriginal = partial(BaseFieldTransformation, transformation=lambda x: x)
        HideField = partial(
            BaseFieldTransformation, transformation=lambda x: "**redacted**"
        )
        FormatTime = partial(
            BaseFieldTransformation,
            transformation=lambda ft: ft.strftime("%Y-%m-%d %H:%M"),
        )
```

This descriptor is interesting. It was created with a function that takes one argument and returns one value. This function will be the transformation we want to apply to the field. From the base definition that defines generically how it is going to work, the rest of the descriptor classes are defined, simply by changing the particular function each one needs.

The example uses `functools.partial` as a way of simulating sub-classes, by applying a partial application of the transformation function for that class, leaving a new callable that can be instantiated directly.

In order to keep the example simple, we will implement the `__init__()` and `serialize()` methods, although they could be abstracted away as well. Under these considerations, the class for the event will now be defined as follows:

```python
class LoginEvent:

    username = ShowOriginal()
    password = HideField()
    ip = ShowOriginal()
    timestamp = FormatTime()

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

    def serialize(self):
        return {
            "username": self.username,
            "password": self.password,
            "ip": self.ip,
            "timestamp": self.timestamp,
        }
```

We can see how the object behaves at runtime:

```
>>> le = LoginEvent("john", "secret password", "1.1.1.1",
datetime.utcnow())
>>> vars(le)
{'username': 'john', 'password': 'secret password', 'ip': '1.1.1.1',
'timestamp': ...}
>>> le.serialize()
{'username': 'john', 'password': '**redacted**', 'ip': '1.1.1.1',
'timestamp': '...'}
>>> le.password
'**redacted**'
```

There are some differences with respect to the previous implementation that used a decorator. This example added the `serialize()` method and hid the fields before presenting them to its resulting dictionary, but if we asked for any of these attributes to an instance of the event in memory at any point, it would still give us the original value, without any transformation applied to it (we could have chosen to apply the transformation when setting the value, and return it directly on the `__get__()` , as well).

Depending on the sensitivity of the application, this may or may not be acceptable, but in this case, when we ask the object for its public attributes, the descriptor will apply the transformation before presenting the results. It is still possible to access the original values by asking for the dictionary of the object (by accessing `__dict__`), but when we ask for the value, by default, it will return it converted.

In this example, all descriptors follow a common logic, which is defined in the base class. The descriptor should store the value in the object and then ask for it, applying the transformation it defines. We could create a hierarchy of classes, each one defining its own conversion function, in a way that the template method design pattern works. In this case, since the changes in the derived classes are relatively small (just one function), we opted for creating the derived classes as partial applications of the base class. Creating any new transformation field should be as simple as defining a new class that will be the base class, which is partially applied with the function we need. This can even be done ad hoc, so there might be no need to set a name for it.

Regardless of this implementation, the point is that since descriptors are objects, we can create models, and apply all rules of object-oriented programming to them. Design patterns also apply to descriptors. We could define our hierarchy, set the custom behavior, and so on. This example follows the OCP, because adding a new type of conversion method would just be about creating a new class, derived from the base one with the function it needs, without having to modify the base class itself (to be fair, the previous implementation with decorators was also OCP- compliant, but there were no classes involved for each transformation mechanism).

Let's take an example where we create a base class that implements the `__init__()``and ``serialize()` methods so that we can define the `LoginEvent` class simply by deriving from it, as follows:

```python
class LoginEvent(BaseEvent):
    username = ShowOriginal()
    password = HideField()
    ip = ShowOriginal()
    timestamp = FormatTime()
```

Once we achieve this code, the class looks cleaner. It only defines the attributes it needs, and its logic can be quickly analyzed by looking at the class for each attribute. The base class will abstract only the common methods, and the class of each event will look simpler and more compact.

Not only do the classes for each event look simple, but the descriptor itself is very compact and a lot simpler than the class decorators. The original implementation with class decorators was good, but descriptors made it even better.

6.4 4. Analysis of descriptors

We have seen how descriptors work so far and explored some interesting situations in which they contribute to clean design by simplifying their logic and leveraging more compact classes.

Up to this point, we know that by using descriptors, we can achieve cleaner code, abstracting away repeated logic and implementation details. But how do we know our implementation of the descriptors is clean and correct? What makes a good descriptor? Are we using this tool properly or over-engineering with it?

6.4.1 4.1. How Python uses descriptors internally

Referring to the question as to what makes a good descriptor?, a simple answer would be that a good descriptor is pretty much like any other good Python object. It is consistent with Python itself. The idea that follows this premise is that analyzing how Python uses descriptors will give us a good idea of good implementations so that we know what to expect from the descriptors we write.

We will see the most common scenarios where Python itself uses descriptors to solve parts of its internal logic, and we will also discover elegant descriptors and that they have been there in plain sight all along.

4.1.1. Functions and methods

The most resonating case of an object that is a descriptor is probably a function. Functions implement the `__get__` method, so they can work as methods when defined inside a class. Methods are just functions that take an extra argument. By convention, the first argument of a method is named “self”, and it represents an instance of the class that the method is being defined in. Then, whatever the method does with “self”, would be the same as any other function receiving the object and applying modifications to it.

In order words, when we define something like this:

```
class MyClass:
    def method(self, ...):
        self.x = 1
```

It is actually the same as if we define this:

```
class MyClass:
    pass

def method(myclass_instance, ...):
    myclass_instance.x = 1
    method(MyClass())
```

So, it is just another function, modifying the object, only that it’s defined inside the class, and it is said to be bound to the object.

When we call something in the form of this:

```
instance = MyClass()
instance.method(...)
```

Python is, in fact, doing something equivalent to this:

```
instance = MyClass()
MyClass.method(instance, ...)
```

Notice that this is just a syntax conversion that is handled internally by Python. The way this works is by means of descriptors.

Since functions implement the descriptor protocol (see the following listing) before calling the method, the `__get__()` method is invoked first, and some transformations happen before running the code on the internal callable:

```
>>> def function(): pass
...
>>> function.__get__
<method-wrapper '__get__' of function object at 0x...>
```

In the `instance.method(...)` statement, before processing all the arguments of the callable inside the parenthesis, the "instance.method" part is evaluated.

Since `method` is an object defined as a class attribute, and it has a __get__ method, this is called. What this does is convert the function to a method, which means binding the callable to the instance of the object it is going to work with.

Let's see this with an example so that we can get an idea of what Python might be doing internally.

We will define a callable object inside a class that will act as a sort of function or method that we want to define to be invoked externally. An instance of the `Method` class is supposed to be a function or method to be used inside a different class. This function will just print its three parameters: the instance that it received (which would be the self parameter on the class it's being defined in), and two more arguments. Notice that in the `__call__()` method, the self parameter does not represent the instance of `MyClass`, but instead an instance of `Method`. The parameter named instance is meant to be a `MyClass` type of object:

```
class Method:
    def __init__(self, name):
        self.name = name

    def __call__(self, instance, arg1, arg2):
        print(f"{self.name}: {instance} called with {arg1} and {arg2}")

class MyClass:
    method = Method("Internal call")
```

Under these considerations and, after creating the object, the following two calls should be equivalent, based on the preceding definition:

```
instance = MyClass()
Method("External call")(instance, "first", "second")
instance.method("first", "second")
```

However, only the first one works as expected, as the second one gives an error:

```
Traceback (most recent call last):
File "file", line , in <module>
instance.method("first", "second")
TypeError: __call__() missing 1 required positional argument: 'arg2'
```

We are seeing the same error we faced with a decorator. The arguments are being shifted to the left by one, instance is taking the place of `self`, `arg1` is going to be instance, and there is nothing to provide for `arg2`.

In order to fix this, we need to make `Method` a descriptor.

This way, when we call `instance.method` first, we are going to call its `__get__()`, on which we bind this callable to the object accordingly (bypassing the object as the first parameter), and then proceed:

```
from types import MethodType

class Method:
    def __init__(self, name):
        self.name = name

    def __call__(self, instance, arg1, arg2):
        print(f"{self.name}: {instance} called with {arg1} and {arg2}")

    def __get__(self, instance, owner):
```

(continues on next page)

(continued from previous page)

```
        if instance is None:
            return self

        return MethodType(self, instance)
```

Now, both calls work as expected:

```
External call: <MyClass object at 0x...> called with fist and second
Internal call: <MyClass object at 0x...> called with first and second
```

What we did is convert the function (actually the callable object we defined instead) to a method by using `MethodType` from the `types` module. The first parameter of this class should be a callable (`self`, in this case, is one by definition because it implements `__call__`), and the second one is the object to bind this function to.

Something similar to this is what function objects use in Python so they can work as methods when they are defined inside a class.

Since this is a very elegant solution, it's worth exploring it to keep it in mind as a Pythonic approach when defining our own objects. For instance, if we were to define our own callable, it would be a good idea to also make it a descriptor so that we can use it in classes as class attributes as well.

4.1.2. Built-in decorators for methods

All `@property` , `@classmethod`, and `@staticmethod` decorators are descriptors.

We have mentioned several times that the idiom makes the descriptor return itself when it's being called from a class directly. Since properties are actually descriptors, that is the reason why, when we ask it from the class, we don't get the result of computing the property, but the entire property object instead:

```
>>> class MyClass:
...     @property
...     def prop(self): pass
...
>>> MyClass.prop
<property object at 0x...>
```

For class methods, the `__get__` function in the descriptor will make sure that the class is the first parameter to be passed to the function being decorated, regardless of whether it's called from the class directly or from an instance. For static methods, it will make sure that no parameters are bound other than those defined by the function, namely undoing the binding done by `__get__()` on functions that make self the first parameter of that function.

Let's take an example; we create a `@classproperty` decorator that works as the regular `@property` decorator, but for classes instead. With a decorator like this one, the following code should be able to work:

```
class TableEvent:
    schema = "public"
    table = "user"

    @classproperty
    def topic(cls):
        prefix = read_prefix_from_config()
        return f"{prefix}{cls.schema}.{cls.table}"


>>> TableEvent.topic
'public.user'
>>> TableEvent().topic
'public.user'
```

4.1.3. Slots

When a class defines the `__slots__` attribute, it can contain all the attributes that the class expects and no more.

Trying to add extra attributes dynamically to a class that defines `__slots__` will result in an `AttributeError`. By defining this attribute, the class becomes static, so it will not have a `__dict__` attribute where you can add more objects dynamically.

How, then, are its attributes retrieved if not from the dictionary of the object? By using descriptors. Each name defined in a slot will have its own descriptor that will store the value for retrieval later:

```
class Coordinate2D:
    __slots__ = ("lat", "lon")
    def __init__(self, lat, lon):
        self.lat = lat
        self.lon = lon

    def __repr__(self):
        return f"{self.__class__.__name__}({self.lat}, {self.lon})"
```

While this is an interesting feature, it has to be used with caution because it is taking away the dynamic nature of Python. In general, this ought to be reserved only for objects that we know are static, and if we are absolutely sure we are not adding any attributes to them dynamically in other parts of the code.

As an upside of this, objects defined with slots use less memory, since they only need a fixed set of fields to hold values and not an entire dictionary.

6.4.2 4.2. Implementing descriptors in decorators

We now understand how Python uses descriptors in functions to make them work as methods when they are defined inside a class. We have also seen examples of cases where we can make decorators work by making them comply with the descriptor protocol by using the `__get__()` method of the interface to adapt the decorator to the object it is being called with. This solves the problem for our decorators in the same way that Python solves the issue of functions as methods in objects.

The general recipe for adapting a decorator in such a way is to implement the `__get__()` method on it and use `types.MethodType` to convert the callable (the decorator itself) to a method bound to the object it is receiving (the instance parameter received by `__get__`).

For this to work, we will have to implement the decorator as an object, because otherwise, if we are using a function, it will already have a `__get__()` method, which will be doing something different that will not work unless we adapt it. The cleaner way to proceed is to define a class for the decorator.

Note: Use a decorator class when defining a decorator that we want to apply to class methods, and implement the `__get__()` method on it.

CHAPTER

SEVEN

USING GENERATORS

7.1 1. Creating generators

Generators were introduced in Python a long time ago (PEP-255), with the idea of introducing iteration in Python while improving the performance of the program (by using less memory) at the same time.

The idea of a generator is to create an object that is iterable, and, while it's being iterated, will produce the elements it contains, one at a time. The main use of generators is to save memory: instead of having a very large list of elements in memory, holding everything at once, we have an object that knows how to produce each particular element, one at a time, as they are required.

This feature enables lazy computations or heavyweight objects in memory, in a similar manner to what other functional programming languages (Haskell, for instance) provide. It would even be possible to work with infinite sequences because the lazy nature of generators allows for such an option.

7.1.1 1.1. A first look at generators

Let's start with an example. The problem at hand now is that we want to process a large list of records and get some metrics and indicators over them. Given a large data set with information about purchases, we want to process it in order to get the lowest sale, highest sale, and the average price of a sale.

For the simplicity of this example, we will assume a CSV with only two fields, in the following format:

```
<purchase_date>, <price>
...
```

We are going to create an object that receives all the purchases, and this will give us the necessary metrics. We could get some of these values out of the box by simply using the `min()` and `max()` built-in functions, but that would require iterating all of the purchases more than once, so instead, we are using our custom object, which will get these values in a single iteration.

The code that will get the numbers for us looks rather simple. It's just an object with a method that will process all prices in one go, and, at each step, will update the value of each particular metric we are interested in. First, we will show the first implementation in the following listing, and, later on (once we have seen more about iteration), we will revisit this implementation and get a much better (and compact) version of it. For now, we are settling on the following:

```python
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):
```

(continues on next page)

(continued from previous page)

```
        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})"
        )
```

This object will receive all the totals for the `purchases` and process the required values. Now, we need a function that loads these numbers into something that this object can process. Here is the first version:

```
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
```

This code works; it loads all the numbers of the file into a list that, when passed to our custom object, will produce the numbers we want. It has a performance issue, though. If you run it with a rather large dataset, it will take a while to complete, and it might even fail if the dataset is large enough as to not fit into the main memory.

If we take a look at our code that consumes this data, it is processing the `purchases`, one at a time, so we might be wondering why our producer fits everything in memory at once. It is creating a list where it puts all of the content of the file, but we know we can do better.

The solution is to create a generator. Instead of loading the entire content of the file in a list, we will produce the results one at a time. The code will now look like this:

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

If you measure the process this time, you will notice that the usage of memory has dropped significantly. We can also see how the code looks simpler: there is no need to define the list (therefore, there is no need to append to it), and that the return statement also disappeared.

In this case, the `load_purchases` function is a generator function, or simply a generator.

In Python, the mere presence of the keyword `yield` in any function makes it a generator, and, as a result, when calling it, nothing other than creating an instance of the generator will happen:

```
>>> load_purchases("file")
<generator object load_purchases at 0x...>
```

A generator object is an iterable (we will revisit iterables in more detail later on), which means that it can work with for loops. Notice how we did not have to change anything on the consumer code: our statistics processor remained the same, with the for loop unmodified, after the new implementation.

Working with iterables allows us to create these kinds of powerful abstractions that are polymorphic with respect to for loops. As long as we keep the iterable interface, we can iterate over that object transparently.

7.1.2 1.2. Generator expressions

Generators save a lot of memory, and since they are iterators, they are a convenient alternative to other iterables or containers that require more space in memory such as lists, tuples, or sets.

Much like these data structures, they can also be defined by comprehension, only that it is called a generator expression (there is an ongoing argument about whether they should be called generator comprehensions).

In the same way, we would define a list comprehension. If we replace the square brackets with parenthesis, we get a generator that results from the expression. Generator expressions can also be passed directly to functions that work with iterables, such as `sum()`, and, `max()`:

```
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> (x**2 for x in range(10))
<generator object <genexpr> at 0x...>
>>> sum(x**2 for x in range(10))
285
```

Note: Always pass a generator expression, instead of a list comprehension, to functions that expect iterables, such as `min()`, `max()`, and `sum()`. This is more efficient and pythonic.

It is also worth mentioning, that we can only iterate 1 time over generators:

```
>>> a = (x for x in range(3))
>>> a
<generator object <genexpr> at 0x7f95ece4dad0>
>>> for x in a:
...     print(x)
...
0
1
2
```

(continues on next page)

(continued from previous page)

```
>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

7.2 2. Iterating idiomatically

In this section, we will first explore some idioms that come in handy when we have to deal with iteration in Python. These code recipes will help us get a better idea of the types of things we can do with generators (especially after we have already seen generator expressions), and how to solve typical problems in relation to them.

Once we have seen some idioms, we will move on to exploring iteration in Python in more depth, analyzing the methods that make iteration possible, and how iterable objects work.

7.2.1 2.1. Idioms for iteration

We are already familiar with the built-in `enumerate()` function that, given an iterable, will return another one on which the element is a tuple, whose first element is the enumeration of the second one (corresponding to the element in the original iterable):

```
>>> list(enumerate("abcdef"))
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e'), (5, 'f')]
```

We wish to create a similar object, but in a more low-level fashion; one that can simply create an infinite sequence. We want an object that can produce a sequence of numbers, from a starting one, without any limits.

An object as simple as the following one can do the trick. Every time we call this object, we get the next number of the sequence *ad infinitum*:

```
class NumberSequence:
    def __init__(self, start=0):
        self.current = start

    def next(self):
        current = self.current
        self.current += 1
        return current
```

Based on this interface, we would have to use this object by explicitly invoking its next() method:

```
>>> seq = NumberSequence()
>>> seq.next()
0
>>> seq.next()
1
>>> seq2 = NumberSequence(10)
>>> seq2.next()
10
>>> seq2.next()
11
```

But with this code, we cannot reconstruct the `enumerate()` function as we would like to, because its interface does not support being iterated over a regular Python for loop, which also means that we cannot pass it as a parameter to functions that expect something to iterate over. Notice how the following code fails:

```
>>> list(zip(NumberSequence(), "abcdef"))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
File "...", line 1, in <module>
TypeError: zip argument #1 must support iteration
```

The problem lies in the fact that `NumberSequence` does not support iteration. To fix this, we have to make the object an iterable by implementing the magic method `__iter__()`. We have also changed the previous `next()` method, by using the magic method `__next__`, which makes the object an iterator:

```python
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
```

This has an advantage: not only can we iterate over the element, we also don't even need the `next()` method any more because having `__next__()` allows us to use the `next()` built-in function:

```
>>> list(zip(SequenceOfNumbers(), "abcdef"))
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e'), (5, 'f')]
>>> seq = SequenceOfNumbers(100)
>>> next(seq)
100
>>> next(seq)
101
```

2.1.1. The next() function

The `next()` built-in function will advance the iterable to its next element and return it:

```
>>> word = iter("hello")
>>> next(word)
'h'
>>> next(word)
'e'
```

If the iterator does not have more elements to produce, the `StopIteration` exception is raised:

```
>>> ...
>>> next(word)
'o'
>>> next(word)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
```

This exception signals that the iteration is over and that there are no more elements to consume.

If we wish to handle this case, besides catching the `StopIteration` exception, we could provide this function with a default value in its second parameter. Should this be provided, it will be the return value in lieu of throwing `StopIteration`:

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

2.1.2. Using a generator

The previous code can be simplified significantly by simply using a generator. Generator objects are iterators. This way, instead of creating a class, we can define a function that `yield` the values as needed:

```
def sequence(start=0):
    while True:
        yield start
        start += 1
```

Remember that from our first definition, the `yield` keyword in the body of the function makes it a generator. Because it is a generator, it's perfectly fine to create an infinite loop like this, because, when this generator function is called, it will run all the code until the next `yield` statement is reached. It will produce its value and suspend there:

```
>>> 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')]
```

2.1.3. Itertools

Working with iterables has the advantage that the code blends better with Python itself because iteration is a key component of the language. Besides that, we can take full advantage of the itertools module. Actually, the `sequence()` generator we just created is fairly similar to `itertools.count()`. However, there is more we can do.

One of the nicest things about iterators, generators, and itertools, is that they are composable objects that can be chained together.

For instance, getting back to our first example that processed purchases in order to get some metrics, what if we want to do the same, but only for those values over a certain threshold? The naive approach of solving this problem would be to place the condition while iterating:

```
def process(self):
    for purchase in self.purchases:
        if purchase > 1000.0:
            ...
```

This is not only non-Pythonic, but it's also rigid (and rigidity is a trait that denotes bad code). It doesn't handle changes very well. What if the number changes now? Do we pass it by parameter? What if we need more than one? What if the condition is different (less than, for instance)? Do we pass a lambda?

These questions should not be answered by this object, whose sole responsibility is to compute a set of well-defined metrics over a stream of purchases represented as numbers. And, of course, the answer is no. It would be a huge mistake to make such a change (once again, clean code is flexible, and we don't want to make it rigid by coupling this object to external factors). These requirements will have to be addressed elsewhere.

It's better to keep this object independent of its clients. The less responsibility this class has, the more useful it will be for more clients, hence enhancing its chances of being reused.

Instead of changing this code, we're going to keep it as it is and assume that the new data is filtered according to whatever requirements each customer of the class has.

For instance, if we wanted to process only the first 10 purchases that amount to more than 1,000 , we would do the following:

```
>>> from itertools import islice
>>> purchases = islice(filter(lambda p: p > 1000.0, purchases), 10)
>>> stats = PurchasesStats(purchases).process()
```

There is no memory penalization for filtering this way because since they all are generators, the evaluation is always lazy. This gives us the power of thinking as if we had filtered the entire set at once and then passed it to the object, but without actually fitting everything in memory.

2.1.4. Simplifying code through iterators

Now, we will briefly discuss some situations that can be improved with the help of iterators, and occasionally the itertools module. After discussing each case, and its proposed optimization, we will close each point with a corollary.

2.1.4.1. Repeated iterations

Now that we have seen more about iterators, and introduced the itertools module, we can show you how one of the first examples of this chapter (the one for computing statistics about some purchases), can be dramatically simplified:

```
def process_purchases(purchases):
    min_iter, max_iter, avg_iter = itertools.tee(purchases, 3)
    return min(min_iter), max(max_iter), median(avg_iter)
```

In this example, `itertools.tee` will split the original iterable into three new ones. We will use each of these for the different kinds of iterations that we require, without needing to repeat three different loops over purchases.

The reader can simply verify that if we pass an iterable object as the purchases parameter, this one is traversed only once (thanks to the itertools.tee function), which was our main requirement. It is also possible to verify how this version is equivalent to our original implementation. In this case, there is no need to manually raise `ValueError` because passing an empty sequence to the `min()` function will do the same.

Note: If you are thinking about running a loop over the same object more than one time, stop and think if itertools.tee can be of any help.

2.1.4.2. Nested loops

In some situations, we need to iterate over more than one dimension, looking for a value, and nested loops come as the first idea. When the value is found, we need to stop iterating, but the `break` keyword doesn't work entirely because we have to escape from two (or more) for loops, not just one.

What would be the solution for this? A flag signaling escape? No. Raising an exception? No, this would be the same as the flag, but even worse because we know that exceptions are not to be used for control flow logic. Moving the code to a smaller function and return it? Close, but not quite.

The answer is, whenever possible, flat the iteration to a single for loop. This is the kind of code we would like to avoid:

```
def search_nested_bad(array, desired_value):
    coords = None
    for i, row in enumerate(array):
        for j, cell in enumerate(row):
            if cell == desired_value:
                coords = (i, j)
                break
        if coords is not None:
```

(continues on next page)

(continued from previous page)

```
                break

    if coords is None:
        raise ValueError(f"{desired_value} not found")

    logger.info("value %r found at [%i, %i]", desired_value, *coords)
    return coords
```

And here is a simplified version of it that does not rely on flags to signal termination, and has a simpler, more compact structure of iteration:

```
def _iterate_array2d(array2d):
    for i, row in enumerate(array2d):
        for j, cell in enumerate(row):
            yield (i, j), cell

def search_nested(array, desired_value):
    try:
        coord = next(coord for (coord, cell) in _iterate_array2d(array) if cell ==␣
↪desired_value)
    except StopIteration:
        raise ValueError(f"{desired_value} not found")

    logger.info(f"value {desired_value} found at {coords}")
    return coord
```

It's worth mentioning how the auxiliary generator that was created works as an abstraction for the iteration that's required. In this case, we just need to iterate over two dimensions, but if we needed more, a different object could handle this without the client needing to know about it. This is the essence of the iterator design pattern, which, in Python, is transparent, since it supports iterator objects automatically, which is the topic covered in the next section.

Note: Try to simplify the iteration as much as possible with as many abstractions as are required, flatting the loops whenever possible.

7.2.2 2.2. The iterator pattern in Python

Here, we will take a small detour from generators to understand iteration in Python more deeply. Generators are a particular case of iterable objects, but iteration in Python goes beyond generators, and being able to create good iterable objects will give us the chance to create more efficient, compact, and readable code.

In the previous code listings, we have been seeing examples of iterable objects that are also iterators, because they implement both the `__iter__()` and `__next__()` magic methods. While this is fine in general, it's not strictly required that they always have to implement both methods, and here we'll show the subtle differences between an iterable object (one that implements `__iter__`) and an iterator (that implements `__next__`).

We also explore other topics related to iterations, such as sequences and container objects.

2.2.1. The interface for iteration

An iterable is an object that supports iteration, which, at a very high level, means that we can run a `for .. in ...` loop over it, and it will work without any issues. However, iterable does not mean the same as iterator.

Generally speaking, an iterable is just something we can iterate, and it uses an iterator to do so. This means that in the `__iter__` magic method, we would like to return an iterator, namely, an object with a `__next__()` method implemented.

An iterator is an object that only knows how to produce a series of values, one at a time, when it's being called by the already explored built-in `next()` function. While the iterator is not called, it's simply frozen, sitting idly by until it's called again for the next value to produce. In this sense, generators are iterators.

In the following code, we will see an example of an iterator object that is not iterable: it only supports invoking its values, one at a time. Here, the name sequence refers just to a series of consecutive numbers, not to the sequence concept in Python, which will we explore later on:

```
class SequenceIterator:
    def __init__(self, start=0, step=1):
        self.current = start
        self.step = step

    def __next__(self):
        value = self.current
        self.current += self.step
        return value
```

Notice that we can get the values of the sequence one at a time, but we can't iterate over this object (this is fortunate because it would otherwise result in an endless loop):

```
>>> si = SequenceIterator(1, 2)
>>> next(si)
1
>>> next(si)
3
>>> next(si)
5
>>> for _ in SequenceIterator(): pass
...
Traceback (most recent call last):
...
TypeError: 'SequenceIterator' object is not iterable
```

The error message is clear, as the object doesn't implement `__iter__()`.

Just for explanatory purposes, we can separate the iteration in another object (again, it would be enough to make the object implement both `__iter__` and `__next__`, but doing so separately will help clarify the distinctive point we're trying to make in this explanation).

2.2.2. Sequence objects as iterables

As we have just seen, if an object implements the `__iter__()` magic method, it means it can be used in a for loop. While this is a great feature, it's not the only possible form of iteration we can achieve. When we write a for loop, Python will try to see if the object we're using implements `__iter__`, and, if it does, it will use that to construct the iteration, but if it doesn't, there are fallback options.

If the object happens to be a sequence (meaning that it implements `__getitem__()` and `__len__()` magic methods), it can also be iterated. If that is the case, the interpreter will then provide values in sequence, until the `IndexError` exception is raised, which, analogous to the aforementioned `StopIteration`, also signals the stop for the iteration.

With the sole purpose of illustrating such a behavior, we run the following experiment that shows a sequence object that implements `map()` over a range of numbers:

```
class MappedRange:
    """Apply a transformation to a range of numbers."""
    def __init__(self, transformation, start, end):
        self._transformation = transformation
        self._wrapped = range(start, end)

    def __getitem__(self, index):
        value = self._wrapped.__getitem__(index)
        result = self._transformation(value)
        logger.info(f"Index {index}: {result}")
        return result

    def __len__(self):
        return len(self._wrapped)
```

Keep in mind that this example is only designed to illustrate that an object such as this one can be iterated with a regular for loop. There is a logging line placed in the `__getitem__` method to explore what values are passed while the object is being iterated, as we can see from the following test:

```
>>> mr = MappedRange(abs, -10, 5)
>>> mr[0]
Index 0: 10
10
>>> mr[-1]
Index -1: 4
4
>>> list(mr)
Index 0: 10
Index 1: 9
Index 2: 8
Index 3: 7
Index 4: 6
Index 5: 5
Index 6: 4
Index 7: 3
Index 8: 2
Index 9: 1
Index 10: 0
Index 11: 1
Index 12: 2
Index 13: 3
Index 14: 4
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 1, 2, 3, 4]
```

As a word of caution, it's important to highlight that while it is useful to know this, it's also a fallback mechanism for when the object doesn't implement `__iter__`, so most of the time we'll want to resort to these methods by thinking in creating proper sequences, and not just objects we want to iterate over.

Note: When thinking about designing an object for iteration, favor a proper iterable object (with `__iter__`), rather than a sequence that can coincidentally also be iterated.

7.3 3. Coroutines

As we already know, generator objects are iterables. They implement `__iter__()` and `__next__()`. This is provided by Python automatically so that when we create a generator object function, we get an object that can be iterated or advanced through the `next()` function.

Besides this basic functionality, they have more methods so that they can work as coroutines. Here, we will explore how generators evolved into coroutines to support the basis of asynchronous programming before we go into more detail in the next section, where we explore the new features of Python and the syntax that covers programming asynchronously. The basic methods added to support coroutines are as follows:

- `.close()`
- `.throw(ex_type[, ex_value[, ex_traceback]])`
- `.send(value)`

7.3.1 3.1. The methods of the generator interface

In this section, we will explore what each of the aforementioned methods does, how it works, and how it is expected to be used. By understanding how to use these methods, we will be able to make use of simple coroutines.

Later on, we will explore more advanced uses of coroutines, and how to delegate to sub- generators (coroutines) in order to refactor code, and how to orchestrate different coroutines.

3.1.1. close()

When calling this method, the generator will receive the `GeneratorExit` exception. If it's not handled, then the generator will finish without producing any more values, and its iteration will stop.

This exception can be used to handle a finishing status. In general, if our coroutine does some sort of resource management, we want to catch this exception and use that control block to release all resources being held by the coroutine. In general, it is similar to using a context manager or placing the code in the finally block of an exception control, but handling this exception specifically makes it more explicit.

In the following example, we have a coroutine that makes use of a database handler object that holds a connection to a database, and runs queries over it, streaming data by pages of a fixed length (instead of reading everything that is available at once):

```python
def stream_db_records(db_handler):
    try:
        while True:
            yield db_handler.read_n_records(10)
    except GeneratorExit:
        db_handler.close()
```

At each call to the generator, it will return 10 rows obtained from the database handler, but when we decide to explicitly finish the iteration and call `close()`, we also want to close the connection to the database:

```
>>> streamer = stream_db_records(DBHandler("testdb"))
>>> next(streamer)
[(0, 'row 0'), (1, 'row 1'), (2, 'row 2'), (3, 'row 3'), ...]
>>> next(streamer)
[(0, 'row 0'), (1, 'row 1'), (2, 'row 2'), (3, 'row 3'), ...]
>>> streamer.close()
INFO:...:closing connection to database 'testdb'
```

Use the `close()` method on generators to perform finishing-up tasks when needed.

3.1.2. throw(ex_type[, ex_value[, ex_traceback]])

This method will throw the exception at the line where the generator is currently suspended. If the generator handles the exception that was sent, the code in that particular except clause will be called, otherwise, the exception will propagate to the caller.

Here, we are modifying the previous example slightly to show the difference when we use this method for an exception that is handled by the coroutine, and when it's not:

```
class CustomException(Exception):
    pass

def stream_data(db_handler):
    while True:
        try:
            yield db_handler.read_n_records(10)
        except CustomException as e:
            logger.info(f"controlled error {e}, continuing")
        except Exception as e:
            logger.info(f"unhandled error {e}, stopping")
            db_handler.close()
        break
```

Now, it is a part of the control flow to receive a `CustomException`, and, in such a case, the generator will log an informative message (of course, we can adapt this according to our business logic on each case), and move on to the next `yield` statement, which is the line where the coroutine reads from the database and returns that data.

This particular example handles all exceptions, but if the last block (`except Exception:`) wasn't there, the result would be that the generator is raised at the line where the generator is paused (again, the `yield`), and it will propagate from there to the caller:

```
>>> streamer = stream_data(DBHandler("testdb"))
>>> next(streamer)
[(0, 'row 0'), (1, 'row 1'), (2, 'row 2'), (3, 'row 3'), (4, 'row 4'), ...]
>>> next(streamer)
[(0, 'row 0'), (1, 'row 1'), (2, 'row 2'), (3, 'row 3'), (4, 'row 4'), ...]
>>> streamer.throw(CustomException)
WARNING:controlled error CustomException(), continuing
[(0, 'row 0'), (1, 'row 1'), (2, 'row 2'), (3, 'row 3'), (4, 'row 4'), ...]
>>> streamer.throw(RuntimeError)
ERROR:unhandled error RuntimeError(), stopping
INFO:closing connection to database 'testdb'
Traceback (most recent call last):
...
StopIteration
```

When our exception from the domain was received, the generator continued. However, when it received another exception that was not expected, the default block caught where we closed the connection to the database and finished the iteration, which resulted in the generator being stopped. As we can see from the `StopIteration` that was raised, this generator can't be iterated further.

3.1.3. send(value)

In the previous example, we created a simple generator that reads rows from a database, and when we wished to finish its iteration, this generator released the resources linked to the database. This is a good example of using one of the methods that generators provide (`close`), but there is more we can do.

An obvious of such a generator is that it was reading a fixed number of rows from the database.

We would like to parametrize that number so that we can change it throughout different calls. Unfortunately, the `next()` function does not provide us with options for that. But luckily, we have `send()`:

```
def stream_db_records(db_handler):
    retrieved_data = None
    previous_page_size = 10

    try:
        while True:
            page_size = yield retrieved_data
            if page_size is None:
                page_size = previous_page_size

            previous_page_size = page_size
            retrieved_data = db_handler.read_n_records(page_size)

    except GeneratorExit:
        db_handler.close()
```

The idea is that we have now made the coroutine able to receive values from the caller by means of the `send()` method. This method is the one that actually distinguishes a generator from a coroutine because when it's used, it means that the `yield` keyword will appear on the right-hand side of the statement, and its return value will be assigned to something else.

In coroutines, we generally find the `yield` keyword to be used in the following form: `receive = yield produced`

The `yield`, in this case, will do two things. It will send `produced` back to the caller, which will pick it up on the next round of iteration (after calling `next()`, for example), and it will suspend there. At a later point, the caller will want to send a value back to the coroutine by using the `send()` method. This value will become the result of the `yield` statement, assigned in this case to the variable named `receive`.

Sending values to the coroutine only works when this one is suspended at a `yield` statement, waiting for something to produce. For this to happen, the coroutine will have to be advanced to that status. The only way to do this is by calling `next()` on it. This means that before sending anything to the coroutine, this has to be advanced at least once via the `next()` method. Failure to do so will result in an exception:

```
>>> c = coro()
>>> c.send(1)
Traceback (most recent call last):
...
TypeError: can't send non-None value to a just-started generator
```

Important: Always remember to advance a coroutine by calling `next()` before sending any values to it.

Back to our example. We are changing the way elements are produced or streamed to make it able to receive the length of the records it expects to read from the database.

The first time we call `next()`, the generator will advance up to the line containing `yield`; it will provide a value to the caller (`None`, as set in the variable), and it will suspend there).

From here, we have two options. If we choose to advance the generator by calling `next()`, the default value of 10 will be used, and it will go on with this as usual. This is because `next()` is technically the same as `send(None)`, but this is covered in the if statement that will handle the value that we previously set.

If, on the other hand, we decide to provide an explicit value via `send(<value>)`, this one will become the result of the `yield` statement, which will be assigned to the variable containing the length of the page to use, which, in turn, will be used to read from the database.

Successive calls will have this logic, but the important point is that now we can dynamically change the length of the data to read in the middle of the iteration, at any point.

Now that we understand how the previous code works, most Pythonistas would expect a simplified version of it (after all, Python is also about brevity and clean and compact code):

```
def stream_db_records(db_handler):
    retrieved_data = None
    page_size = 10
    try:
        while True:
            page_size = (yield retrieved_data) or page_size
            retrieved_data = db_handler.read_n_records(page_size)
    except GeneratorExit:
        db_handler.close()
```

This version is not only more compact, but it also illustrates the idea better. The parenthesis around the `yield` makes it clearer that it's a statement (think of it as if it were a function call), and that we are using the result of it to compare it against the previous value.

This works as we expect it does, but we always have to remember to advance the coroutine before sending any data to it. If we forget to call the first `next()`, we'll get a `TypeError`. This call could be ignored for our purposes because it doesn't return anything we'll use.

It would be good if we could use the coroutine directly, right after it is created without having to remember to call `next()` the first time, every time we are going to use it. Some authors devised an interesting decorator to achieve this. The idea of this decorator is to advance the coroutine, so the following definition works automatically:

```
@prepare_coroutine
def stream_db_records(db_handler):
    retrieved_data = None
    page_size = 10
    try:
        while True:
            page_size = (yield retrieved_data) or page_size
            retrieved_data = db_handler.read_n_records(page_size)
    except GeneratorExit:
    db_handler.close()

>>> streamer = stream_db_records(DBHandler("testdb"))
>>> len(streamer.send(5))
5
```

7.3.2 3.2. More advanced coroutines

So far, we have a better understanding of coroutines, and we are able to create simple ones to handle small tasks. We can say that these coroutines are, in fact, just more advanced generators (and that would be right, coroutines are just fancy generators), but, if we actually want to start supporting more complex scenarios, we usually have to go for a design that handles many coroutines concurrently, and that requires more features.

When handling many coroutines, we find new problems. As the control flow of our application becomes more complex, we want to pass values up and down the stack (as well as exceptions), be able to capture values from sub-coroutines we might call at any level, and finally schedule multiple coroutines to run toward a common goal.

To make things simpler, generators had to be extended once again. This is addressed by changing the semantic of generators so that they are able to return values, and introducing the new yield from construction.

3.2.1. Returning values in coroutines

As introduced at the beginning, the iteration is a mechanism that calls `next()` on an iterable object many times until a `StopIteration` exception is raised.

So far, we have been exploring the iterative nature of generators: we produce values one at a time, and, in general, we only care about each value as it's being produced at every step of the `for` loop. This is a very logical way of thinking about generators, but coroutines have a different idea; even though they are technically generators, they weren't conceived with the idea of iteration in mind, but with the goal of suspending the execution of a code until it's resumed later on.

This is an interesting challenge; when we design a coroutine, we usually care more about suspending the state rather than iterating (and iterating a coroutine would be an odd case). The challenge lies in that it is easy to mix them both. This is because of a technical implementation detail; the support for coroutines in Python was built upon generators.

If we want to use coroutines to process some information and suspend its execution, it would make sense to think of them as lightweight threads (or green threads, as they are called in other platforms). In such a case, it would make sense if they could return values, much like calling any other regular function.

But let's remember that generators are not regular functions, so in a generator, the construction `value = generator()` will do nothing other than create a generator object. What would be the semantics for making a generator return a value? It will have to be after the iteration is done.

When a generator returns a value, its iteration is immediately stopped (it can't be iterated any further). To preserve the semantics, the `StopIteration` exception is still raised, and the value to be returned is stored inside the exception object. It's the responsibility of the caller to catch it.

In the following example, we are creating a simple generator that produces two values and then returns a third. Notice how we have to catch the exception in order to get this value , and how it's stored precisely inside the exception under the attribute named `value`:

```
>>> def generator():
...     yield 1
...     yield 2
...     return 3
...
>>> value = generator()
>>> next(value)
1
>>> next(value)
2
>>> try:
...     next(value)
... except StopIteration as e:
...     print(f">>>>>> returned value {e.value}")
...
>>>>>> returned value 3
```

3.2.2. Delegating into smaller coroutines: the yield from syntax

The previous feature is interesting in the sense that it opens up a lot of new possibilities with coroutines (generators), now that they can return values. But this feature, by itself, would not be so useful without proper syntax support, because catching the returned value this way is a bit cumbersome.

This is one of the main features of the yield from syntax. Among other things (that we'll review in detail), it can collect the value returned by a sub-generator. Remember that we said that returning data in a generator was nice, but that, unfortunately, writing statements as `value = generator()` wouldn't work. Well, writing it as `value = yield from generator()` would.

3.2.2.1. The simplest use of yield from

In its most basic form, the new `yield from` syntax can be used to chain generators from nested for loops into a single one, which will end up with a single string of all the values in a continuous stream.

The canonical example is about creating a function similar to `itertools.chain()` from the standard library. This is a very nice function because it allows you to pass any number of iterables and will return them all together in one stream.

The naive implementation might look like this:

```python
def chain(*iterables):
    for it in iterables:
        for value in it:
            yield value
```

It receives a variable number of iterables , traverses through all of them, and since each value is iterable, it supports a `for... in..` construction, so we have another for loop to get every value inside each particular iterable, which is produced by the caller function. This might be helpful in multiple cases, such as chaining generators together or trying to iterate things that it wouldn't normally be possible to compare in one go (such as lists with tuples, and so on).

However, the `yield from` syntax allows us to go further and avoid the nested loop because it's able to produce the values from a sub-generator directly. In this case, we could simplify the code like this:

```python
def chain(*iterables):
    for it in iterables:
        yield from it
```

Notice that for both implementations, the behavior of the generator is exactly the same:

```
>>> list(chain("hello", ["world"], ("tuple", " of ", "values.")))
['h', 'e', 'l', 'l', 'o', 'world', 'tuple', ' of ', 'values.']
```

This means that we can use `yield from` over any other iterable, and it will work as if the top-level generator (the one the `yield from` is using) were generating those values itself.

This works with any iterable, and even generator expressions aren't the exception. Now that we're familiar with its syntax, let's see how we could write a simple generator function that will produce all the powers of a number (for instance, if provided with `all_powers(2, 3)`, it will have to produce 2^0, 2^1,... 2^3):

```python
def all_powers(n, pow):
    yield from (n ** i for i in range(pow + 1))
```

While this simplifies the syntax a bit, saving one line of a for statement isn't a big advantage, and it wouldn't justify adding such a change to the language.

Indeed, this is actually just a side effect and the real raison d'être of the `yield from` construction is what we are going to explore in the following two sections.

3.2.2.2. Capturing the value returned by a sub-generator

In the following example, we have a generator that calls another two nested generators, producing values in a sequence. Each one of these nested generators returns a value, and we will see how the top-level generator is able to effectively capture the return value since it's calling the internal generators through `yield from`:

```python
def sequence(name, start, end):
    logger.info(f"{name} started at {start}")
    yield from range(start, end)
    logger.info(f"{name} finished at {end}")
    return end
```

(continues on next page)

(continued from previous page)

```
def main():
    step1 = yield from sequence("first", 0, 5)
    step2 = yield from sequence("second", step1, 10)
    return step1 + step2
```

This is a possible execution of the code in main while it's being iterated:

```
>>> g = main()
>>> next(g)
INFO:generators_yieldfrom_2:first started at 0
0
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)
4
>>> next(g)
INFO:generators_yieldfrom_2:first finished at 5
INFO:generators_yieldfrom_2:second started at 5
5
>>> next(g)
6
>>> next(g)
7
>>> next(g)
8
>>> next(g)
9
>>> next(g)
INFO:generators_yieldfrom_2:second finished at 10
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration: 15
```

The first line of main delegates into the internal generator, and produces the values, extracting them directly from it. This is nothing new, as we have already seen. Notice, though, how the `sequence()` generator function returns the end value, which is assigned in the first line to the variable named `step1`, and how this value is correctly used at the start of the following instance of that generator.

In the end, this other generator also returns the second `end` value, and the main generator, in turn, returns the sum of them, which is the value we see once the iteration has stopped.

Tip: We can use `yield from` to capture the last value of a coroutine after it has finished its processing.

3.2.2.3. Sending and receiving data to and from a sub-generator

Now, we will see the other nice feature of the `yield from` syntax, which is probably what gives it its full power. As we have already introduced when we explored generators acting as coroutines, we know that we can send values and throw exceptions at them, and, in such cases, the coroutine will either receive the value for its internal processing, or it will have to handle the exception accordingly.

If we now have a coroutine that delegates into other ones (such as in the previous example), we would also like to preserve this logic. Having to do so manually would be quite complex if we didn't have this handled by `yield from` automatically.

In order to illustrate this, let's keep the same top-level generator (main) unmodified with respect to the previous example (calling other internal generators), but let's modify the internal generators to make them able to receive values and handle exceptions. The code is probably not idiomatic, only for the purposes of showing how this mechanism works:

```
def sequence(name, start, end):
    value = start
    logger.info("%s started at %i", name, value)

    while value < end:
        try:
            received = yield value
            logger.info("%s received %r", name, received)
            value += 1

        except CustomException as e:
            logger.info("%s is handling %s", name, e)
            received = yield "OK"

    return end
```

Now, we will call the main coroutine, not only by iterating it, but also by passing values and throwing exceptions at it to see how they are handled inside sequence :

```
>>> g = main()
>>> next(g)
INFO: first started at 0
0
>>> next(g)
INFO: first received None
1
>>> g.send("value for 1")
INFO: first received 'value for 1'
2
>>> g.throw(CustomException("controlled error"))
INFO: first is handling controlled error
'OK'
... # advance more times
INFO:second started at 5
5
>>> g.throw(CustomException("exception at second generator"))
INFO: second is handling exception at second generator
'OK'
```

This example is showing us a lot of different things. Notice how we never send values to `sequence`, but only to main, and even so, the code that is receiving those values is the nested generators. Even though we never explicitly send anything to `sequence`, it's receiving the data as it's being passed along by yield from .

The main coroutine calls two other coroutines internally, producing their values, and it will be suspended at a particular point in time in any of those. When it's stopped at the first one, we can see the logs telling us that it is that instance of the coroutine that received the value we sent. The same happens when we throw an exception to it. When the first coroutine finishes, it returns the value that was assigned in the variable named `step1`, and passed as input for the second coroutine, which will do the same (it will handle the `send()` and `throw()` calls, accordingly).

The same happens for the values that each coroutine produces. When we are at any given step, the return from calling `send()` corresponds to the value that the subcoroutine (the one that main is currently suspended at) has produced. When we throw an exception that is being handled, the sequence coroutine produces the value OK , which is propagated to the called (`main`), and which in turn will end up at main's caller.

7.4 4. Asynchronous programming

With the constructions we have seen so far, we are able to create asynchronous programs in Python. This means that we can create programs that have many coroutines, schedule them to work in a particular order, and switch between them when they're suspended after a `yield from` has been called on each of them.

The main advantage that we can take out of this is the possibility of parallelizing I/O operations in a non-blocking way. What we would need is a low-level generator (usually implemented by a third-party library) that knows how to handle the actual I/O while the coroutine is suspended. The idea is for the coroutine to effect suspension so that our program can handle another task in the meantime. The way the application would retrieve the control back is by means of the `yield from` statement, which will suspend and produce a value to the caller (as in the examples we saw previously when we used this syntax to alter the control flow of the program).

This is roughly the way asynchronous programming had been working in Python for quite a few years, until it was decided that better syntactic support was needed.

The fact that coroutines and generators are technically the same causes some confusion. Syntactically (and technically), they are the same, but semantically, they are different. We create generators when we want to achieve efficient iteration. We typically create coroutines with the goal of running non-blocking I/O operations.

While this difference is clear, the dynamic nature of Python would still allow developers to mix these different type of objects, ending up with a runtime error at a very late stage of the program. Remember that in the simplest and most basic form of the `yield from` syntax, we used this construction over iterables (we created a sort of chain function applied over strings, lists, and so on). None of these objects were coroutines, and it still worked. Then, we saw that we can have multiple coroutines, use `yield from` to send the value (or exceptions), and get some results back. These are clearly two very different use cases, however, if we write something along the lines of the following statement: `result = yield from iterable_or_awaitable()`

It's not clear what `iterable_or_awaitable returns`. It can be a simple iterable such as a string, and it might still be syntactically correct. Or, it might be an actual coroutine. The cost of this mistake will be paid much later.

For this reason, the typing system in Python had to be extended. Before Python 3.5, coroutines were just generators with a @coroutine decorator applied, and they were to be called with the yield from syntax. Now, there is a specific type of object, that is, a coroutine.

This change heralded, syntax changes as well. The `await` and `async def` syntax were introduced. The former is intended to be used instead of `yield from`, and it only works with awaitable objects (which coroutines conveniently happen to be). Trying to call `await` with something that doesn't respect the interface of an awaitable will raise an exception. The `async def` is the new way of defining coroutines, replacing the aforementioned decorator, and this actually creates an object that, when called, will return an instance of a coroutine.

Without going into all the details and possibilities of asynchronous programming in Python, we can say that despite the new syntax and the new types, this is not doing anything fundamentally different from concepts we have covered.

The idea of programming asynchronously in Python is that there is an event loop (typically `asyncio` because it's the one that is included in the standard library, but there are many others that will work just the same) that manages a series of coroutines. These coroutines belong to the event loop, which is going to call them according to its scheduling mechanism. When each one of these runs, it will call our code (according to the logic we have defined inside the coroutine we programmed), and when we want to get control back to the event loop, we call `await <coroutine>` , which will process a task asynchronously. The event loop will resume and another coroutine will take place while that operation is left running.

In practice, there are more particularities and edge cases that are beyond the scope. It is, however, worth mentioning that these concepts are related to the ideas introduced in this chapter and that this arena is another place where generators demonstrate being a core concept of the language, as there are many things constructed on top of them.

CHAPTER

EIGHT

UNIT TESTING AND REFACTORING

8.1 1. Design principles and unit testing

8.2 2. Frameworks and tools for testing

8.3 3. Refactoring

8.4 4. More about unit testing

8.5 5. A brief introduction to test-driven development

CHAPTER

NINE

COMMON DESIGN PATTERNS

9.1 1. Considerations for design patterns in Python

9.2 2. Design patterns in action

9.3 3. The null object pattern

9.4 4. Final thoughts about design patterns

CHAPTER

TEN

CLEAN ARCHITECTURE

10.1 1. From clean code to clean architecture

10.2 2. Software components

10.3 3. Use case