Clean Code in Python

Sergio Bugallo

Jan 07, 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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
class DataTransport:
    """An example of an object handling exceptions of different levels."""
    retry_threshold: int = 5
    retry_n_times: int = 3

    def __init__(self, connector):
        self._connector = connector
        self.connection = None

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

    def connect(self):
```

(continues on next page)

(continued from previous page)

```
        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):
```

(continues on next page)

(continued from previous page)

```
        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:

```
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)
```

(continues on next page)

(continued from previous page)

```
    # 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:

```
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):
```

(continues on next page)

(continued from previous page)

```
        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."""
```

(continues on next page)

(continued from previous page)

```
    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:

```
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.

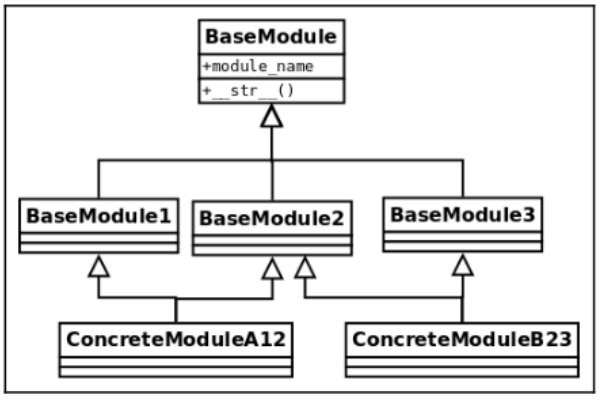
Multiple inheritance is a perfectly valid solution when used correctly, and this opens up new patterns (such as the adapter pattern) and mixins.

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:

```
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")
```

(continues on next page)

(continued from previous page)

```
>>> 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)
...
```

(continues on next page)

(continued from previous page)

```
>>> 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
>>> 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
```

(continues on next page)

(continued from previous page)

```
    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)
```

(continues on next page)

(continued from previous page)

```
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.

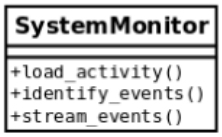
The SRP is closely related to the idea of cohesion in software design, which we already explored, when we discussed separation of concerns in software. What we strive to achieve here is that classes are designed in such a way that most of their properties and their attributes are used by its methods, 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:

```
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.

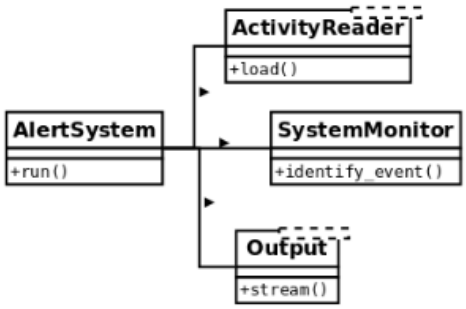
Consider the loader method, which retrieves the information from a particular source. Regardless of how this is done (we can abstract the implementation details here), it is clear that it will have its own sequence of steps, for instance connecting to the data source, loading the data, parsing it into 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:

```
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):
```

(continues on next page)

(continued from previous page)

```
    """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:

```
>>> 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!).

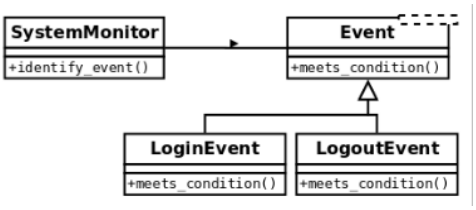
We want to be able to add new types of event without having to change this method (closed for modification). We also want to be able to support new types of event (open for extension) so that when a new event is added, we only have to add code, not change the code that already exists.

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:
    """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.

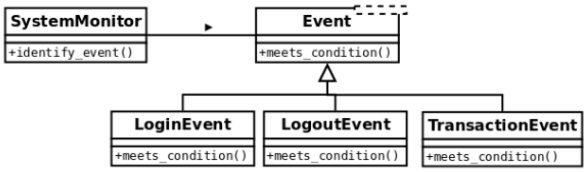
Notice how events are discovered through the `__subclasses__()` method. Supporting new types of event

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)

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:
```

(continues on next page)

(continued from previous page)

```
                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.

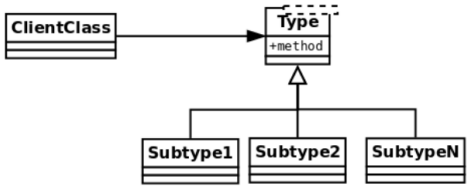
More formally, this is the original definition (LISKOV 01) of Liskov's substitution principle:

```
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"
```

(continues on next page)

(continued from previous page)

```
        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()`.

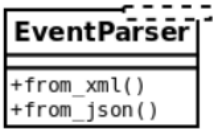
This module also contains a way of registering some types as part of a hierarchy, in what is called a **virtual subclass**. The idea is that this extends the concept of duck typing a little bit further by adding a new criterion—walks like a duck, quacks like a duck, or... it says it is a duck.

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

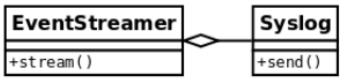
have to invert the dependency: make B have to adapt to A. This is done by presenting an interface and forcing our code not to depend on the concrete implementation of B, but rather on the interface we have defined. It is then B's responsibility to comply with that interface.

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

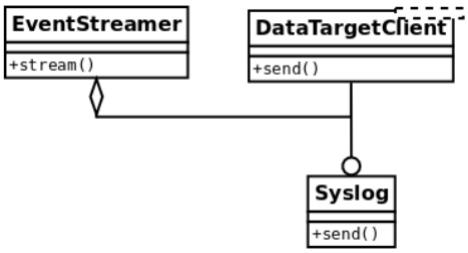
The last part of our event's monitoring system is to deliver the identified events to a data collector to be further analyzed. A naive implementation of such an idea would consist of having an event streamer class that interacts with a data destination, for example, `Syslog`:



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

CHAPTER

SIX

GETTING MORE OUT OF OUR OBJECTS WITH DESCRIPTORS

CHAPTER

SEVEN

USING GENERATORS

CHAPTER

EIGHT

UNIT TESTING AND REFACTORING

CHAPTER

NINE

COMMON DESIGN PATTERNS

CHAPTER

TEN

CLEAN ARCHITECTURE