Mastering Python

Sergio Bugallo

Mar 17, 2020

CONTENTS

Important: Web version availabe at: https://sbugallo.github.io/mastering_python

PDF version available at: https://github.com/sbugallo/mastering_python/raw/master/python.pdf

Errata reports, mistakes or contributions: https://github.com/sbugallo/mastering_python

Part I

Craftmanship

CHAPTER

ONE

STRINGS AND BYTES

The topic of strings may provide some confusion for programmers that used to program only in Python 2. In Python 3, there is only one datatype capable of storing textual information. It is `str`, or simply string. It is an immutable sequence that stores Unicode code points. This is the major difference from Python 2, where `str` represented byte strings: something that is now handled by the `bytes` objects (but not exactly in the same way).

Strings in Python are sequences. This single fact should be enough to include them in a section covering other container types. But they differ from other container types in one important detail. Strings have very specific limitations on what type of data they can store, and that is Unicode text.

`bytes`, and its mutable alternative, `bytearray`, differs from `str` by allowing only bytes as a sequence value, and bytes in Python are integers in the `0 <= x < 256` range. This may be a bit confusing at the beginning, because, when printed, they may look very similar to strings:

```
>>> print(bytes([102, 111, 111]))
b'foo'
```

The `bytes` and `bytearray` types allow you to work with raw binary data that may not always have to be textual (for example, audio/video files, images, and network packets). The true nature of these types is revealed when they are converted into other sequence types, such as list or tuple:

```
>>> list(b'foo bar')
[102, 111, 111, 32, 98, 97, 114]
>>> tuple(b'foo bar')
(102, 111, 111, 32, 98, 97, 114)
```

A lot of Python 3 controversy was about breaking the backwards compatibility for string literals and how Python deals with Unicode. Starting from Python 3.0, every string literal without any prefix is Unicode. So, literals enclosed by single quotes (`'`), double quotes (`"`), or groups of three quotes (single or double) without any prefix represent the `str` data type:

```
>>> type("some string")
<class 'str'>
```

In Python 2, the Unicode literals required a `u` prefix (like `u"some string"`). This prefix is still allowed for backwards compatibility (starting from Python 3.3), but does not hold any syntactic meaning in Python 3.

Byte literals were already presented in some of the previous examples, but let's explicitly present their syntax for the sake of consistency. Bytes literals are enclosed by single quotes, double quotes, or triple quotes, but must be preceded with a `b` or `B` prefix:

```
>>> type(b"some bytes")
<class 'bytes'>
```

Note that Python does not provide a syntax for `bytearray` literals. If you want to create a `bytearray` value, you need to use a `byte``s literal and a ``bytearray()` type constructor:

```
>>> bytearray(b'some bytes')
bytearray(b'some bytes')
```

It is important to remember that Unicode strings contain abstract text that is independent from the byte representation. This makes them unable to be saved on the disk or sent over the network without encoding them to binary data. There are two ways to encode string objects into byte sequences:

- Using the `str.encode(encoding, errors)` method, which encodes the string using a registered codec for encoding. Codec is specified using the encoding argument, and, by default, it is 'utf-8'. The second argument, errors, specifies the error handling scheme. It can be 'strict' (default), 'ignore' , 'replace' , 'xmlcharrefreplace', or any other registered handler (refer to the built-in codecs module documentation).
- Using the `bytes(source, encoding, errors)` constructor, which creates a new bytes sequence. When the source is of the `str` type, then the encoding argument is obligatory and it does not have a default value. The usage of the encoding and errors arguments is the same as for the `str.encode()` method.

Binary data represented by `bytes` can be converted into a string in an analogous way:

- Using the `bytes.decode(encoding, errors)` method, which decodes the bytes using the codec registered for encoding. The arguments of this method have the same meaning and defaults as the arguments of `str.encode()`.
- Using the `str(source, encoding, error)` constructor, which creates a new string instance. Similar to the `bytes()` constructor, the encoding argument in the `str()` call has no default value and must be provided if the bytes sequence is used as a source.

Tip: Due to changes made in Python 3, some people tend to refer to the `bytes` instances as byte strings. This is mostly due to historic reasons: `bytes` in Python 3 is the sequence type that is the closest one to the `str` type from Python 2 (but not the same). Still, the `bytes` instance is a sequence of bytes and also does not need to represent textual data. So, in order to avoid any confusion, it is advised to always refer to them as either bytes or byte sequence, despite their similarities to strings. The concept of strings is reserved for textual data in Python 3, and this is now always `str`.

1.1 1. Implementation details

Python strings are immutable. This is also true for byte sequences. This is an important fact, because it has both advantages and disadvantages. It also affects the way strings should be handled in Python efficiently. Thanks to immutability, strings can be used as dictionary keys or set collection elements because, once initialized, they will never change their value. On the other hand, whenever a modified string is required (even with only tiny modification), a completely new instance needs to be created. Fortunately, `bytearray`, as a mutable version of `bytes`, does not have such an issue. Byte arrays can be modified inplace (without creating new objects) through item assignments and can be dynamically resized, exactly like lists: using appends, pops, inserts, and so on.

1.2 2. String concatenation

The fact that Python strings are immutable imposes some problems when multiple string instances need to be joined together. As we stated previously, concatenating immutable sequences results in the creation of a new sequence object. Consider that a new string is built by repeated concatenation of multiple strings, as follows:

```
substrings = ["These ", "are ", "strings ", "to ", "concatenate."]
s = ""

for substring in substrings:
    s += substring
```

This will result in quadratic runtime costs in the total string length. In other words, it is highly inefficient. For handling such situations, the `str.join()` method is available. It accepts iterables of strings as the argument and returns joined strings. The call to `join()` of the str type can be done in two forms:

```
# using empty literal
s = "".join(substrings)

# using "unbound" method call
str.join("", substrings)
```

The first form of the `join()` call is the most common idiom. The string that provides this method will be used as a separator between concatenated substrings. Consider the following example:

```
>>> ','.join(['some', 'comma', 'separated', 'values'])
'some,comma,separated,values'
```

It is worth remembering that just because it is faster (especially for large lists), it does not mean that the `join()` method should be used in every situation where two strings need to be concatenated. Despite being a widely recognized idiom, it does not improve code readability. And readability counts! There are also some situations where `join()` may not perform as well as ordinary concatenation with a + operator. Here are some examples:

- If the number of substrings is very small and they are not contained already by some iterable variable (existing list or tuple of strings): in some cases the overhead of creating a new sequence just to perform concatenation can overshadow the gain of using `join()`.
- When concatenating short literals: thanks to some interpreter-level optimizations, such as constant folding in CPython, some complex literals (not only strings), such as `'a' + 'b' +'c'`, can be translated into a shorter form at compile time (here `'abc'`). Of course, this is enabled only for constants (literals) that are relatively short.

Ultimately, if the number of strings to concatenate is known beforehand, the best readability is ensured by proper string formatting either using the `str.format()` method, the `%` operator, or f-string formatting. In code sections where the performance is not critical or the gain from optimizing string concatenation is very little, string formatting is recommended as the best alternative to concatenation.

1.2.1 2.1. Constant folding, the peephole optimizer, and the AST optimizer

CPython uses various techniques to optimize your code. The first optimization takes place as soon as source code is transformed into the form of the abstract syntax tree, just before it is compiled into byte code. CPython can recognize specific patterns in the abstract syntax tree and make direct modifications to it. The other kind of optimizations are handled by the peephole optimizer. It implements a number of common optimizations directly on Python's byte code. As we mentioned earlier, constant folding is one such feature. It allows the interpreter to convert complex literal expressions (such as `"one" + " " + "thing"`, `" " * 79`, or `60 * 1000`) into a single literal that does not require any additional operations (concatenation or multiplication) at runtime.

Until Python 3.5, all constant folding was done in CPython only by the peephole optimizer. For strings, the resulting constants were limited in length by a hardcoded value. In Python 3.5, this value was equal to 20. In Python 3.7, most of the constant folding optimizations are handled earlier on the abstract syntax tree level. These particular details are a curiosity rather than a thing that can be relied on in your day-to-day programming. Information about other interesting optimizations performed by AST and peephole optimizers can be found in the *Python/ast_opt.c* and *Python/peephole.c* files of Python's source code.

1.3 3. String formatting with f-strings

F-strings are one of the most beloved new Python features that came with Python 3.6. It's also one of the most controversial features of that release. The f-strings or formatted string literals that were introduced by the PEP 498 document add a new way to format strings in Python. Before Python 3.6, there were two basic ways to format strings:

- Using `%` formatting for example `"Some string with included % value" % "other"`
- Using the `str.format()` method for example `"Some string with included {other} value".format(other="other")`

Formatted string literals are denoted with the `f` prefix, and their syntax is closest to the `str.format()` method, as they use a similar markup for denoting replacement fields in the text that has to be formatted. In the `str.format()` method, the text substitutions refer to arguments and keyword arguments that are passed to the formatting method. You can use either anonymous substitutions that will translate to consecutive argument indexes, explicit argument indexes, or keyword names.

This means that the same string can be formatted in different ways:

```
>>> from sys import version_info
>>> "This is Python {}.{}".format(*version_info)
'This is Python 3.7'
>>> "This is Python {0}.{1}".format(*version_info)
'This is Python 3.7'
>>> "This is Python {major}.{minor}".format(major=version_info.major,␣
↪minor=version_info.minor)
'This is Python 3.7'
```

What makes f-strings special is that replacement fields can be any Python expression, and it will be evaluated at runtime. Inside of strings, you have access to any variable that is available in the same namespace as the formatted literal. With f-strings, the preceding examples could be written in the following way:

```
>>> from sys import version_info
>>> f"This is Python {version_info.major}.{version_info.minor}"
'This is Python 3.7'
```

The ability to use expressions as replacement fields make formatting code simpler and shorter. You can also use the same formatting specifiers of replacement fields (for padding, aligning, signs, and so on) as the `str.format()` method, and the syntax is as follows:

```
f"{replacement_field_expression:format_specifier}"
```

The following is a simple example of code that prints the first ten powers of the number 10 using f-strings and aligns results using string formatting with padding:

```
>>> for x in range(10):
...     print(f"10^{x} == {10**x:10d}")
...
10^0 ==          1
10^1 ==         10
10^2 ==        100
10^3 ==       1000
10^4 ==      10000
10^5 ==     100000
10^6 ==    1000000
10^7 ==   10000000
10^8 ==  100000000
10^9 == 1000000000
```

The full formatting specification of the Python string is almost like a separate minilanguage inside Python. The best reference for it is the official documentation which you can find under https://docs.python.org/3/library/string.

html. Another useful internet resource for that topic is https://pyformat.info/, which presents the most important elements of this specification using practical examples.

CHAPTER

TWO

PYTHON'S PROTOCOLS: DUNDER METHODS AND ATTRIBUTES

The Python data model specifies a lot of specially named methods that can be overridden in your custom classes to provide them with additional syntax capabilities. You can recognize these methods by their specific naming conventions that wrap the method name with **double underscores**. Because of this, they are sometimes referred to as **dunder**. It is simply a speech shorthand for double underscores.

The most common and obvious example of such dunder methods is `__init__()`, which is used for class instance initialization:

```
class CustomUserClass:
    def __init__(self, initiatization_argument):
        ...
```

These methods, either alone or when defined in specific combination, constitute the so-called language protocols. If an object implements specific language protocols, it becomes compatible with specific parts of the Python language syntax. The following is the table of the most important protocols within the Python language:

Protocol name	Methods	Description
Callable protocol	`__call__()`	Allows objects to be called with the parentheses syntax: `instance()`
Descriptor protocols	`__set__()`, `__get__()` and `__del__()`	Allows us to manipulate the attribute access pattern of classes
Container protocol	`__contains__()`	Allows us to test whether or not an object contains some value using the `in` keyword: `value in instance`
Iterable protocol	`__iter__()`	Allows objects to be iterated over using the `for` keyword: `for value in instance`
Sequence protocols	`__len__()` and `__getitem__()`	Allows objects to be indexed with square bracket syntax and queried for length using a built-in function: `item = instance[index]`, `length = len(instance)`

These are the most important language protocols from the perspective of this chapter. The full list is, of course, a lot longer. For instance, Python provides over 50 dunder methods that allow us to emulate numeric values. Each of these methods is correlated to some specific mathematical operator, and so could be considered a separate language protocol. The full list of all the dunder methods can be found in the official documentation of the Python data model (see https://docs.python.org/3/reference/datamodel.html).

Language protocols are the foundation of the concept of interfaces in Python. One implementation of Python interfaces is in abstract base classes that allow us to define an arbitrary set of attributes and methods as an interface definition. These definitions of interfaces in the form of abstract classes can be later used to test whether or not the given object is compatible with a specific interface. The `collections.abc` module from the Python standard library provides a collection of abstract base classes that refer to the most common Python language protocol.

The same dunder convention is also used for specific attributes of custom user functions and is used to store various metadata about Python objects. These attributes are as follows:

- `__doc__`: A writable attribute that holds the function's documentation. It is, by default, populated by the `docstring` function.
- `__name__`: A writable attribute that holds the function's name.
- `__qualname__`: A writable attribute that holds the function's **qualified name**. The qualified name is a full dotted path to the object (with class names) in the global scope of the module where the object is defined.
- `__module__`: A writable attribute that holds the name of the module that function belongs to.
- `__defaults__`: A writable attribute that holds the default argument values if the function has any.
- `__code__`: A writable attribute that holds the function's compile code object.
- `__globals__`: A read-only attribute that holds the reference to the dictionary of global variables for that function's scope. The global scope for a function is the namespace of the module where this function is defined.
- `__dict__`: A writable attribute that holds a dictionary of function attributes. Functions in Python are first-class objects, so they can have any arbitrary arguments defined, just like any other object.
- `__closure__`: A read-only attribute that holds a tuple of cells with the function's free variables. Closure cells allow you to create parametrized function decorators.
- `__annotations__`: A writable attribute that holds the function's argument and return annotations.
- `__kwdefaults__`: A writable attribute that holds the default argument values for keyword-only arguments if the function has any.

CHAPTER

THREE

SEQUENCES

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
```

3.1 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.

CHAPTER

FOUR

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.

4.1 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.

CHAPTER

FIVE

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.

5.1 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. `__iter__` should return the iterator itself.

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)

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.

5.2 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.

CHAPTER

SIX

CONTAINER OBJECTS

Python provides a good selection of built-in data containers that allow you to efficiently solve many problems if you choose them wisely. Types that you should already know of are those that have dedicated literals:

- Lists
- Tuples
- Dictionaries
- Sets

Python is, of course, not limited to these four containers, and it extends the list of possible choices through its standard library. In many cases, solutions to some problems may be as simple as making a good choice for the data structure to hold your data.

6.1 1. Lists and tuples

Two of the most basic collection types in Python are lists and tuples, and they both represent sequences of objects. The basic difference between them should be obvious for anyone who has spent more than a few hours with Python; lists are dynamic, so they can change their size, while tuples are immutable (cannot be modified after they are created).

Lists and tuples in Python have various optimizations that make allocations/deallocations of small objects fast. They are also the recommended datatypes for structures where the position of the element is information by itself. For example, a tuple may be a good choice for storing a pair of (x, y) coordinates. Implementation details regarding tuples are not interesting. The only important thing about them in the scope of this chapter is that tuple is immutable and thus hashable. A detailed explanation of this section will be covered later in the Dictionaries section. More interesting than tuples is its dynamic counterpart: lists. In the next section, we will discuss how it really works, and how to deal with it efficiently.

6.1.1 1.1. Implementation details

Many programmers easily confuse Python's `list` type with the concept of linked lists which are found often in standard libraries of other languages, such as C, C++, or Java. In fact, CPython lists are not lists at all. In CPython, lists are implemented as variable length arrays. This should be also true for other implementations, such as Jython and IronPython, although such implementation details are often not documented in these projects. The reasons for such confusion is clear. This datatype is named `list` and also has an interface that could be expected from any linked list implementation.

Why it is important and what does it mean? Lists are one of the most popular data structures, and the way in which they are used greatly affects every application's performance. CPython is the most popular and used implementation, so knowing its internal implementation details is crucial.

Lists in Python are contiguous arrays of references to other objects. The pointer to this array and the length is stored in the list's head structure. This means that every time an item is added or removed, the array of references needs to be resized (reallocated). Fortunately, in Python, these arrays are created with exponential over allocation, so not every operation requires an actual resize of the underlying array. This is how the amortized cost of

appending and popping elements can be low in terms of complexity. Unfortunately, some other operations that are considered cheap in ordinary linked lists have relatively high computational complexity in Python:

- Inserting an item at an arbitrary place using the `list.insert` method has complexity O(n).
- Deleting an item using `list.delete` or using the del operator har has complexity O(n)

At least retrieving or setting an element using an index is an operation where cost is independent of the list's size, and the complexity of these operations is always O(1).

Let's define `n` as the length of a list. Here is a full table of average time complexities for most of the list operations: ========== ========== Operation Complexity ========== ========== Copy O(n) Append O(1) Insert O(n) Get item O(1) Set item O(1) Delete item O(n) Iteration O(n) Get slice of length `k` O(k) Del slice O(n) Set slice of length `k` O(k+n) Extend O(k) Multiply by `k` O(nk) Test existence (`element in list`) O(n) `min()`/`max()` O(n) Get length O(1) ========== ==========

For situations where a real linked list or doubly linked list is required, Python provides a `deque` type in the `collections` built-in module. This is a data structure that allows us to append and pop elements at each side with O(1) complexity. This is a generalization of stacks and queues, and should work fine anywhere where a doubly linked list is required.

6.1.2 1.2. List comprehensions

As you probably know, writing a piece of code such as this can be tedious:

```
>>> evens = []
>>> for i in range(10):
...     if i % 2 == 0:
...         evens.append(i)
...
>>> evens
[0, 2, 4, 6, 8]
```

This may work for C, but it actually makes things slower for Python for the following reasons:

- It makes the interpreter work on each loop to determine what part of the sequence has to be changed
- It makes you keep a counter to track what element has to be processed
- It requires additional function lookups to be performed at every iteration because `append()` is a list's method

A list comprehension is a better pattern for these kind of situations. It allows us to define a list by using a single line of code:

```
>>> [i for i in range(10) if i % 2 == 0]
[0, 2, 4, 6, 8]
```

This form of writing is much shorter and involves fewer elements. In a bigger program, this means less bugs and code that is easier to understand. This is the reason why many experienced Python programmers will consider such forms as being more readable.

Tip: There is a myth among some Python programmers that list comprehensions can be a workaround for the fact that the internal array representing the list object must be resized with every few additions. Some say that the array will be allocated once in just the right size. Unfortunately, this isn't true.

The interpreter, during evaluation of the comprehension, can't know how big the resulting container will be, and it can't preallocate the final size of the array for it. Due to this, the internal array is reallocated in the same pattern as it would be in the for loop. Still, in many cases, list creation using comprehensions is both cleaner and faster than using ordinary loops.

6.1.3 1.3. Other idioms

Another typical example of a Python idiom is the use of `enumerate()`. This built-in function provides a convenient way to get an index when a sequence is iterated inside of a loop. Consider the following piece of code as an example of tracking the element index without the `enumerate()` function:

```
>>> i = 0
>>> for element in ['one', 'two', 'three']:
...     print(i, element)
...     i += 1
...
0 one
1 two
2 three
```

This can be replaced with the following code, which is shorter and definitely cleaner:

```
>>> for i, element in enumerate(['one', 'two', 'three']):
...     print(i, element)
...
0 one
1 two
2 three
```

If you need to aggregate elements of multiple lists (or any other iterables) in the one-by-one fashion, you can use the built-in `zip()`. This is a very common pattern for uniform iteration over two same-sized iterables:

```
>>> for items in zip([1, 2, 3], [4, 5, 6]):
...     print(items)
...
(1, 4)
(2, 5)
(3, 6)
```

Note that the results of `zip()` can be reversed by another `zip()` call:

```
>>> for items in zip(*zip([1, 2, 3], [4, 5, 6])):
...     print(items)
...
(1, 2, 3)
(4, 5, 6)
```

One important thing you need to remember about the `zip()` function is that it expects input iterables to be the same size. If you provide arguments of different lengths, then it will trim the output to the shortest argument, as shown in the following example:

```
>>> for items in zip([1, 2, 3, 4], [1, 2]):
...     print(items)
...
(1, 1)
(2, 2)
```

Another popular syntax element is sequence unpacking. It is not limited to lists and tuples, and will work with any sequence type (even strings and byte sequences). It allows us to unpack a sequence of elements into another set of variables as long as there are as many variables on the left-hand side of the assignment operator as the number of elements in the sequence. If you paid attention to the code snippets, then you might have already noticed this idiom when we were discussing the `enumerate()` function.

The following is a dedicated example of that syntax element:

```
>>> first, second, third = "foo", "bar", 100
>>> first
```

(continues on next page)

(continued from previous page)

```
'foo'
>>> second
'bar'
>>> third
100
```

Unpacking also allows us to capture multiple elements in a single variable using starred expressions as long as it can be interpreted unambiguously. Unpacking can also be performed on nested sequences. This can come in handy, especially when iterating on some complex data structures built out of multiple sequences. Here are some examples of more complex sequence unpacking:

```
>>> first, second, *rest = 0, 1, 2, 3
>>> first
0
>>> second
1
>>> rest
[2, 3]

>>> first, *inner, last = 0, 1, 2, 3
>>> first
0
>>> inner
[1, 2]
>>> last
3

>>> (a, b), (c, d) = (1, 2), (3, 4)
>>> a, b, c, d
(1, 2, 3, 4)
```

6.2 2. Dictionaries

Dictionaries are one of most versatile data structures in Python. The `dict` type allows you to map a set of unique keys to values, as follows:

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

Dictionary literals are a very basic thing, and you should already know about them. Python allows programmers to also create a new dictionary using comprehensions, similar to the list comprehensions mentioned earlier. Here is a very simple example that maps numbers in a range from 0 to 99 to their squares:

```
squares = {number: number**2 for number in range(100)}
```

What is important is that the same benefits of using list comprehensions apply to dictionary comprehensions. So, in many cases, they are more efficient, shorter, and cleaner. For more complex code, when many `if` statements or function calls are required to create a dictionary, the simple `for` loop may be a better choice, especially if it improves readability.

For Python programmers new to Python 3, there is one important note about iterating over dictionary elements. The `keys()`, `values()`, and `items()` dictionary methods are no longer return lists. Also, their counterparts, `iterkeys()`, `itervalues()`, and `iteritems()`, which returned iterators instead, are missing in Python 3. Now, the `keys()`, `values()`, and `items()` methods return special view objects:

- `keys()`: This returns the dict_keys object which provides a view on all keys of the dictionary

- `values()`: This returns the dict_values object which provides a view on all values of the dictionary
- `items()`: This returns the dict_items object, providing views on all (key, value) two-tuples of the dictionary

View objects provide a view on the dictionary content in a dynamic way so that every time the dictionary changes, the views will reflect these changes, as shown in this example:

```
>>> person = {'name': 'John', 'last_name': 'Doe'}
>>> items = person.items()
>>> person['age'] = 42
>>> items
dict_items([('name', 'John'), ('last_name', 'Doe'), ('age', 42)])
```

View objects join the behavior of lists returned by the implementation of old methods with iterators that have been returned by their ´´iter´´ counterparts. Views do not need to redundantly store all values in memory (like lists do), but are still allowed to access their length (using the ´´len()´´ function) and testing for membership (using the ´´in´´ keyword). Views are, of course, iterable.

The last important thing about views is that both view objects returned by the `keys()` and `values()` methods ensure the same order of keys and values. In Python 2, you could not modify the dictionary content between these two calls if you wanted to ensure the same order of retrieved keys and values. `dict_keys` and `dict_values` are now dynamic, so even if the content of the dictionary changes between the `keys()` and `values()` calls, the order of iteration is consistent between these two views.

6.2.1 2.1. Implementation details

CPython uses hash tables with pseudo-random probing as an underlying data structure for dictionaries. It seems like a very deep implementation detail, but it is very unlikely to change in the near future, so it is also a very interesting fact for the Python programmer.

Due to this implementation detail, only objects that are hashable can be used as a dictionary key. An object is hashable if it has a hash value that never changes during its lifetime, and can be compared to different objects. Every Python built-in type that is immutable is also hashable. Mutable types, such as list, dictionaries, and sets, are not hashable, and so they cannot be used as dictionary keys. Protocol that defines if a type is hashable consists of two methods:

- `__hash__`: This provides the hash value (as an integer) that is needed by the internal `dict` implementation. For objects that are instances of user-defined classes, it is derived from their `id()`.
- `__eq__`: This compares if two objects have the same value. All objects that are instances of user-defined classes compare as unequal by default, except for themselves.

Two objects that are compared as equal must have the same hash value. The reverse does not need to be true. This means that collisions of hashes are possible: two objects with the same hash may not be equal. It is allowed, and every Python implementation must be able to resolve hash collisions. CPython uses open addressing to resolve them. The probability of collisions greatly affects dictionary performance, and, if it is high, the dictionary will not benefit from its internal optimizations.

While three basic operations, adding, getting, and deleting an item, have an average time complexity equal to O(1), their amortized worst case complexities are a lot higher. It is O(n), where `n` is the current dictionary size. Additionally, if user-defined class objects are used as dictionary keys and they are hashed improperly (with a high risk of collisions), this will have a huge negative impact on the dictionary's performance. The full table of CPython's time complexities for dictionaries is as follows:

Operation	Average complexity	Amortized worst case complexity
Get item	O(1)	O(n)
Set item	O(1)	O(n)
Delete item	O(1)	O(n)
Copy	O(n)	O(n)
Iteration	O(n)	O(n)

It is also important to know that the `n` number in worst case complexities for copying and iterating the dictionary is the maximum size that the dictionary ever achieved, rather than the size at the time of operation. In other words, iterating over the dictionary that once was huge but greatly shrunk in time may take a surprisingly long time. In some cases, it may be better to create a new dictionary object from a dictionary that needs to be shrunk if it has to be iterated often instead of just removing elements from it.

6.2.2 2.2. Weaknesses and alternatives

For a very long time, one of the most common pitfalls regarding dictionaries was expecting that they preserve the order of elements in which new keys were added. The situation has changed a bit in Python 3.6, and the problem was finally solved in Python 3.7 on the level of language specification.

But, before we dig deeper into the situation of Python 3.6 and later releases, we need to make a small detour and examine the problem as if we were still stuck in the past, when the only Python releases available were older than 3.6. In the past, you could have a situation where the consecutive dictionary keys also had hashes that were consecutive values too. And, for a very long time, this was the only situation when you could expect that you would iterate over dictionary elements in the same order as they were added to the dictionary. The easiest way to present this is by using integer numbers, as hashes of integer numbers are the same as their value:

```
>>> {number: None for number in range(5)}.keys()
dict_keys([0, 1, 2, 3, 4])
```

Using other datatypes that hash differently could show that the order is not preserved. Here is an example that was executed in CPython 3.5:

```
>>> {str(number): None for number in range(5)}.keys()
dict_keys(['1', '2', '4', '0', '3'])
>>> {str(number): None for number in reversed(range(5))}.keys()
dict_keys(['2', '3', '1', '4', '0'])
```

As shown in the preceding code, for CPython 3.5 (and also earlier versions), the resulting order is both dependent on the hashing of the object and also on the order in which the elements were added. This is definitely not what can be relied on, because it can vary with different Python implementations.

So, what about Python 3.6 and later releases? Starting from Python 3.6, the CPython interpreter uses a new compact dictionary representation that has a noticeably smaller memory footprint and also preserves order as a side effect of that new implementation. In Python 3.6, the order preserving nature of dictionaries was only an implementation detail, but in Python 3.7, it has been officially declared in the Python language specification. So, starting from Python 3.7, you can finally rely on the item insertion order of dictionaries.

In parallel to the CPython implementation of dictionaries, Python 3.6 introduced another change in the syntax that is related to the order of items in dictionaries. As defined in the PEP 486 "Preserving the order of `**kwargs` in a function" document, the order of keyword arguments collected using the `**kwargs` syntax must be the same as presented in function call. This behavior can be clearly presented with the following example:

```
>>> def fun(**kwargs):
...     print(kwargs)
...
>>> fun(a=1, b=2, c=3)
{'a': 1, 'b': 2, 'c': 3}
>>> fun(c=1, b=2, a=3)
{'c': 1, 'b': 2, 'a': 3}
```

However the preceding changes can be used effectively only in the newest releases of Python. So, what should you do if you have a library that must work on older versions of Python too, and some parts of its code requires order-preserving dictionaries? The best option is to be clear about your expectations regarding dictionary ordering and use a type that explicitly preserves the order of elements.

Fortunately, the Python standard library provides an ordered dictionary type called `OrderedDict` in the `collections` module. The constructor of this type accepts `iterable` as the initialization argument. Each element of that argument should be a pair of a dictionary key and value, as in the following example:

```
>>> from collections import OrderedDict
>>> OrderedDict((str(number), None) for number in range(5)).keys()
odict_keys(['0', '1', '2', '3', '4'])
```

It also has some additional features, such as popping items from both ends using the `popitem()` method, or moving the specified element to one of the ends using the`` move_to_end()`` method. A full reference on that collection is available in the Python documentation (refer to https://docs.python.org/3/library/collections.html). Even if you target only Python in version 3.7 or newer, which guarantees the preservation of the item insertion order, the `OrderedDict type` is still useful. It allows you to make your intention clear. If you define your variable with `OrderedDict` instead of a plain dict, it becomes obvious that, in this particular case, the order of inserted items is important.

The last interesting note is that, in very old code bases, you can find `dict` as a primitive set implementation that ensures uniqueness of elements. While this will give proper results, you should avoid such use of that type unless you target Python versions lower than 2.3. Using dictionaries in this way is wasteful in terms of resources. Python has a builtin `set` type that serves this purpose. In fact, it has very similar internal implementation to dictionaries in CPython, but offers some additional features, as well as specific set-related optimizations.

6.3 3. Sets

Sets are a very robust data structure that are mostly useful in situations where the order of elements is not as important as their uniqueness. They are also useful if you need to efficiently check efficiency if the element is contained in a collection. Sets in Python are generalizations of mathematic sets, and are provided as built-in types in two flavors:

- `set()`: This is a mutable, non-ordered, finite collection of unique, immutable (hashable) objects
- `frozenset()`: This is an immutable, hashable, non-ordered collection of unique, immutable (hashable) objects

The immutability of `frozenset()` objects makes it possible for them to be included as dictionary keys and also other `set()` and `frozenset()` elements. A plain mutable `set()` object cannot be used within another `set()` or `frozenset()`. Attempting to do so will raise a `TypeError` exception, as in the following example:

```
>>> set([set([1,2,3]), set([2,3,4])])
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

On the other hand, the following `set` initializations are completely correct, and do not raise exceptions:

```
>>> set([frozenset([1,2,3]), frozenset([2,3,4])])
{frozenset({1, 2, 3}), frozenset({2, 3, 4})}
>>> frozenset([frozenset([1,2,3]), frozenset([2,3,4])])
frozenset({frozenset({1, 2, 3}), frozenset({2, 3, 4})})
```

Mutable sets can be created in three ways:

- Using a `set()` call that accepts optional iterables as the initialization argument, such as `set([0, 1, 2])`
- Using a set comprehension such as `{element for element in range(3)}`
- Using set literals such as `{1, 2, 3}`

Note that using literals and comprehensions for sets requires extra caution, because they are very similar in form to dictionary literals and comprehensions. Also, there is no literal for empty set objects: empty curly brackets `{}` are reserved for empty dictionary literals.

6.3.1 3.1. Implementation details

Sets in CPython are very similar to dictionaries. As a matter of fact, they are implemented like dictionaries with dummy values, where only keys are actual collection elements. Sets also exploit this lack of values in mapping for additional optimizations.

Thanks to this, sets allow very fast additions, deletions, and checks for element existence with the average time complexity equal to O(1). Still, since the implementation of sets in CPython relies on a similar hash table structure, the worst case complexity for these operations is still O(n), where `n` is the current size of a set.

Other implementation details also apply. The item to be included in a set must be hashable, and, if instances of user-defined classes in the set are hashed poorly, this will have a negative impact on their performance.

Despite their conceptual similarity to dictionaries, sets in Python 3.7 do not preserve the order of elements in specification, or as a detail of CPython implementation. Let's take a look at the supplemental data types and containers.

6.4 4. Supplemental data types and containers

In the previous subsections, we concentrated mostly on those data types that have dedicated literals in the Python syntax. These were also the types that are implemented at the interpreter-level. However, Python's standard library offers a great collection of supplemental data types that can be effectively used in places where the basic built-in types show their shortcomings, or places where the nature of the data requires specialized handling (for example, in the presentation of time and dates).

The most common are data containers that are found in the collections, and we have already briefly mentioned two of them: `deque` and `OrderedDict`. However, the landscape of data structures available for Python programmers is enormous and almost every module of the Python standard library defines some specialized types for handling the data of different problem domains.

6.4.1 4.1. Specialized data containers from the collections module

Every data structure has its shortcomings. There is no single collection that can suit every problem, and four basic types of them (tuple, list, set, and dictionary) is still not a wide range of choices. These are the most basic and important collections that have a dedicated literal syntax. Fortunately, Python provides far more options in its standard library through the `collections` built-in module. Here are the most important universal data containers provided by this module:

- `namedtuple()`: This is a factory function for creating tuple subclasses whose indexes can be accessed as named attributes
- `deque`: This is a double-ended queue, a list-like generalization of stacks and queues with fast appends and pops on both ends
- `ChainMap`: This is a dictionary-like class to create a single view of multiple mappings
- `Counter`: This is a dictionary subclass for counting hashable objects
- `OrderedDict`: This is a dictionary subclass that preserves the order that the entries were added in
- `defaultdict`: This is a dictionary subclass that can supply missing values using a user-defined factory function

6.4.2 4.2. Symbolic enumeration with the enum module

One of the special handy types found in the Python standard is the `Enum` class from the `enum` module. This is a base class that allows you to define symbolic enumerations, similar in concept to the enumerated types found in many other programming languages (C, C++, C#, Java, and many more) that are often denoted with the `enum` keyword.

In order to define your own enumeration in Python, you will need to subclass the `Enum` class and define all enumeration members as class attributes. The following is an example of a simple Python `enum`:

```
from enum import Enum

class Weekday(Enum):
    MONDAY = 0
    TUESDAY = 1
    WEDNESDAY = 2
    THURSDAY = 3
    FRIDAY = 4
    SATURDAY = 5
    SUNDAY = 6
```

The Python documentation defines the following nomenclature for `enum`:

- `enumeration` or `enum`: This is the subclass of `Enum` base class. Here, it would be `Weekday`.
- `member`: This is the attribute you define in the Enum subclass. Here, it would be `Weekday.MONDAY`, `Weekday.TUESDAY`, and so on.
- `name`: This is the name of the `Enum` subclass attribute that defines the member. Here, it would be `MONDAY` for `Weekday.MONDAY`, `TUESDAY` for `Weekday.TUESDAY`, and so on.
- `value`: This is the value assigned to the Enum subclass attribute that defines the `member`. Here, for `Weekday.MONDAY` it would be one, for `Weekday.TUESDAY` it would be two, and so on.

You can use any type as the `enum` member value. If the member value is not important in your code, you can even use the `auto()` type, which will be replaced with automatically generated values. Here is the previous example rewritten with the use of `auto` in it:

```
from enum import Enum, auto


class Weekday(Enum):
     MONDAY = auto()
     TUESDAY = auto()
     WEDNESDAY = auto()
     THURSDAY = auto()
     FRIDAY = auto()
     SATURDAY = auto()
     SUNDAY = auto()
```

Enumerations in Python are really useful in every place where some variable can take a finite number of values/choices. For instance, they can be used to define statues of objects, as shown in the following example:

```
from enum import Enum, auto


class OrderStatus(Enum):
     PENDING = auto()
     PROCESSING = auto()
     PROCESSED = auto()

class Order:
     def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        self.status = OrderStatus.PENDING

    def process(self):
        if self.status == OrderStatus.PROCESSED:
            raise RuntimeError("Can't process order that has been already processed
↪")

        self.status = OrderStatus.PROCESSING
        ...
        self.status = OrderStatus.PROCESSED
```

Another use case for enumerations is storing selections of non-exclusive choices. This is something that is often implemented using bit flags and bit masks in languages where bit manipulation of numbers is very common, like C. In Python, this can be done in a more expressive and convenient way using `FlagEnum`:

```
from enum import Flag, auto


class Side(Flag):
    GUACAMOLE = auto()
    TORTILLA = auto()
    FRIES = auto()
    BEER = auto()
    POTATO_SALAD = auto()
```

You can combine such flags using bitwise operations (the `|` and `&` operators) and test for flag membership with the `in` keyword. Here are some examples for a `Side` enumeration:

```
>>> mexican_sides = Side.GUACAMOLE | Side.BEER | Side.TORTILLA
>>> bavarian_sides = Side.BEER | Side.POTATO_SALAD
>>> common_sides = mexican_sides & bavarian_sides
>>> Side.GUACAMOLE in mexican_sides
True
>>> Side.TORTILLA in bavarian_sides
False
>>> common_sides
<Side.BEER: 8>
```

Symbolic enumerations share some similarity with dictionaries and named tuples because they all map names/keys to values. The main difference is that the `Enum` definition is immutable and global. It should be used whenever there is a closed set of possible values that can't change dynamically during program runtime, and especially if that set should be defined only once and globally. Dictionaries and named tuples are data containers. You can create as many instances of them as you like.

6.5 5. Custom containers

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
```

CHAPTER

SEVEN

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.

CHAPTER

EIGHT

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
```

CHAPTER

NINE

DOCSTRINGS AND ANNOTATIONS

9.1 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.

9.2 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'>}
```

9.2.1 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

TEN

CAVEATS IN PYTHON

10.1 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})"
```

10.2 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]
```

(continues on next page)

(continued from previous page)

```
'[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

ELEVEN

GENERAL TRAITS OF GOOD CODE

11.1 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).

11.1.1 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).

11.1.2 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.

11.1.3 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.

11.1.4 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.

11.2 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.

11.2.1 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:
```

(continues on next page)

(continued from previous page)

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

    def connect(self):
        for _ in range(self.retry_n_times):
            try:
                self.connection = self._connector.connect()
            except ConnectionError as e:
                logger.info(f"{e}: attempting new connection in {self.retry_
↪threshold}")
                time.sleep(self.retry_threshold)
            else:
                return self.connection

        raise ConnectionError(f"Couldn't connect after {self.retry_n_times} times")

    def send(self, data):
            return self.connection.send(data)
```

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

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

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

```
def connect_with_retry(connector, retry_n_times, retry_threshold=5):
    """Tries to establish the connection of <connector> retrying
    <retry_n_times>.
    If it can connect, returns the connection object.
    If it's not possible after the retries, raises ConnectionError
    :param connector: An object with a `.connect()` method.
    :param retry_n_times int: The number of times to try to call
    ``connector.connect()``.
    :param retry_threshold int: The time lapse between retry calls.
    """
    for _ in range(retry_n_times):
    try:
        return connector.connect()
    except ConnectionError as e:
        logger.info(f"{e}: attempting new connection in {retry_threshold}")
        time.sleep(retry_threshold)
    exc = ConnectionError(f"Couldn't connect after {retry_n_times} times")
    logger.exception(exc)
    raise exc
```

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

much more compact and easier to read:

```
class DataTransport:
    """An example of an object that separates the exception handling by
    abstraction levels.
    """
    retry_threshold: int = 5
    retry_n_times: int = 3

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

    def deliver_event(self, event):
        self.connection = connect_with_retry(self._connector, self.retry_n_times,␣
↪self.retry_threshold)
        self.send(event)

    def send(self, event):
        try:
            return self.connection.send(event.decode())
        except ValueError as e:
            logger.error(f"{event} contains incorrect data: {e}")
            raise
```

2.1.2.2 Do not expose tracebacks

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

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

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

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

2.1.2.3 Avoid empty except blocks

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

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

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

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

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

There are two alternatives:

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

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

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

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

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

2.1.2.4. Include the original exception

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

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

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

```
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.

11.2.2 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)
```

11.3 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.

11.3.1 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.

11.4 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.

11.4.1 4.1. DRY/OAOO

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

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

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

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

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

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

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

```
def process_students_list(students):
    # do some processing...
    students_ranking = sorted(students, key=lambda s: s.passed * 11 - s.failed * 5
↪- s.years * 2)

    # more processing
    for student in students_ranking:
        print(f"Name: {student.name}, Score: {student.passed * 11 - student.failed
↪* 5 - student.years * 2}")
```

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

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

```
def score_for_student(student):
    return student.passed * 11 - student.failed * 5 - student.years * 2

def process_students_list(students):
    # do some processing...
    students_ranking = sorted(students, key=score_for_student)
    # more processing
    for student in students_ranking:
    print(f"Name: {student.name}, Score: {core_for_student(student)}")
```

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

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

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

11.4.2 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.

11.4.3 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):
        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.

11.4.4 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.

11.5 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.

11.5.1 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`.

11.5.2 5.2. Anti-patterns for inheritance

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

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

For example, if we see that a class derived from `BaseHTTPRequestHandler` implements a method named `handle()`, it would make sense because it is overriding one of the parents. If it had any other method whose

name relates to an action that has to do with an HTTP request, then we could also think that is correctly placed (but we would not think that if we found something called process_purchase() on that class).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
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.

11.5.3 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.

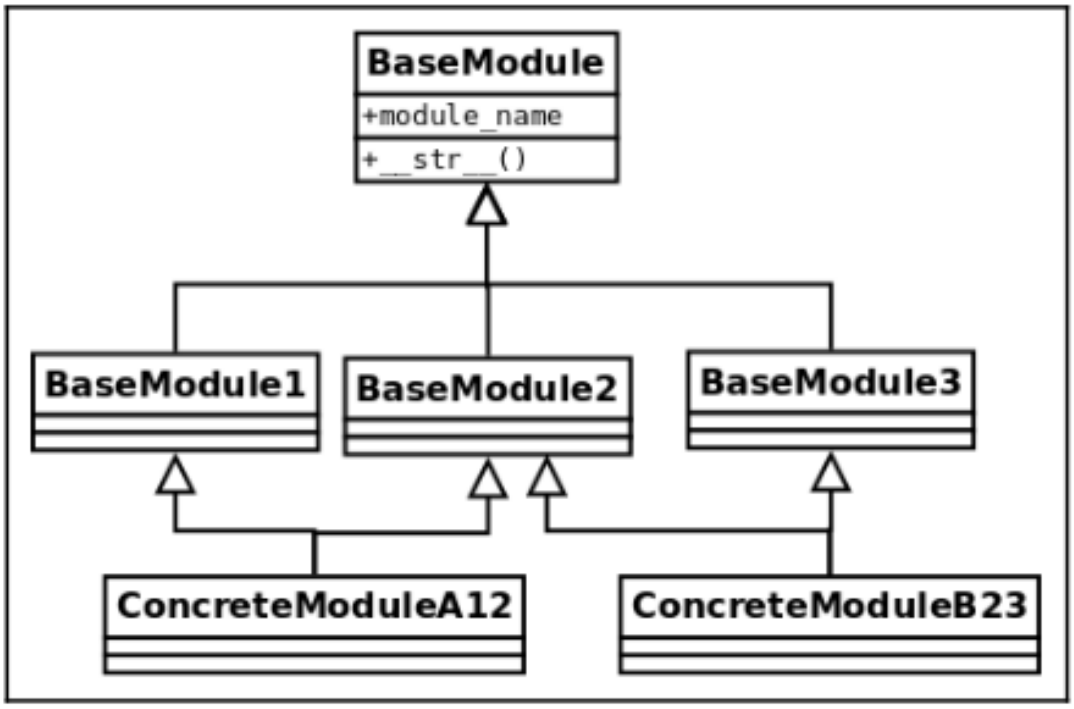
Multiple inheritance is a perfectly valid solution when used correctly, and this opens up new patterns (such as the adapter pattern) and mixins.

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"""
```

(continues on next page)

(continued from previous page)

```
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.

11.6 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.

11.6.1 6.1. How function arguments work in Python

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

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

6.1.1. How arguments are copied to functions

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

In the following we can see the difference:

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

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

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

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

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

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

6.1.2. Variable number of arguments

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

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

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

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

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

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

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

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

```
>>> def show(e, rest):
...     print("Element: {0} - Rest: {1}".format(e, rest))
```

(continues on next page)

(continued from previous page)

```
...
>>> 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

    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'}
```

11.6.2 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.

11.7 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.

11.7.1 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 the importance of minimizing side-effects in our code.

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

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

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

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

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

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

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

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

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

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

11.7.2 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

TWELVE

SOLID

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

12.1 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.

12.1.1 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.

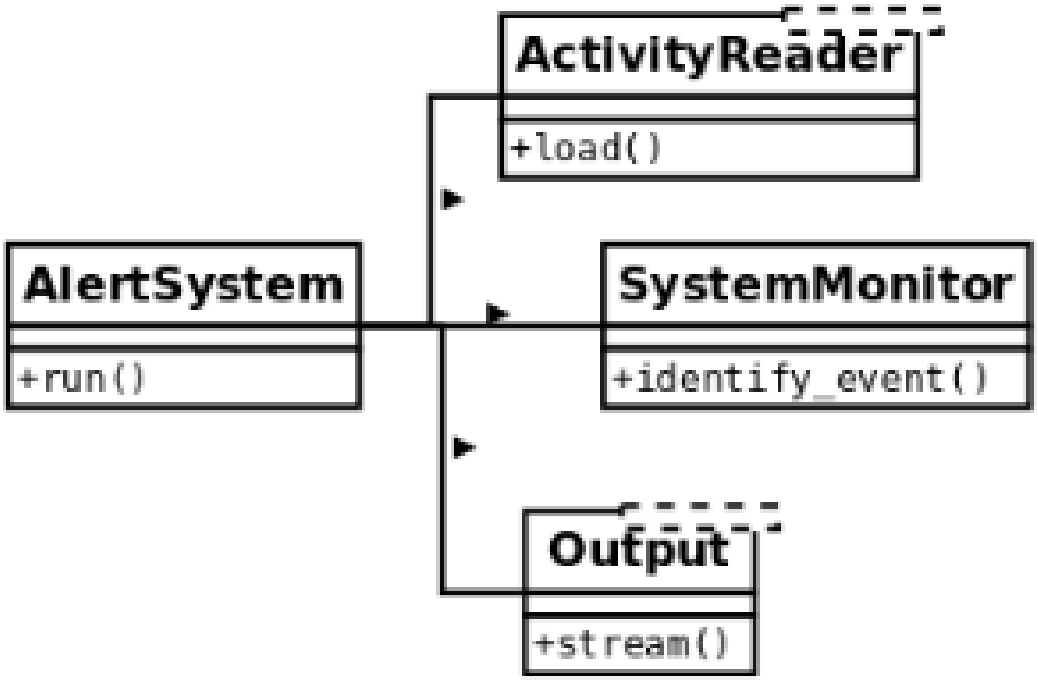
Consider the loader method, which retrieves the information from a particular source. Regardless of how this is done (we can abstract the implementation details here), it is clear that it will have its own sequence of steps, for instance connecting to the data source, loading the data, parsing it into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

12.1.2 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.

12.2 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.

12.2.1 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):
    """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)
```

(continues on next page)

(continued from previous page)

```
        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!).

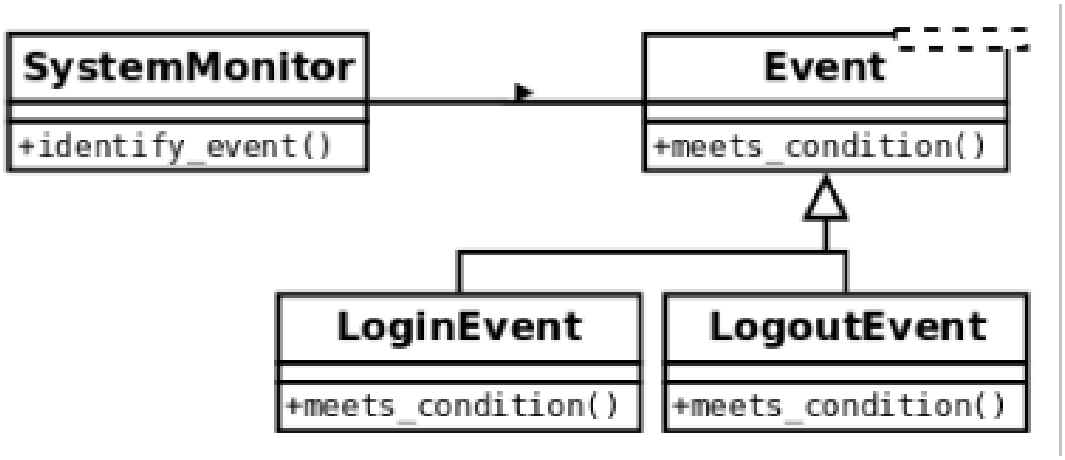
We want to be able to add new types of event without having to change this method (closed for modification). We also want to be able to support new types of event (open for extension) so that when a new event is added, we only have to add code, not change the code that already exists.

12.2.2 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)
```

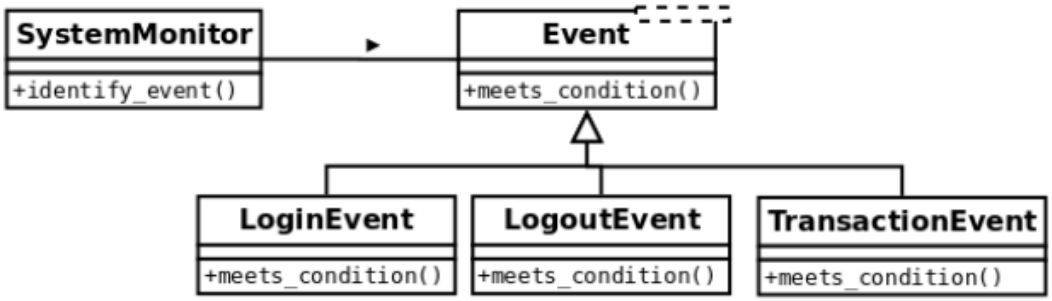
Notice how the interaction is now oriented toward an abstraction (in this case, it would be the generic base class `Event`, which might even be an abstract base class or an interface, but for the purposes of this example it is enough to have a concrete base class). The method no longer works with specific 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.

12.2.3 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.

12.2.4 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.

12.3 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.

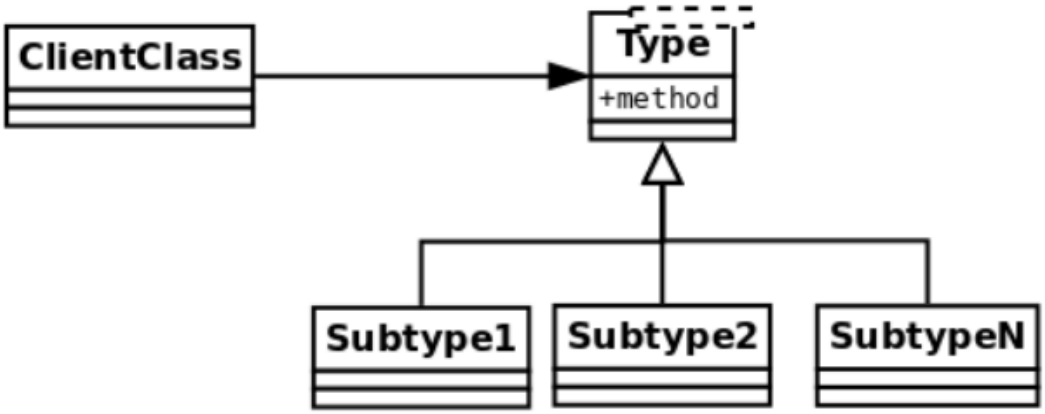
More formally, this is the original definition (LISKOV 01) of Liskov's substitution principle:

```
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.

12.3.1 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
```

12.3.2 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):
```

(continues on next page)

(continued from previous page)

```
        return False

    @staticmethod
    def meets_condition_pre(event_data: dict):
        """Precondition of the contract of this interface.
        Validate that the ``event_data`` parameter is properly formed.
        """
        assert isinstance(event_data, dict), f"{event_data!r} is not a dict"
        for moment in ("before", "after"):
            assert moment in event_data, f"{moment} not in {event_data}"
            assert isinstance(event_data[moment], dict)
```

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

```
class SystemMonitor:
    """Identify events that occurred in the system."""
    def __init__(self, event_data):
        self.event_data = event_data

    def identify_event(self):
        Event.meets_condition_pre(self.event_data)
        event_cls = next((event_cls for event_cls in Event.__subclasses__()
            if event_cls.meets_condition(self.event_data)), UnknownEvent)

        return event_cls(self.event_data)
```

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

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

```
class TransactionEvent(Event):
    """Represents a transaction that has just occurred on the system."""

    @staticmethod
    def meets_condition(event_data: dict):
        return event_data["after"].get("transaction") is not None
```

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

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

```
>>> l1 = SystemMonitor({"before": {"session": 0}, "after": {"session": 1}})
>>> l1.identify_event().__class__.__name__
'LoginEvent'
>>> l2 = SystemMonitor({"before": {"session": 1}, "after": {"session": 0}})
>>> l2.identify_event().__class__.__name__
'LogoutEvent'
>>> l3 = SystemMonitor({"before": {"session": 1}, "after": {"session": 1}})
>>> l3.identify_event().__class__.__name__
'UnknownEvent'
>>> l4 = SystemMonitor({"before": {}, "after": {"transaction": "Tx001"}})
>>> l4.identify_event().__class__.__name__
'TransactionEvent'
```

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

12.3.3 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.

12.4 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.

12.4.1 4.1. An interface that provides too much

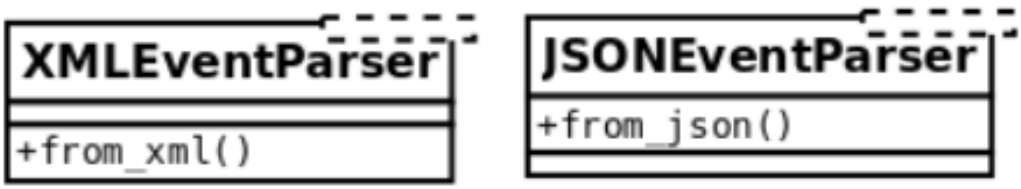
Now, we want to be able to parse an event from several data sources, in different formats (XML and JSON, for instance). Following good practice, we decide to target an interface as our dependency instead of a concrete class, and something like the following is devised:

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.

12.4.2 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).

12.4.3 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.

12.5 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.

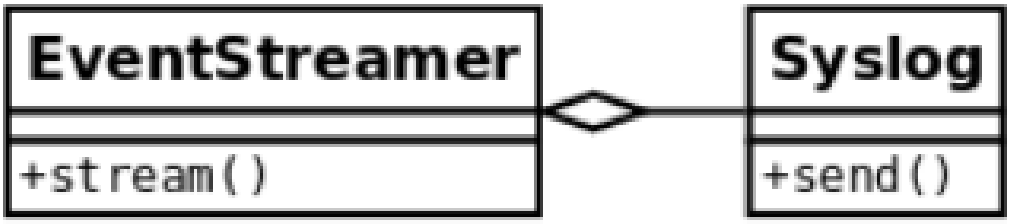
Imagine that two objects in our design need to collaborate, A and B. A works with an instance of B, but as it turns out, our module doesn't control B directly (it might be an external library, or a module maintained by another team, and so on). If our code heavily depends on B, when this changes the 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.

12.5.1 5.1. A case of rigid dependencies

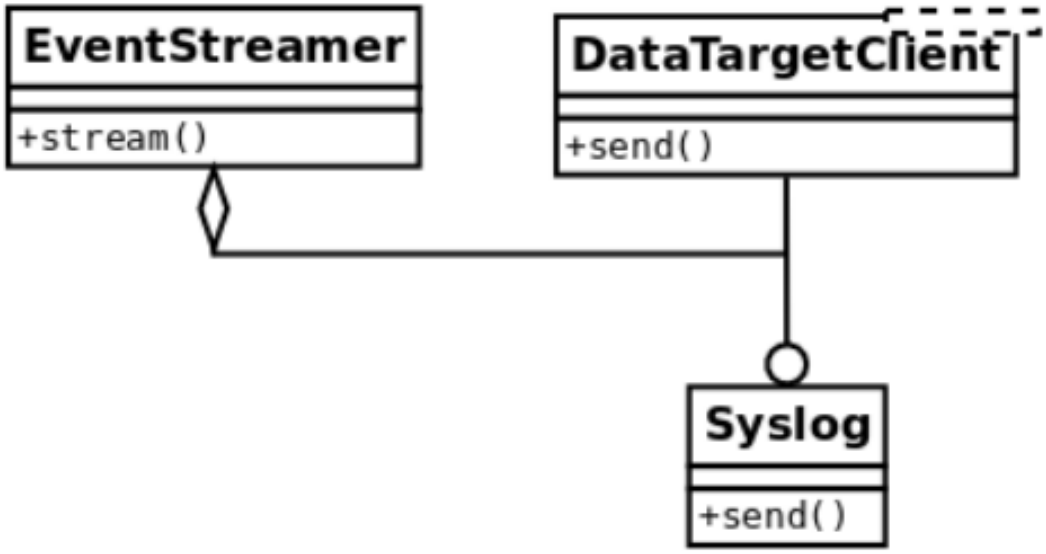
The last part of our event's monitoring system is to deliver the identified events to a data collector to be further analyzed. A naive implementation of such an idea would consist of having an event streamer class that interacts with a data destination, for example, `Syslog`:



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.

12.5.2 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

THIRTEEN

DECORATORS

13.1 1. What are decorators?

13.1.1 1.1. What are decorators in Python?

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

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

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

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

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

original = modifier(original)
```

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

The previous example could be rewritten like so:

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

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

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

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

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

13.1.2 1.2. Decorate functions

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

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

```
class ControlledException(Exception):
    """A generic exception on the program's domain."""

def retry(operation):

    @wraps(operation)
    def wrapped(*args, **kwargs):
        last_raised = None
        RETRIES_LIMIT = 3

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

    return wrapped
```

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

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

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

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

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

13.1.3 1.2. Decorate classes

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

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

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

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

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

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

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

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

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

class LoginEvent:
    SERIALIZER = LoginEventSerializer

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

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

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

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

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

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

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

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

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

def show_original(event_field):
    return event_field

class EventSerializer:
    def __init__(self, serialization_fields: dict) -> None:
        self.serialization_fields = serialization_fields

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

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

        event_class.serialize = serialize_method
    return event_class

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

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

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

Moreover, we could have another class decorator that, just by defining the attributes of the class, implements the logic of the init method, but this is beyond the scope of this example. This is what libraries such as `attrs` do,

and a similar functionality is proposed in for the Standard library.

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

```
from dataclasses import dataclass
from datetime import datetime

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

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

13.1.4 1.3. Other types of decorator

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

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

13.1.5 1.4. Passing arguments to decorators

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

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

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

1.4.1. Decorators with nested functions

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

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

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

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

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

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

```
RETRIES_LIMIT = 3

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

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

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

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

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

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

@with_retry(
    retries_limit=4, allowed_exceptions=(ZeroDivisionError, AttributeError)
```

(continues on next page)

(continued from previous page)

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

1.4.2. Decorator objects

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

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

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

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

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

            raise last_raised

        return wrapped
```

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

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

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

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

13.1.6 1.5. Good uses for decorators

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

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

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

1.5.1. Transforming parameters

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

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

1.5.2. Tracing code

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

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

13.2 2. Effective decorators: avoid common mistakes

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

13.2.1 2.1. Preserving data about the original wrapped object

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

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

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

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

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

But maybe there are changes.

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

Let's try to get help for this function:

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

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

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

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

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

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

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

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

    return wrapped
```

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

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

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

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

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

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

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

return decorated_function
```

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

13.2.2 2.2. Dealing with side-effects in decorators

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

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

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

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

2.2.1. Incorrect handling of side-effects in a decorator

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

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

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

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

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

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

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

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

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

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

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

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

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

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

With this new version, the previous problems are resolved.

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

2.2.2. Requiring decorators with side-effects

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

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

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

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

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

```
EVENTS_REGISTRY = {}

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

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

class UserEvent:
    TYPE = "user"

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

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

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

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

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

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

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

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

13.2.3 2.3. Creating decorators that will always work

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

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

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

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

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

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

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

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

```
import logging
from functools import wraps

logger = logging.getLogger(__name__)

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

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

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

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

    return wrapped

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

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

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

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

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

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

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

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

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

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

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

```
from functools import wraps
from types import MethodType

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

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

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

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

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

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

13.3 3. The DRY principle with decorators

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

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

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

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

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

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

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

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

13.4 4. Decorators and separation of concerns

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

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

```
def traced_function(function):

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

(continues on next page)

(continued from previous page)

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

    return wrapped
```

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

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

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

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

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

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

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

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

13.5 5. Analyzing good decorators

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

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

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

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

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

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

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

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

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

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

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

CHAPTER

FOURTEEN

PROPERTIES, ATTRIBUTES AND 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).

14.1 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'}
```

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.

14.2 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.

The properties provide a built-in descriptor type that knows how to link an attribute to a set of methods. `property` takes four optional arguments: `fget`, `fset`, `fdel`, and `doc`. The last one can be provided to define a `docstring` function that is linked to the attribute as if it were a method. Here is an example of a `Rectangle` class that can be controlled either by direct access to attributes that store two corner points or by using the `width` and `height` properties:

```
class Rectangle:

    def __init__(self, x1, y1, x2, y2):
        self.x1, self.y1 = x1, y1
        self.x2, self.y2 = x2, y2

    def _width_get(self):
        return self.x2 - self.x1

    def _width_set(self, value):
        self.x2 = self.x1 + value
```

(continues on next page)

(continued from previous page)

```
    def _height_get(self):
        return self.y2 - self.y1

    def _height_set(self, value):
        self.y2 = self.y1 + value

    width = property(
        _width_get, _width_set,
        doc="rectangle width measured from left"
    )
    height = property(
        _height_get, _height_set,
        doc="rectangle height measured from top"
    )

    def __repr__(self):
        return "{}({}, {}, {}, {})".format(
            self.__class__.__name__,
            self.x1, self.y1, self.x2, self.y2
        )
```

The following is an example of such defined properties in an interactive session:

```
>>> rectangle = Rectangle(10, 10, 25, 34)
>>> rectangle.width, rectangle.height
(15, 24)
>>> rectangle.width = 100
>>> rectangle
Rectangle(10, 10, 110, 34)
>>> rectangle.height = 100
>>> rectangle
Rectangle(10, 10, 110, 110)
>>> help(Rectangle)
Help on class Rectangle in module sample_module:
class Rectangle(builtins.object)
|   Methods defined here:
|
|   __init__(self, x1, y1, x2, y2)
|       Initialize self. See help(type(self)) for accurate signature.
|   __repr__(self)
|       Return repr(self).
|
| ----------------------------------------------------------
|   Data descriptors defined here:
|   (...)
|
|   height
|       rectangle height measured from top
|
|   width
|       rectangle width measured from left
```

The properties make it easier to write descriptors, but must be handled carefully when using inheritance over classes. The attribute created is made on the fly using the methods of the current class and will not use methods that are overridden in the derived classes.

For instance, the following example will fail to override the implementation of the `fget` method of the parent's class (`Rectangle`) `width` property:

```
>>> class MetricRectangle(Rectangle):
```

(continues on next page)

(continued from previous page)

```
...     def _width_get(self):
...         return "{} meters".format(self.x2 - self.x1)
...
>>> Rectangle(0, 0, 100, 100).width
100
```

In order to resolve this, the whole property simply needs to be overwritten in the derived class:

```
>>> class MetricRectangle(Rectangle):
...     def _width_get(self):
...         return "{} meters".format(self.x2 - self.x1)
...     width = property(_width_get, Rectangle.width.fset)
...
>>> MetricRectangle(0, 0, 100, 100).width
'100 meters'
```

Unfortunately, the preceding code has some maintainability issues. It can be a source of confusion if the developer decides to change the parent class, but forgets to update the property call. This is why overriding only parts of the property behavior is not advised. Instead of relying on the parent class's implementation, it is recommended that you rewrite all the property methods in the derived classes if you need to change how they work. In most cases, this is the only option, because usually the change to the property `setter` behavior implies a change to the behavior of `getter` as well.

Because of this, the best syntax for creating properties is to use property as a decorator. This will reduce the number of method signatures inside the class and make the code more readable and maintainable:

```
class Rectangle:
    def __init__(self, x1, y1, x2, y2):
        self.x1, self.y1 = x1, y1
        self.x2, self.y2 = x2, y2

    @property
    def width(self):
        """rectangle width measured from left"""
        return self.x2 - self.x1

    @width.setter
    def width(self, value):
        self.x2 = self.x1 + value

    @property
    def height(self):
        """rectangle height measured from top"""
        return self.y2 - self.y1

    @height.setter
    def height(self, value):
        self.y2 = self.y1 + value
```

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. 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.

14.3 3. Slots

An interesting feature that is very rarely used by developers is slots. They allow you to set a static attribute list for a given class with the `__slots__` attribute, and skip the creation of the `__dict__` dictionary in each instance of the class. They were intended to save memory space for classes with very few attributes, since `__dict__` is not created at every instance.

When a class defines the `__slots__` attribute, it can contain all the attributes that the class expects and no more.

Trying to add extra attributes dynamically to a class that defines `__slots__` will result in an `AttributeError`. By defining this attribute, the class becomes static, so it will not have a `__dict__` attribute where you can add more objects dynamically.

How, then, are its attributes retrieved if not from the dictionary of the object? By using descriptors. Each name defined in a slot will have its own descriptor that will store the value for retrieval later.

`__slots__` can help to design classes whose signature needs to be frozen. For instance, if you need to restrict the dynamic features of the language over a class, defining slots can help:

```
class Coordinate2D:
    __slots__ = ("lat", "lon")
    def __init__(self, lat, lon):
        self.lat = lat
        self.lon = lon

    def __repr__(self):
        return f"{self.__class__.__name__}({self.lat}, {self.lon})"
```

While this is an interesting feature, it has to be used with caution because it is taking away the dynamic nature of Python. In general, this ought to be reserved only for objects that we know are static, and if we are absolutely sure we are not adding any attributes to them dynamically in other parts of the code. Some techniques, such as monkey patching, will not work with instances of classes that have slots defined. Fortunately, the new attributes can be added to the derived classes if they do not have their own slots defined:

```
>>> class Frozen:
...     __slots__ = ['ice', 'cream']
...
>>> '__dict__' in dir(Frozen)
False
>>> 'ice' in dir(Frozen)
True
>>> frozen = Frozen()
```

(continues on next page)

(continued from previous page)

```
>>> frozen.ice = True
>>> frozen.cream = None
>>> frozen.icy = True
Traceback (most recent call last): File "<input>", line 1, in <module>
AttributeError: 'Frozen' object has no attribute 'icy'


>>> class Unfrozen(Frozen):
...     pass
...
>>> unfrozen = Unfrozen()
>>> unfrozen.icy = False
>>> unfrozen.icy
False
```

As an upside of this, objects defined with slots use less memory, since they only need a fixed set of fields to hold values and not an entire dictionary.

CHAPTER

FIFTEEN

DATA CLASSES

Before we dive deeper into details of Python classes, we will take a small detour. We will discuss a relatively new addition to the Python language, which are data classes. The `dataclasses` module, introduced in Python 3.7, provides a decorator and function that allows you to easily add generated special methods to your own classes.

Consider the following example. We are building a program that does some geometric computation and want to have a class that allows us to hold information about two- dimensional vectors. We will display the data of the vectors on the screen and perform common mathematical operations, such as addition, subtraction, and equality comparison. We already know that we can use special methods to achieve that goal. We can implement our `Vector` class as follows:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        """Add two vectors using + operator"""
        return Vector(
            self.x + other.x,
            self.y + other.y
        )

    def __sub__(self, other):
        """Subtract two vectors using - operator"""
        return Vector(
            self.x - other.x,
            self.y - other.y
        )

    def __repr__(self):
        """Return textual representation of vector"""
        return f"<Vector: x={self.x}, y={self.y}>"

    def __eq__(self, other):
        """Compare two vectors for equality"""
        return self.x == other.x and self.y == other.y
```

The following is the interactive session example that shows how it behaves when used with common operators:

```
>>> Vector(2, 3)
<Vector: x=2, y=3)
>>> Vector(5, 3) + Vector(1, 2)
<Vector: x=6, y=5>
>>> Vector(5, 3) - Vector(1, 2)
<Vector: x=4, y=1>
>>> Vector(1, 1) == Vector(2, 2)
False
>>> Vector(2, 2) == Vector(2, 2)
True
```

The preceding vector implementation is quite simple, but involves a lot of repetitive code that could be avoided. If your program uses many similar simple classes that do not require complex initialization, you'll end up writing a lot of boilerplate code just for the `__init__()`, `__repr__()`, and `__eq__()` methods.

With the `dataclasses` module, we can make our `Vector` class code a lot shorter:

```
from dataclasses import dataclass


@dataclass
class Vector:
    x: int
    y: int

    def __add__(self, other):
        """Add two vectors using + operator"""
        return Vector(
            self.x + other.x,
            self.y + other.y,
        )

    def __sub__(self, other):
        """Subtract two vectors using - operator"""
        return Vector(
            self.x - other.x,
            self.y - other.y,
        )
```

The dataclass class decorator reads annotations of the Vector class attribute and automatically creates the `__init__()`, `__repr__()`, and `__eq__()` methods. The default equality comparison assumes that two instances are equal if all their respective attributes are equal to each other.

But that's not all. Data classes offer many useful features. They can easily be made compatible with other Python protocols, too. Let's assume we want our `Vector` class instances to be immutable. Thanks to this, they could be used as dictionary keys and as content sets. You can do this by simply adding a `frozen=True` argument to the dataclass decorator, as in the following example:

```
@dataclass(frozen=True)
class FrozenVector:
    x: int
    y: int
```

Such a frozen `Vector` data class becomes completely immutable, so you won't be able to modify any of its attributes. You can still add and subtract two Vector instances as in our example; these operations simply create new `Vector` objects.

The final piece of useful information we will cover about data classes in this chapter is that you can define default values for specific attributes using the `field()` constructor. You can use both static values and constructors of other objects. Consider the following example:

```
>>> @dataclass
... class DataClassWithDefaults:
...     static_default: str = field(default="this is static default value")
...     factory_default: list = field(default_factory=list)
...
>>> DataClassWithDefaults()
DataClassWithDefaults(static_default='this is static default value', factory_
↪default=[])
```

CHAPTER

SIXTEEN

DESCRIPTORS

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

A descriptor lets you customize what should be done when you refer to an attribute of an object.

Descriptors are the base of a complex attribute access in Python. They are used internally to implement properties, methods, class methods, static methods, and the `super` type. They are classes that define how attributes of another class can be accessed. In other words, a class can delegate the management of an attribute to another one.

Important: Like every advanced Python syntax feature, this one should also be used with caution and documented well in code. For inexperienced developers, the altered class behavior might be very confusing and unexpected, because descriptors affect the very basic part of class behavior. Because of that, it is very important to make sure that all your team members are familiar with descriptors and understand this concept well if it plays an important role in your project's code base.

16.1 1. A first look at descriptors

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

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

16.1.1 1.1. The machinery behind descriptors

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

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

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

- `__get__(self, obj, owner=None)`: This is called whenever the attribute is read (referred to as a `getter`)
- `__set__(self, obj, value)`: .This is called whenever the attribute is set. In the following examples, I will refer to this as a `setter`.

- `__delete__(self, obj)`: This is called when `del` is invoked on the attribute.
- `__set_name__(self, owner, name)`

A descriptor that implements `__get__()` and `__set__()` is called a data descriptor. If it just implements `__get__()`, then it is called a non-data descriptor.

Methods of this protocol are, in fact, called by the object's special `__getattribute__()` method (do not confuse it with `__getattr__()`, which has a different purpose) on every attribute lookup. Whenever such a lookup is performed, either by using a dotted notation in the form of `instance.attribute`, or by using the `getattr(instance, 'attribute')` function call, the `__getattribute__()` method is implicitly invoked and it looks for an attribute in the following order:

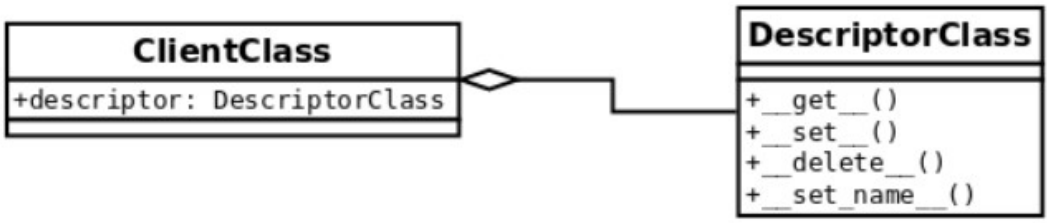
1. It verifies whether the attribute is a data descriptor on the class object of the instance.
2. If not, it looks to see whether the attribute can be found in the `__dict__` lookup of the instance object.
3. Finally, it looks to see whether the attribute is a non-data descriptor on the class object of the instance.

In other words, data descriptors take precedence over `__dict__` lookup, and `__dict__` lookup takes precedence over non-data descriptors. Without `__dict__` taking precedence over non-data descriptors, we would not be able to dynamically override specific methods on already constructed instances at runtime. Fortunately, thanks to how descriptors work in Python, it is possible; so, developers may use a popular technique called monkey patching to change the way in which instances work without the need for subclassing.

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

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

This relationship is illustrated in the following diagram:



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

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

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

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

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

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

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

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

```
class DescriptorClass:

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

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

class ClientClass:
    descriptor = DescriptorClass()
```

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

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

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

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

16.1.2 1.2. Exploring each method of the descriptor protocol

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

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

In this section, we will explore each method of the descriptor protocol, in full detail, explaining what each parameter signifies, and how they are intended to be used.

1.2.1. __get__(self, instance, owner)

The first parameter, `instance`, refers to the object from which the *descriptor* is being called. In our first example, this would mean the *client* object.

The `owner` parameter is a reference to the class of that object, which following our example would be *ClientClass*.

From the previous paragraph we conclude that the parameter named `instance` in the signature of `__get__` is the object over which the *descriptor* is taking action, and `owner` is the class of `instance`. The avid reader might be wondering why is the signature define like this, after all the class can be taken from `instance` directly (`owner = instance.__class__`). There is an edge case: when the *descriptor* is called from the class (*ClientClass*), not from the instance (*client*), then the value of `instance` is None, but we might still want to do some processing in that case.

With the following simple code we can demonstrate the difference of when a descriptor is being called from the class, or from an instance. In this case, the `__get__` method is doing two separate things for each case.

```
class DescriptorClass:
    def __get__(self, instance, owner):
        if instance is None:
            return f"{self.__class__.__name__}.{owner.__name__}"
        return f"value for {instance}"

class ClientClass:
    descriptor = DescriptorClass()
```

When we call it from *ClientClass* directly it will do one thing, which is composing a namespace with the names of the classes:

```
>>> ClientClass.descriptor
'DescriptorClass.ClientClass'
```

And then if we call it from an object we have created, it will return the other message instead:

```
>>> ClientClass().descriptor
'value for <descriptors_methods_1.ClientClass object at 0x...>'
```

In general, unless we really need to do something with the `owner` parameter, the most common idiom, is to just return the descriptor itself, when instance is None.

1.2.2. __set__(self, instance, value)

This method is called when we try to assign something to a *descriptor*. It is activated with statements such as the following, in which a *descriptor* is an object that implements `__set__()`. The `instance` parameter, in this case, would be *client*, and the value would be the "value" string: `client.descriptor = "value"`

If `client.descriptor` doesn't implement `__set__()`, then "value" will override the descriptor entirely.

Note: Be careful when assigning a value to an attribute that is a descriptor. Make sure it implements the `__set__` method, and that we are not causing an undesired side effect.

By default, the most common use of this method is just to store data in an object. Nevertheless, we have seen how powerful descriptors are so far, and that we can take advantage of them, for example, if we were to create generic validation objects that can be applied multiple times (again, this is something that if we don't abstract, we might end up repeating multiple times in setter methods of properties).

The following listing illustrates how we can take advantage of this method in order to create generic validation objects for attributes, which can be created dynamically with functions to validate on the values before assigning them to the object:

```
class Validation:

    def __init__(self, validation_function, error_msg: str):
        self.validation_function = validation_function
        self.error_msg = error_msg

    def __call__(self, value):
        if not self.validation_function(value):
            raise ValueError(f"{value!r} {self.error_msg}")

class Field:

    def __init__(self, *validations):
        self._name = None
        self.validations = validations

    def __set_name__(self, owner, name):
        self._name = name

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

        return instance.__dict__[self._name]

    def validate(self, value):
        for validation in self.validations:
            validation(value)

    def __set__(self, instance, value):
        self.validate(value)
        instance.__dict__[self._name] = value

class ClientClass:
    descriptor = Field(
        Validation(lambda x: isinstance(x, (int, float)), "is not a number"),
        Validation(lambda x: x >= 0, "is not >= 0")
    )
```

We can see this object in action in the following listing:

```
>>> client = ClientClass()
>>> client.descriptor = 42
>>> client.descriptor
42
>>> client.descriptor = -42
Traceback (most recent call last):
...
ValueError: -42 is not >= 0
>>> client.descriptor = "invalid value"
...
ValueError: 'invalid value' is not a number
```

The idea is that something that we would normally place in a property can be abstracted away into a *descriptor*, and reuse it multiple times. In this case, the `__set__()` method would be doing what the `@property.setter` would have been doing.

1.2.3. __delete__(self, instance)

This method is called upon with the following statement, in which `self` would be the *descriptor* attribute, and `instance` would be the client object in this example:

```
>>> del client.descriptor
```

In the following example, we use this method to create a *descriptor* with the goal of preventing you from removing attributes from an object without the required administrative privileges. Notice how, in this case, that the *descriptor* has logic that is used to predicate with the values of the object that is using it, instead of different related objects:

```
class ProtectedAttribute:

    def __init__(self, requires_role=None) -> None:
        self.permission_required = requires_role
        self._name = None

    def __set_name__(self, owner, name):
        self._name = name

    def __set__(self, user, value):
        if value is None:
            raise ValueError(f"{self._name} can't be set to None")

        user.__dict__[self._name] = value

    def __delete__(self, user):
        if self.permission_required in user.permissions:
            user.__dict__[self._name] = None
        else:
            raise ValueError(f"User {user!s} doesn't have {self.permission_
↪required} permission")

class User:
    """Only users with "admin" privileges can remove their email
    address."""

    email = ProtectedAttribute(requires_role="admin")

    def __init__(self, username: str, email: str, permission_list: list = None) ->␣
↪None:
        self.username = username
        self.email = email
        self.permissions = permission_list or []
```

(continues on next page)

(continued from previous page)

```
    def __str__(self):
        return self.username
```

Before seeing examples of how this object works, it's important to remark some of the criteria of this descriptor. Notice the `User` class requires the `username` and `email` as mandatory parameters. According to its `__init__` method, it cannot be a user if it doesn't have an email attribute. If we were to delete that attribute, and extract it from the object entirely we would be creating an inconsistent object, with some invalid intermediate state that does not correspond to the interface defined by the class `User`. Details like this one are really important, in order to avoid issues. Some other object is expecting to work with this `User`, and it also expects that it has an email attribute.

For this reason, it was decided that the "deletion" of an email will just simply set it to None. For the same reason, we must forbid someone trying to set a None value to it, because that would bypass the mechanism we placed in the `__delete__` method.

Here, we can see it in action, assuming a case where only users with "admin" privileges can remove their email address:

```
>>> admin = User("root", "root@d.com", ["admin"])
>>> user = User("user", "user1@d.com", ["email", "helpdesk"])
>>> admin.email
'root@d.com'
>>> del admin.email
>>> admin.email is None
True
>>> user.email
'user1@d.com'
>>> user.email = None
ValueError: email can't be set to None
>>> del user.email
ValueError: User user doesn't have admin permission
```

Here, in this simple *descriptor*, we see that we can delete the email from users that contain the "admin" permission only. As for the rest, when we try to call `del` on that attribute, we will get a `ValueError` exception.

In general, this method of the *descriptor* is not as commonly used as the two previous ones, but it is worth showing it for completeness.

1.2.4. __set_name__(self, owner, name)

When we create the *descriptor* object in the class that is going to use it, we generally need the *descriptor* to know the name of the attribute it is going to be handling.

This attribute name is the one we use to read from and write to `__dict__` in the `__get__` and `__set__` methods, respectively.

Before Python 3.6, the descriptor couldn't take this name automatically, so the most general approach was to just pass it explicitly when initializing the object. This works fine, but it has an issue in that it requires that we duplicate the name every time we want to use the descriptor for a new attribute.

This is what a typical *descriptor* would look like if we didn't have this method:

```
class DescriptorWithName:

    def __init__(self, name):
        self.name = name

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

(continues on next page)

(continued from previous page)

```
        logger.info("getting %r attribute from %r", self.name, instance)
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value

class ClientClass:
    descriptor = DescriptorWithName("descriptor")
```

We can see how the descriptor uses this value:

```
>>> client = ClientClass()
>>> client.descriptor = "value"
>>> client.descriptor
INFO:getting 'descriptor' attribute from <ClientClass object at 0x...>
'value'
```

Now, if we wanted to avoid writing the name of the attribute twice (once for the variable assigned inside the class, and once again as the name of the first parameter of the descriptor), we have to resort to a few tricks, like using a class decorator, or (even worse) using a metaclass.

In Python 3.6, the new method `__set_name__` was added, and it receives the class where that descriptor is being created, and the name that is being given to that descriptor. The most common idiom is to use this method for the descriptor so that it can store the required name in this method.

For compatibility, it is generally a good idea to keep a default value in the `__init__` method but still take advantage of `__set_name__`.

With this method, we can rewrite the previous descriptors as follows:

```
class DescriptorWithName:
    def __init__(self, name=None):
        self.name = name

    def __set_name__(self, owner, name):
        self.name = name

    ...
```

16.2 2. Types of descriptors

Based on the methods we have just explored, we can make an important distinction among descriptors in terms of how they work. Understanding this distinction plays an important role in working effectively with descriptors, and will also help to avoid caveats or common errors at runtime.

If a descriptor implements the `__set__` or `__delete__` methods, it is called a **data descriptor**. Otherwise, a descriptor that solely implements `__get__` is a **non-data descriptor**. Notice that `__set_name__` does not affect this classification at all.

When trying to resolve an attribute of an object, a data descriptor will always take precedence over the dictionary of the object, whereas a non-data descriptor will not. This means that in a non-data descriptor if the object has a key on its dictionary with the same name as the descriptor, this one will always be called, and the descriptor itself will never run. Conversely, in a data descriptor, even if there is a key in the dictionary with the same name as the descriptor, this one will never be used since the descriptor itself will always end up being called.

The following two sections explain this in more detail, with examples, in order to get a deeper idea of what to expect from each type of descriptor.

16.2.1 2.1. Non-data descriptors

We will start with a descriptor that only implements the __get__ method, and see how it is used:

```
class NonDataDescriptor:
    def __get__(self, instance, owner):
        if instance is None:
            return self
        return 42

class ClientClass:
    descriptor = NonDataDescriptor()
```

As usual, if we ask for the descriptor, we get the result of its __get__ method:

```
>>> client = ClientClass()
>>> client.descriptor
42
```

But if we change the descriptor attribute to something else, we lose access to this value, and get what was assigned to it instead:

```
>>> client.descriptor = 43
>>> client.descriptor
43
```

Now, if we delete the descriptor, and ask for it again, let's see what we get:

```
>>> del client.descriptor
>>> client.descriptor
42
```

Let's rewind what just happened. When we first created the client object, the descriptor attribute lay in the class, not the instance, so if we ask for the dictionary of the client object, it will be empty:

```
>>> vars(client)
{}
```

And then, when we request the .descriptor attribute, it doesn't find any key in client.__dict__ named "descriptor", so it goes to the class, where it will find it, but only as a descriptor, hence why it returns the result of the __get__ method.

But then, we change the value of the .descriptor attribute to something else, and what this does is set this into the dictionary of the instance, meaning that this time it won't be empty:

```
>>> client.descriptor = 99
>>> vars(client)
{'descriptor': 99}
```

So, when we ask for the .descriptor attribute here, it will look for it in the object (and this time it will find it, because there is a key named descriptor in the __dict__ attribute of the object, as the vars result is showing us), and return it without having to look for it in the class. For this reason, the descriptor protocol is never invoked, and the next time we ask for this attribute, it will instead return the value we have overridden it with (99).

Afterward, we delete this attribute by calling del, and what this does is remove the key "descriptor" from the dictionary of the object, leaving us back in the first scenario, where it's going to default to the class where the descriptor protocol will be activated:

```
>>> del client.descriptor
>>> vars(client)
{}
>>> client.descriptor
42
```

This means that if we set the attribute of the descriptor to something else, we might be accidentally breaking it. Why? Because the descriptor doesn't handle the delete action (some of them don't need to).

This is called a non-data descriptor because it doesn't implement the `__set__` magic method, as we will see in the next example.

16.2.2 2.2. Data descriptors

Now, let's look at the difference of using a data descriptor. For this, we are going to create another simple descriptor that implements the `__set__` method:

```
class DataDescriptor:
    def __get__(self, instance, owner):
        if instance is None:
            return self
        return 42

    def __set__(self, instance, value):
        logger.debug("setting %s.descriptor to %s", instance, value)
        instance.__dict__["descriptor"] = value

class ClientClass:
    descriptor = DataDescriptor()
```

Let's see what the value of the descriptor returns:

```
>>> client = ClientClass()
>>> client.descriptor
42
```

Now, let's try to change this value to something else, and see what it returns instead:

```
>>> client.descriptor = 99
>>> client.descriptor
42
```

The value returned by the descriptor didn't change. But when we assign a different value to it, it must be set to the dictionary of the object (as it was previously):

```
>>> vars(client)
{'descriptor': 99}
>>> client.__dict__["descriptor"]
99
```

So, the `__set__()` method was called, and indeed it did set the value to the dictionary of the object, only this time, when we request this attribute, instead of using the `__dict__` attribute of the dictionary, the descriptor takes precedence (because it's an overriding descriptor).

One more thing: deleting the attribute will not work anymore:

```
>>> del client.descriptor
Traceback (most recent call last):
...
AttributeError: __delete__
```

The reason is as follows: given that now, the descriptor always takes place, calling `del` on an object doesn't try to delete the attribute from the dictionary (`__dict__`) of the object, but instead it tries to call the `__delete__()` method of the descriptor (which is not implemented in this example, hence the attribute error).

This is the difference between data and non-data descriptors. If the descriptor implements `__set__()`, then it will always take precedence, no matter what attributes are present in the dictionary of the object. If this method is not implemented, then the dictionary will be looked up first, and then the descriptor will run.

An interesting observation you might have noticed is this line on the set method: `instance.__dict__["descriptor"] = value`. There are a lot of things to question about that line, but let's break it down into parts.

First, why is it altering just the name of a "descriptor" attribute? This is just a simplification for this example, but, as it transpires when working with descriptors, it doesn't know at this point the name of the parameter it was assigned to, so we just used the one from the example, knowing that it was going to be "descriptor".

In a real example, you would do one of two things: either receive the name as a parameter and store it internally in the `init` method, so that this one will just use the internal attribute, or, even better, use the `__set_name__` method.

Why is it accessing the `__dict__` attribute of the instance directly? Another good question, which also has at least two explanations. First, you might be thinking why not just do the following: `setattr(instance, "descriptor", value)`. Remember that this method (`__set__`) is called when we try to assign something to the attribute that is a descriptor. So, using `setattr()` will call this descriptor again, which, in turn, will call it again, and so on and so forth. This will end up in an infinite recursion.

Note: Do not use `setattr()` or the assignment expression directly on the descriptor inside the `__set__` method because that will trigger an infinite recursion.

Why, then, is the descriptor not able to book-keep the values of the properties for all of its objects?

The client class already has a reference to the descriptor. If we add a reference from the descriptor to the client object, we are creating circular dependencies, and these objects will never be garbage-collected. Since they are pointing at each other, their reference counts will never drop below the threshold for removal.

A possible alternative here is to use weak references, with the `weakref` module, and create a weak reference key dictionary if we want to do that. This implementation is explained later on, but we prefer to use this idiom, since it is fairly common and accepted when writing descriptors.

16.3 3. Descriptors in action

Now that we have seen what descriptors are, how they work, and what the main ideas behind them are, we can see them in action. In this section, we will be exploring some situations that can be elegantly addressed through descriptors.

Here, we will look at some examples of working with descriptors, and we will also cover implementation considerations for them (different ways of creating them, with their pros and cons), and finally we will discuss what are the most suitable scenarios for descriptors.

16.3.1 3.1. An application of descriptors

We will start with a simple example that works, but that will lead to some code duplication. It is not very clear how this issue will be addressed. Later on, we will devise a way of abstracting the repeated logic into a descriptor, which will address the duplication problem, and we will notice that the code on our client classes will be reduced drastically.

3.1.1. A first attempt without using descriptors

The problem we want to solve now is that we have a regular class with some attributes, but we wish to track all of the different values a particular attribute has over time, for example, in a list. The first solution that comes to our mind is to use a property, and every time a value is changed for that attribute in the setter method of the property, we add it to an internal list that will keep this trace as we want it.

Imagine that our class represents a traveler in our application that has a current city, and we want to keep track of all the cities that user has visited throughout the running of the program. The following code is a possible implementation that addresses these requirements:

```
class Traveller:
    def __init__(self, name, current_city):
        self.name = name
        self._current_city = current_city
        self._cities_visited = [current_city]

    @property
    def current_city(self):
        return self._current_city

    @current_city.setter
    def current_city(self, new_city):
        if new_city != self._current_city:
            self._cities_visited.append(new_city)
        self._current_city = new_city

    @property
    def cities_visited(self):
        return self._cities_visited
```

We can easily check that this code works according to our requirements:

```
>>> alice = Traveller("Alice", "Barcelona")
>>> alice.current_city = "Paris"
>>> alice.current_city = "Brussels"
>>> alice.current_city = "Amsterdam"
>>> alice.cities_visited
['Barcelona', 'Paris', 'Brussels', 'Amsterdam']
```

So far, this is all we need and nothing else has to be implemented. For the purposes of this problem, the property would be more than enough. What happens if we need the exact same logic in multiple places of the application? This would mean that this is actually an instance of a more generic problem: tracing all the values of an attribute in another one. What would happen if we want to do the same with other attributes, such as keeping track of all tickets Alice bought, or all the countries she has been in? We would have to repeat the logic in all of these places.

Moreover, what would happen if we need this same behavior in different classes? We would have to repeat the code or come up with a generic solution (maybe a decorator, a property builder, or a descriptor).

3.1.2. The idiomatic implementation

We will now look at how to address the questions of the previous section by using a descriptor that is generic enough as to be applied in any class. Again, this example is not really needed because the requirements do not specify such generic behavior (we haven't even followed the rule of three instances of the similar pattern previously creating the abstraction), but it is shown with the goal of portraying descriptors in action.

Note: Do not implement a descriptor unless there is actual evidence of the repetition we are trying to solve, and the complexity is proven to have paid off.

Now, we will create a generic descriptor that, given a name for the attribute to hold the traces of another one, will store the different values of the attribute in a list.

As we mentioned previously, the code is more than what we need for the problem, but its intention is just to show how a descriptor would help us in this case. Given the generic nature of descriptors, the reader will notice that the logic on it (the name of their method, and attributes) does not relate to the domain problem at hand (a traveler object). This is because the idea of the descriptor is to be able to use it in any type of class, probably on different projects, with the same outcomes.

In order to address this gap, some parts of the code are annotated, and the respective explanation for each section (what it does, and how it relates to the original problem) is described in the following code:

```
class HistoryTracedAttribute:
    def __init__(self, trace_attribute_name) -> None:
        self.trace_attribute_name = trace_attribute_name # [1]
        self._name = None

    def __set_name__(self, owner, name):
        self._name = name

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

        return instance.__dict__[self._name]

    def __set__(self, instance, value):
        self._track_change_in_value_for_instance(instance, value)
        instance.__dict__[self._name] = value

    def _track_change_in_value_for_instance(self, instance, value):
        self._set_default(instance) # [2]
        if self._needs_to_track_change(instance, value):
            instance.__dict__[self.trace_attribute_name].append(value)

    def _needs_to_track_change(self, instance, value) -> bool:
        try:
            current_value = instance.__dict__[self._name]
        except KeyError: # [3]
            return True

        return value != current_value # [4]

    def _set_default(self, instance):
        instance.__dict__.setdefault(self.trace_attribute_name, []) # [6]

class Traveller:
    current_city = HistoryTracedAttribute("cities_visited") # [1]

    def __init__(self, name, current_city):
        self.name = name
        self.current_city = current_city # [5]
```

Some annotations and comments on the code are as follows (numbers in the list correspond to the number annotations in the previous listing):

1. The name of the attribute is one of the variables assigned to the descriptor, in this case, `current_city`. We pass to the descriptor the name of the variable in which it will store the trace for the variable of the descriptor. In this example, we are telling our object to keep track of all the values that `current_city` has had in the attribute named `cities_visited`.
2. The first time we call the descriptor, in the `init`, the attribute for tracing values will not exist, in which case we initialize it to an empty list to later append values to it.

3. In the `init` method, the name of the attribute `current_city` will not exist either, so we want to keep track of this change as well. This is the equivalent of initializing the list with the first value in the previous example.
4. Only track changes when the new value is different from the one that is currently set.
5. In the `init method`, the descriptor already exists, and this assignment instruction triggers the actions from step 2 (create the empty list to start tracking values for it), and step 3 (append the value to this list, and set it to the key in the object for retrieval later).
6. The `setdefault` method in a dictionary is used to avoid a `KeyError`. In this case an empty list will be returned for those attributes that aren't still available.

It is true that the code in the descriptor is rather complex. On the other hand, the code in the client class is considerably simpler. Of course, this balance only pays off if we are going to use this descriptor multiple times, which is a concern we have already covered.

What might not be so clear at this point is that the descriptor is indeed completely independent from the client class. Nothing in it suggests anything about the business logic. This makes it perfectly suitable to apply it in any other class; even if it does something completely different, the descriptor will take the same effect.

This is the true Pythonic nature of descriptors. They are more appropriate for defining libraries, frameworks, or internal APIs, and not that much for business logic.

16.3.2 3.2. Different forms of implementing descriptors

We have to first understand a common issue that's specific to the nature of descriptors before thinking of ways of implementing them. First, we will discuss the problem of a global shared state, and afterward we will move on and look at different ways descriptors can be implemented while taking this into consideration.

16.3.3 3.2.1. The issue of global shared state

As we have already mentioned, descriptors need to be set as class attributes to work. This should not be a problem most of the time, but it does come with some warnings that need to be taken into consideration.

The problem with class attributes is that they are shared across all instances of that class. Descriptors are not an exception here, so if we try to keep data in a descriptor object, keep in mind that all of them will have access to the same value.

Let's see what happens when we incorrectly define a descriptor that keeps the data itself, instead of storing it in each object:

```
class SharedDataDescriptor:
    def __init__(self, initial_value):
        self.value = initial_value

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

    def __set__(self, instance, value):
        self.value = value

    class ClientClass:
        descriptor = SharedDataDescriptor("first value")
```

In this example, the descriptor object stores the data itself. This carries with it the inconvenience that when we modify the value for an instance all other instances of the same classes are also modified with this value as well. The following code listing puts that theory in action:

```
>>> client1 = ClientClass()
>>> client1.descriptor
'first value'
>>> client2 = ClientClass()
>>> client2.descriptor
'first value'
>>> client2.descriptor = "value for client 2"
>>> client2.descriptor
'value for client 2'
>>> client1.descriptor
'value for client 2'
```

Notice how we change one object, and suddenly all of them are from the same class, and we can see that this value is reflected. This is because `ClientClass.descriptor` is unique; it's the same object for all of them.

In some cases, this might be what we actually want (for instance, if we were to create a sort of Borg pattern implementation, on which we want to share state across all objects from a class), but in general, that is not the case, and we need to differentiate between objects.

To achieve this, the descriptor needs to know the value for each instance and return it accordingly. That is the reason we have been operating with the dictionary (`__dict__`) of each instance and setting and retrieving the values from there.

This is the most common approach. We have already covered why we cannot use `getattr()` and `setattr()` on those methods, so modifying the `__dict__` attribute is the last standing option, and, in this case, is acceptable.

3.2.2. Accessing the dictionary of the object

The way we implement descriptors is making the descriptor object store the values in the dictionary of the object, `__dict__`, and retrieve the parameters from there as well.

Note: Always store and return the data from the `__dict__` attribute of the instance.

3.2.3. Using weak references

Another alternative (if we don't want to use `__dict__`) is to make the descriptor object keep track of the values for each instance itself, in an internal mapping, and return values from this mapping as well.

There is a caveat, though. This mapping cannot just be any dictionary. Since the client class has a reference to the descriptor, and now the descriptor will keep references to the objects that use it, this will create circular dependencies, and, as a result, these objects will never be garbage-collected because they are pointing at each other.

In order to address this, the dictionary has to be a weak key one, as defined in the `weakref` module.

In this case, the code for the descriptor might look like the following:

```
from weakref import WeakKeyDictionary

class DescriptorClass:
    def __init__(self, initial_value):
        self.value = initial_value
        self.mapping = WeakKeyDictionary()

    def __get__(self, instance, owner):
        if instance is None:
            return self
        return self.mapping.get(instance, self.value)
```

(continues on next page)

(continued from previous page)

```
    def __set__(self, instance, value):
        self.mapping[instance] = value
```

This addresses the issues, but it does come with some considerations:

- The objects no longer hold their attributes: the descriptor does instead. This is somewhat controversial, and it might not be entirely accurate from a conceptual point of view. If we forget this detail, we might be asking the object by inspecting its dictionary, trying to find things that just aren't there (calling `vars(client)` will not return the complete data, for example).
- It poses the requirement over the objects that they need to be hashable. If they aren't, they can't be part of the mapping. This might be too demanding a requirement for some applications.

For these reasons, we prefer the implementation that has been shown so far, which uses the dictionary of each instance. However, for completeness, we have shown this alternative as well.

16.3.4 3.3. More considerations about descriptors

Here, we will discuss general considerations about descriptors in terms of what we can do with them when it is a good idea to use them, and also how things that we might have initially conceived as having been resolved by means of another approach can be improved through descriptors. We will then analyze the pros and cons of the original implementation versus the one after descriptors have been used.

3.3.1. Reusing code

Descriptors are a generic tool and a powerful abstraction that we can use to avoid code duplication. The best way to decide when to use descriptors is to identify cases where we would be using a property (whether for its get logic, set logic, or both), but repeating its structure many times.

Properties are just a particular case of descriptors (the @property decorator is a descriptor that implements the full descriptor protocol to define their get, set, and delete actions), which means that we can use descriptors for far more complex tasks.

Another powerful type we have seen for reusing code was decorators. Descriptors can help us create to better decorators by making sure that they will be able to work correctly for class methods as well.

When it comes to decorators, we could say that it is safe to always implement the `__get__()` method on them, and also make it a descriptor. When trying to decide whether the decorator is worth creating, consider the problems but note that there are no extra considerations toward descriptors.

As for generic descriptors, besides the aforementioned three instances rule that applies to decorators (and, in general, any reusable component), it is advisable to also keep in mind that you should use descriptors for cases when we want to define an internal API, which is some code that will have clients consuming it. This is a feature-oriented more toward designing libraries and frameworks, rather than one-time solutions.

Unless there is a very good reason to, or that the code will look significantly better, we should avoid putting business logic in a descriptor. Instead, the code of a descriptor will contain more implementational code rather than business code. It is more similar to defining a new data structure or object that another part of our business logic will use as a tool.

Note: In general, descriptors will contain implementation logic, and not so much business logic.

3.3.2. Avoiding class decorators

If we recall the class decorator we used previously to determine how an event object is going to be serialized, we ended up with an implementation that (for Python 3.7+) relied on two class decorators:

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

The first one takes the attributes from the annotations to declare the variables, whereas the second one defines how to treat each file. Let's see whether we can change these two decorators for descriptors instead.

The idea is to create a descriptor that will apply the transformation over the values of each attribute, returning the modified version according to our requirements (for example, hiding sensitive information, and formatting dates correctly):

```
from functools import partial
from typing import Callable


class BaseFieldTransformation:
    def __init__(self, transformation: Callable[[], str]) -> None:
        self._name = None
        self.transformation = transformation

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

        raw_value = instance.__dict__[self._name]
        return self.transformation(raw_value)

    def __set_name__(self, owner, name):
        self._name = name

    def __set__(self, instance, value):
        instance.__dict__[self._name] = value
        ShowOriginal = partial(BaseFieldTransformation, transformation=lambda x: x)
        HideField = partial(
            BaseFieldTransformation, transformation=lambda x: "**redacted**"
        )
        FormatTime = partial(
            BaseFieldTransformation,
            transformation=lambda ft: ft.strftime("%Y-%m-%d %H:%M"),
        )
```

This descriptor is interesting. It was created with a function that takes one argument and returns one value. This function will be the transformation we want to apply to the field. From the base definition that defines generically how it is going to work, the rest of the descriptor classes are defined, simply by changing the particular function each one needs.

The example uses `functools.partial` as a way of simulating sub-classes, by applying a partial application of the transformation function for that class, leaving a new callable that can be instantiated directly.

In order to keep the example simple, we will implement the `__init__()` and `serialize()` methods, although they could be abstracted away as well. Under these considerations, the class for the event will now be defined as follows:

```
class LoginEvent:

    username = ShowOriginal()
    password = HideField()
    ip = ShowOriginal()
    timestamp = FormatTime()

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

    def serialize(self):
        return {
            "username": self.username,
            "password": self.password,
            "ip": self.ip,
            "timestamp": self.timestamp,
        }
```

We can see how the object behaves at runtime:

```
>>> le = LoginEvent("john", "secret password", "1.1.1.1",
datetime.utcnow())
>>> vars(le)
{'username': 'john', 'password': 'secret password', 'ip': '1.1.1.1',
'timestamp': ...}
>>> le.serialize()
{'username': 'john', 'password': '**redacted**', 'ip': '1.1.1.1',
'timestamp': '...'}
>>> le.password
'**redacted**'
```

There are some differences with respect to the previous implementation that used a decorator. This example added the `serialize()` method and hid the fields before presenting them to its resulting dictionary, but if we asked for any of these attributes to an instance of the event in memory at any point, it would still give us the original value, without any transformation applied to it (we could have chosen to apply the transformation when setting the value, and return it directly on the `__get__()`, as well).

Depending on the sensitivity of the application, this may or may not be acceptable, but in this case, when we ask the object for its public attributes, the descriptor will apply the transformation before presenting the results. It is still possible to access the original values by asking for the dictionary of the object (by accessing `__dict__`), but when we ask for the value, by default, it will return it converted.

In this example, all descriptors follow a common logic, which is defined in the base class. The descriptor should store the value in the object and then ask for it, applying the transformation it defines. We could create a hierarchy of classes, each one defining its own conversion function, in a way that the template method design pattern works. In this case, since the changes in the derived classes are relatively small (just one function), we opted for creating the derived classes as partial applications of the base class. Creating any new transformation field should be as simple as defining a new class that will be the base class, which is partially applied with the function we need. This can even be done ad hoc, so there might be no need to set a name for it.

Regardless of this implementation, the point is that since descriptors are objects, we can create models, and apply all rules of object-oriented programming to them. Design patterns also apply to descriptors. We could define our hierarchy, set the custom behavior, and so on. This example follows the OCP, because adding a new type of conversion method would just be about creating a new class, derived from the base one with the function it needs, without having to modify the base class itself (to be fair, the previous implementation with decorators was also OCP- compliant, but there were no classes involved for each transformation mechanism).

Let's take an example where we create a base class that implements the ``__init__()``and ``serialize()`` methods so that we can define the `LoginEvent` class simply by deriving from it, as follows:

```
class LoginEvent(BaseEvent):
    username = ShowOriginal()
    password = HideField()
    ip = ShowOriginal()
    timestamp = FormatTime()
```

Once we achieve this code, the class looks cleaner. It only defines the attributes it needs, and its logic can be quickly analyzed by looking at the class for each attribute. The base class will abstract only the common methods, and the class of each event will look simpler and more compact.

Not only do the classes for each event look simple, but the descriptor itself is very compact and a lot simpler than the class decorators. The original implementation with class decorators was good, but descriptors made it even better.

16.4 4. Analysis of descriptors

We have seen how descriptors work so far and explored some interesting situations in which they contribute to clean design by simplifying their logic and leveraging more compact classes.

Up to this point, we know that by using descriptors, we can achieve cleaner code, abstracting away repeated logic and implementation details. But how do we know our implementation of the descriptors is clean and correct? What makes a good descriptor? Are we using this tool properly or over-engineering with it?

16.4.1 4.1. How Python uses descriptors internally

Referring to the question as to what makes a good descriptor?, a simple answer would be that a good descriptor is pretty much like any other good Python object. It is consistent with Python itself. The idea that follows this premise is that analyzing how Python uses descriptors will give us a good idea of good implementations so that we know what to expect from the descriptors we write.

We will see the most common scenarios where Python itself uses descriptors to solve parts of its internal logic, and we will also discover elegant descriptors and that they have been there in plain sight all along.

4.1.1. Functions and methods

The most resonating case of an object that is a descriptor is probably a function. Functions implement the `__get__` method, so they can work as methods when defined inside a class. Methods are just functions that take an extra argument. By convention, the first argument of a method is named "self", and it represents an instance of the class that the method is being defined in. Then, whatever the method does with "self", would be the same as any other function receiving the object and applying modifications to it.

In order words, when we define something like this:

```
class MyClass:
    def method(self, ...):
        self.x = 1
```

It is actually the same as if we define this:

```
class MyClass:
    pass

def method(myclass_instance, ...):
```

(continues on next page)

(continued from previous page)

```
    myclass_instance.x = 1
    method(MyClass())
```

So, it is just another function, modifying the object, only that it's defined inside the class, and it is said to be bound to the object.

When we call something in the form of this:

```
instance = MyClass()
instance.method(...)
```

Python is, in fact, doing something equivalent to this:

```
instance = MyClass()
MyClass.method(instance, ...)
```

Notice that this is just a syntax conversion that is handled internally by Python. The way this works is by means of descriptors.

Since functions implement the descriptor protocol (see the following listing) before calling the method, the `__get__()` method is invoked first, and some transformations happen before running the code on the internal callable:

```
>>> def function(): pass
...
>>> function.__get__
<method-wrapper '__get__' of function object at 0x...>
```

In the `instance.method(...)` statement, before processing all the arguments of the callable inside the parenthesis, the "instance.method" part is evaluated.

Since `method` is an object defined as a class attribute, and it has a `__get__` method, this is called. What this does is convert the function to a method, which means binding the callable to the instance of the object it is going to work with.

Let's see this with an example so that we can get an idea of what Python might be doing internally.

We will define a callable object inside a class that will act as a sort of function or method that we want to define to be invoked externally. An instance of the `Method` class is supposed to be a function or method to be used inside a different class. This function will just print its three parameters: the instance that it received (which would be the self parameter on the class it's being defined in), and two more arguments. Notice that in the `__call__()` method, the self parameter does not represent the instance of `MyClass`, but instead an instance of `Method`. The parameter named instance is meant to be a `MyClass` type of object:

```
class Method:
    def __init__(self, name):
        self.name = name

    def __call__(self, instance, arg1, arg2):
        print(f"{self.name}: {instance} called with {arg1} and {arg2}")

class MyClass:
    method = Method("Internal call")
```

Under these considerations and, after creating the object, the following two calls should be equivalent, based on the preceding definition:

```
instance = MyClass()
Method("External call")(instance, "first", "second")
instance.method("first", "second")
```

However, only the first one works as expected, as the second one gives an error:

```
Traceback (most recent call last):
File "file", line, in <module>
instance.method("first", "second")
TypeError: __call__() missing 1 required positional argument: 'arg2'
```

We are seeing the same error we faced with a decorator. The arguments are being shifted to the left by one, instance is taking the place of `self`, `arg1` is going to be instance, and there is nothing to provide for `arg2`.

In order to fix this, we need to make `Method` a descriptor.

This way, when we call `instance.method` first, we are going to call its `__get__()`, on which we bind this callable to the object accordingly (bypassing the object as the first parameter), and then proceed:

```
from types import MethodType

class Method:
    def __init__(self, name):
        self.name = name

    def __call__(self, instance, arg1, arg2):
        print(f"{self.name}: {instance} called with {arg1} and {arg2}")

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

        return MethodType(self, instance)
```

Now, both calls work as expected:

```
External call: <MyClass object at 0x...> called with fist and second
Internal call: <MyClass object at 0x...> called with first and second
```

What we did is convert the function (actually the callable object we defined instead) to a method by using `MethodType` from the `types` module. The first parameter of this class should be a callable (`self`, in this case, is one by definition because it implements `__call__`), and the second one is the object to bind this function to.

Something similar to this is what function objects use in Python so they can work as methods when they are defined inside a class.

Since this is a very elegant solution, it's worth exploring it to keep it in mind as a Pythonic approach when defining our own objects. For instance, if we were to define our own callable, it would be a good idea to also make it a descriptor so that we can use it in classes as class attributes as well.

4.1.2. Built-in decorators for methods

All `@property`, `@classmethod`, and `@staticmethod` decorators are descriptors.

We have mentioned several times that the idiom makes the descriptor return itself when it's being called from a class directly. Since properties are actually descriptors, that is the reason why, when we ask it from the class, we don't get the result of computing the property, but the entire property object instead:

```
>>> class MyClass:
...     @property
...     def prop(self): pass
...
>>> MyClass.prop
<property object at 0x...>
```

For class methods, the `__get__` function in the descriptor will make sure that the class is the first parameter to be passed to the function being decorated, regardless of whether it's called from the class directly or from an instance.

For static methods, it will make sure that no parameters are bound other than those defined by the function, namely undoing the binding done by `__get__()` on functions that make self the first parameter of that function.

Let's take an example; we create a `@classproperty` decorator that works as the regular `@property` decorator, but for classes instead. With a decorator like this one, the following code should be able to work:

```
class TableEvent:
    schema = "public"
    table = "user"

    @classproperty
    def topic(cls):
        prefix = read_prefix_from_config()
        return f"{prefix}{cls.schema}.{cls.table}"


>>> TableEvent.topic
'public.user'
>>> TableEvent().topic
'public.user'
```

16.4.2 4.2. Implementing descriptors in decorators

We now understand how Python uses descriptors in functions to make them work as methods when they are defined inside a class. We have also seen examples of cases where we can make decorators work by making them comply with the descriptor protocol by using the `__get__()` method of the interface to adapt the decorator to the object it is being called with. This solves the problem for our decorators in the same way that Python solves the issue of functions as methods in objects.

The general recipe for adapting a decorator in such a way is to implement the `__get__()` method on it and use `types.MethodType` to convert the callable (the decorator itself) to a method bound to the object it is receiving (the instance parameter received by `__get__`).

For this to work, we will have to implement the decorator as an object, because otherwise, if we are using a function, it will already have a `__get__()` method, which will be doing something different that will not work unless we adapt it. The cleaner way to proceed is to define a class for the decorator.

Note: Use a decorator class when defining a decorator that we want to apply to class methods, and implement the `__get__()` method on it.

CHAPTER

SEVENTEEN

GENERATORS

Generators provide an elegant way to write simple and efficient code for functions that return a sequence of elements. Based on the `yield` statement, they allow you to pause a function and return an intermediate result. The function saves its execution context and can be resumed later, if necessary.

17.1 1. Creating generators

Generators were introduced in Python a long time ago, with the idea of introducing iteration in Python while improving the performance of the program (by using less memory) at the same time.

The idea of a generator is to create an object that is iterable, and, while it's being iterated, will produce the elements it contains, one at a time. The main use of generators is to save memory: instead of having a very large list of elements in memory, holding everything at once, we have an object that knows how to produce each particular element, one at a time, as they are required.

In many cases, the resources required to process one element are less than the resources required to store whole sequences. Therefore, they can be kept low, making the program more efficient.

This feature enables lazy computations or heavyweight objects in memory, in a similar manner to what other functional programming languages (Haskell, for instance) provide. It would even be possible to work with infinite sequences because the lazy nature of generators allows for such an option.

A common use case is to stream data buffers with generators (for example, from files). They can be paused, resumed, and stopped whenever necessary at any stage of the data processing pipeline without any need to load whole datasets into the program's memory

17.1.1 1.1. A first look at generators

Let's start with an example. The problem at hand now is that we want to process a large list of records and get some metrics and indicators over them. Given a large data set with information about purchases, we want to process it in order to get the lowest sale, highest sale, and the average price of a sale.

For the simplicity of this example, we will assume a CSV with only two fields, in the following format:

```
<purchase_date>, <price>
...
```

We are going to create an object that receives all the purchases, and this will give us the necessary metrics. We could get some of these values out of the box by simply using the `min()` and `max()` built-in functions, but that would require iterating all of the purchases more than once, so instead, we are using our custom object, which will get these values in a single iteration.

The code that will get the numbers for us looks rather simple. It's just an object with a method that will process all prices in one go, and, at each step, will update the value of each particular metric we are interested in. First, we will show the first implementation in the following listing, and, later on (once we have seen more about iteration), we will revisit this implementation and get a much better (and compact) version of it. For now, we are settling on the following:

```
class PurchasesStats:
    def __init__(self, purchases):
        self.purchases = iter(purchases)
        self.min_price: float = None
        self.max_price: float = None
        self._total_purchases_price: float = 0.0
        self._total_purchases = 0
        self._initialize()

    def _initialize(self):
        try:
            first_value = next(self.purchases)
        except StopIteration:
            raise ValueError("no values provided")

        self.min_price = self.max_price = first_value
        self._update_avg(first_value)

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

        return self

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

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

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

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

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

This object will receive all the totals for the `purchases` and process the required values. Now, we need a function that loads these numbers into something that this object can process. Here is the first version:

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

    return purchases
```

This code works; it loads all the numbers of the file into a list that, when passed to our custom object, will produce the numbers we want. It has a performance issue, though. If you run it with a rather large dataset, it will take a

while to complete, and it might even fail if the dataset is large enough as to not fit into the main memory.

If we take a look at our code that consumes this data, it is processing the `purchases`, one at a time, so we might be wondering why our producer fits everything in memory at once. It is creating a list where it puts all of the content of the file, but we know we can do better.

The solution is to create a generator. Instead of loading the entire content of the file in a list, we will produce the results one at a time. The code will now look like this:

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

If you measure the process this time, you will notice that the usage of memory has dropped significantly. We can also see how the code looks simpler: there is no need to define the list (therefore, there is no need to append to it), and that the return statement also disappeared.

In this case, the `load_purchases` function is a generator function, or simply a generator.

In Python, the mere presence of the keyword `yield` in any function makes it a generator, and, as a result, when calling it, nothing other than creating an instance of the generator will happen:

```
>>> load_purchases("file")
<generator object load_purchases at 0x...>
```

A generator object is an iterable (we will revisit iterables in more detail later on), which means that it can work with for loops. Notice how we did not have to change anything on the consumer code: our statistics processor remained the same, with the for loop unmodified, after the new implementation.

Working with iterables allows us to create these kinds of powerful abstractions that are polymorphic with respect to for loops. As long as we keep the iterable interface, we can iterate over that object transparently.

17.1.2 1.2. Generator expressions

Generators save a lot of memory, and since they are iterators, they are a convenient alternative to other iterables or containers that require more space in memory such as lists, tuples, or sets.

Much like these data structures, they can also be defined by comprehension, only that it is called a generator expression (there is an ongoing argument about whether they should be called generator comprehensions).

In the same way, we would define a list comprehension. If we replace the square brackets with parenthesis, we get a generator that results from the expression. Generator expressions can also be passed directly to functions that work with iterables, such as `sum()`, and, `max()`:

```
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> (x**2 for x in range(10))
<generator object <genexpr> at 0x...>
>>> sum(x**2 for x in range(10))
285
```

Note: Always pass a generator expression, instead of a list comprehension, to functions that expect iterables, such as `min()`, `max()`, and `sum()`. This is more efficient and pythonic.

It is also worth mentioning, that we can only iterate 1 time over generators:

```
>>> a = (x for x in range(3))
>>> a
<generator object <genexpr> at 0x7f95ece4dad0>
```

(continues on next page)

(continued from previous page)

```
>>> for x in a:
...     print(x)
...
0
1
2

>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Tip: It is better to have a lot of simple iterable functions that work over sequences of values than a complex function that computes the result for one value at a time.

17.2 2. Iterating idiomatically

In this section, we will first explore some idioms that come in handy when we have to deal with iteration in Python. These code recipes will help us get a better idea of the types of things we can do with generators (especially after we have already seen generator expressions), and how to solve typical problems in relation to them.

Once we have seen some idioms, we will move on to exploring iteration in Python in more depth, analyzing the methods that make iteration possible, and how iterable objects work.

17.2.1 2.1. Idioms for iteration

We are already familiar with the built-in `enumerate()` function that, given an iterable, will return another one on which the element is a tuple, whose first element is the enumeration of the second one (corresponding to the element in the original iterable):

```
>>> list(enumerate("abcdef"))
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e'), (5, 'f')]
```

We wish to create a similar object, but in a more low-level fashion; one that can simply create an infinite sequence. We want an object that can produce a sequence of numbers, from a starting one, without any limits.

An object as simple as the following one can do the trick. Every time we call this object, we get the next number of the sequence *ad infinitum*:

```
class NumberSequence:
    def __init__(self, start=0):
        self.current = start

    def next(self):
        current = self.current
        self.current += 1
        return current
```

Based on this interface, we would have to use this object by explicitly invoking its next() method:

```
>>> seq = NumberSequence()
>>> seq.next()
0
>>> seq.next()
1
```

(continues on next page)

(continued from previous page)

```
>>> seq2 = NumberSequence(10)
>>> seq2.next()
10
>>> seq2.next()
11
```

But with this code, we cannot reconstruct the `enumerate()` function as we would like to, because its interface does not support being iterated over a regular Python for loop, which also means that we cannot pass it as a parameter to functions that expect something to iterate over. Notice how the following code fails:

```
>>> list(zip(NumberSequence(), "abcdef"))
Traceback (most recent call last):
File "...", line 1, in <module>
TypeError: zip argument #1 must support iteration
```

The problem lies in the fact that `NumberSequence` does not support iteration. To fix this, we have to make the object an iterable by implementing the magic method `__iter__()`. We have also changed the previous `next()` method, by using the magic method `__next__`, which makes the object an iterator:

```
class SequenceOfNumbers:
    def __init__(self, start=0):
        self.current = start

    def __next__(self):
        current = self.current
        self.current += 1
        return current

    def __iter__(self):
        return self
```

This has an advantage: not only can we iterate over the element, we also don't even need the `next()` method any more because having `__next__()` allows us to use the `next()` built-in function:

```
>>> list(zip(SequenceOfNumbers(), "abcdef"))
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e'), (5, 'f')]
>>> seq = SequenceOfNumbers(100)
>>> next(seq)
100
>>> next(seq)
101
```

2.1.1. The next() function

The `next()` built-in function will advance the iterable to its next element and return it:

```
>>> word = iter("hello")
>>> next(word)
'h'
>>> next(word)
'e'
```

If the iterator does not have more elements to produce, the `StopIteration` exception is raised:

```
>>> ...
>>> next(word)
'o'
>>> next(word)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

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

This exception signals that the iteration is over and that there are no more elements to consume.

If we wish to handle this case, besides catching the `StopIteration` exception, we could provide this function with a default value in its second parameter. Should this be provided, it will be the return value in lieu of throwing `StopIteration`:

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

2.1.2. Using a generator

The previous code can be simplified significantly by simply using a generator. Generator objects are iterators. This way, instead of creating a class, we can define a function that `yield` the values as needed:

```
def sequence(start=0):
    while True:
        yield start
        start += 1
```

Remember that from our first definition, the `yield` keyword in the body of the function makes it a generator. Because it is a generator, it's perfectly fine to create an infinite loop like this, because, when this generator function is called, it will run all the code until the next `yield` statement is reached. It will produce its value and suspend there:

```
>>> seq = sequence(10)
>>> next(seq)
10
>>> next(seq)
11
>>> list(zip(sequence(), "abcdef"))
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e'), (5, 'f')]
```

2.1.3. Itertools

Working with iterables has the advantage that the code blends better with Python itself because iteration is a key component of the language. Besides that, we can take full advantage of the itertools module. Actually, the `sequence()` generator we just created is fairly similar to `itertools.count()`. However, there is more we can do.

One of the nicest things about iterators, generators, and itertools, is that they are composable objects that can be chained together.

For instance, getting back to our first example that processed purchases in order to get some metrics, what if we want to do the same, but only for those values over a certain threshold? The naive approach of solving this problem would be to place the condition while iterating:

```
def process(self):
    for purchase in self.purchases:
        if purchase > 1000.0:
        ...
```

This is not only non-Pythonic, but it's also rigid (and rigidity is a trait that denotes bad code). It doesn't handle changes very well. What if the number changes now? Do we pass it by parameter? What if we need more than one? What if the condition is different (less than, for instance)? Do we pass a lambda?

These questions should not be answered by this object, whose sole responsibility is to compute a set of well-defined metrics over a stream of purchases represented as numbers. And, of course, the answer is no. It would be a huge mistake to make such a change (once again, clean code is flexible, and we don't want to make it rigid by coupling this object to external factors). These requirements will have to be addressed elsewhere.

It's better to keep this object independent of its clients. The less responsibility this class has, the more useful it will be for more clients, hence enhancing its chances of being reused.

Instead of changing this code, we're going to keep it as it is and assume that the new data is filtered according to whatever requirements each customer of the class has.

For instance, if we wanted to process only the first 10 purchases that amount to more than 1,000, we would do the following:

```
>>> from itertools import islice
>>> purchases = islice(filter(lambda p: p > 1000.0, purchases), 10)
>>> stats = PurchasesStats(purchases).process()
```

There is no memory penalization for filtering this way because since they all are generators, the evaluation is always lazy. This gives us the power of thinking as if we had filtered the entire set at once and then passed it to the object, but without actually fitting everything in memory.

2.1.4. Simplifying code through iterators

Now, we will briefly discuss some situations that can be improved with the help of iterators, and occasionally the itertools module. After discussing each case, and its proposed optimization, we will close each point with a corollary.

2.1.4.1. Repeated iterations

Now that we have seen more about iterators, and introduced the itertools module, we can show you how one of the first examples of this chapter (the one for computing statistics about some purchases), can be dramatically simplified:

```
def process_purchases(purchases):
    min_iter, max_iter, avg_iter = itertools.tee(purchases, 3)
    return min(min_iter), max(max_iter), median(avg_iter)
```

In this example, `itertools.tee` will split the original iterable into three new ones. We will use each of these for the different kinds of iterations that we require, without needing to repeat three different loops over purchases.

The reader can simply verify that if we pass an iterable object as the purchases parameter, this one is traversed only once (thanks to the itertools.tee function), which was our main requirement. It is also possible to verify how this version is equivalent to our original implementation. In this case, there is no need to manually raise `ValueError` because passing an empty sequence to the `min()` function will do the same.

Note: If you are thinking about running a loop over the same object more than one time, stop and think if itertools.tee can be of any help.

2.1.4.2. Nested loops

In some situations, we need to iterate over more than one dimension, looking for a value, and nested loops come as the first idea. When the value is found, we need to stop iterating, but the `break` keyword doesn't work entirely because we have to escape from two (or more) for loops, not just one.

What would be the solution for this? A flag signaling escape? No. Raising an exception? No, this would be the same as the flag, but even worse because we know that exceptions are not to be used for control flow logic. Moving the code to a smaller function and return it? Close, but not quite.

The answer is, whenever possible, flat the iteration to a single for loop. This is the kind of code we would like to avoid:

```
def search_nested_bad(array, desired_value):
    coords = None
    for i, row in enumerate(array):
        for j, cell in enumerate(row):
            if cell == desired_value:
                coords = (i, j)
                break
        if coords is not None:
            break

    if coords is None:
        raise ValueError(f"{desired_value} not found")

    logger.info("value %r found at [%i, %i]", desired_value, *coords)
    return coords
```

And here is a simplified version of it that does not rely on flags to signal termination, and has a simpler, more compact structure of iteration:

```
def _iterate_array2d(array2d):
    for i, row in enumerate(array2d):
        for j, cell in enumerate(row):
            yield (i, j), cell

def search_nested(array, desired_value):
    try:
        coord = next(coord for (coord, cell) in _iterate_array2d(array) if cell ==␣
↪desired_value)
    except StopIteration:
        raise ValueError(f"{desired_value} not found")

    logger.info(f"value {desired_value} found at {coords}")
    return coord
```

It's worth mentioning how the auxiliary generator that was created works as an abstraction for the iteration that's required. In this case, we just need to iterate over two dimensions, but if we needed more, a different object could handle this without the client needing to know about it. This is the essence of the iterator design pattern, which, in Python, is transparent, since it supports iterator objects automatically, which is the topic covered in the next section.

Note: Try to simplify the iteration as much as possible with as many abstractions as are required, flatting the loops whenever possible.

17.2.2 2.2. The iterator pattern in Python

Here, we will take a small detour from generators to understand iteration in Python more deeply. Generators are a particular case of iterable objects, but iteration in Python goes beyond generators, and being able to create good iterable objects will give us the chance to create more efficient, compact, and readable code.

In the previous code listings, we have been seeing examples of iterable objects that are also iterators, because they implement both the `__iter__()` and `__next__()` magic methods. While this is fine in general, it's not strictly required that they always have to implement both methods, and here we'll show the subtle differences between an iterable object (one that implements `__iter__`) and an iterator (that implements `__next__`).

We also explore other topics related to iterations, such as sequences and container objects.

2.2.1. The interface for iteration

An iterable is an object that supports iteration, which, at a very high level, means that we can run a `for .. in ...` loop over it, and it will work without any issues. However, iterable does not mean the same as iterator.

Generally speaking, an iterable is just something we can iterate, and it uses an iterator to do so. This means that in the `__iter__` magic method, we would like to return an iterator, namely, an object with a `__next__()` method implemented.

An iterator is an object that only knows how to produce a series of values, one at a time, when it's being called by the already explored built-in `next()` function. While the iterator is not called, it's simply frozen, sitting idly by until it's called again for the next value to produce. In this sense, generators are iterators.

In the following code, we will see an example of an iterator object that is not iterable: it only supports invoking its values, one at a time. Here, the name sequence refers just to a series of consecutive numbers, not to the sequence concept in Python, which will we explore later on:

```
class SequenceIterator:
    def __init__(self, start=0, step=1):
        self.current = start
        self.step = step

    def __next__(self):
        value = self.current
        self.current += self.step
        return value
```

Notice that we can get the values of the sequence one at a time, but we can't iterate over this object (this is fortunate because it would otherwise result in an endless loop):

```
>>> si = SequenceIterator(1, 2)
>>> next(si)
1
>>> next(si)
3
>>> next(si)
5
>>> for _ in SequenceIterator(): pass
...
Traceback (most recent call last):
...
TypeError: 'SequenceIterator' object is not iterable
```

The error message is clear, as the object doesn't implement `__iter__()`.

Just for explanatory purposes, we can separate the iteration in another object (again, it would be enough to make the object implement both `__iter__` and `__next__`, but doing so separately will help clarify the distinctive point we're trying to make in this explanation).

2.2.2. Sequence objects as iterables

As we have just seen, if an object implements the `__iter__()` magic method, it means it can be used in a for loop. While this is a great feature, it's not the only possible form of iteration we can achieve. When we write a for loop, Python will try to see if the object we're using implements `__iter__`, and, if it does, it will use that to construct the iteration, but if it doesn't, there are fallback options.

If the object happens to be a sequence (meaning that it implements `__getitem__()` and `__len__()` magic methods), it can also be iterated. If that is the case, the interpreter will then provide values in sequence, until the `IndexError` exception is raised, which, analogous to the aforementioned `StopIteration`, also signals the stop for the iteration.

With the sole purpose of illustrating such a behavior, we run the following experiment that shows a sequence object that implements `map()` over a range of numbers:

```
class MappedRange:
    """Apply a transformation to a range of numbers."""
    def __init__(self, transformation, start, end):
        self._transformation = transformation
        self._wrapped = range(start, end)

    def __getitem__(self, index):
        value = self._wrapped.__getitem__(index)
        result = self._transformation(value)
        logger.info(f"Index {index}: {result}")
        return result

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

Keep in mind that this example is only designed to illustrate that an object such as this one can be iterated with a regular for loop. There is a logging line placed in the `__getitem__` method to explore what values are passed while the object is being iterated, as we can see from the following test:

```
>>> mr = MappedRange(abs, -10, 5)
>>> mr[0]
Index 0: 10
10
>>> mr[-1]
Index -1: 4
4
>>> list(mr)
Index 0: 10
Index 1: 9
Index 2: 8
Index 3: 7
Index 4: 6
Index 5: 5
Index 6: 4
Index 7: 3
Index 8: 2
Index 9: 1
Index 10: 0
Index 11: 1
Index 12: 2
Index 13: 3
Index 14: 4
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 1, 2, 3, 4]
```

As a word of caution, it's important to highlight that while it is useful to know this, it's also a fallback mechanism for when the object doesn't implement `__iter__`, so most of the time we'll want to resort to these methods by thinking in creating proper sequences, and not just objects we want to iterate over.

Note: When thinking about designing an object for iteration, favor a proper iterable object (with `__iter__`), rather than a sequence that can coincidentally also be iterated.

17.3 3. Coroutines

As we already know, generator objects are iterables. They implement `__iter__()` and `__next__()`. This is provided by Python automatically so that when we create a generator object function, we get an object that can be iterated or advanced through the `next()` function.

Besides this basic functionality, they have more methods so that they can work as coroutines. Here, we will explore how generators evolved into coroutines to support the basis of asynchronous programming before we go into more detail in the next section, where we explore the new features of Python and the syntax that covers programming asynchronously. The basic methods added to support coroutines are as follows:

- `.close()`
- `.throw(ex_type[, ex_value[, ex_traceback]])`
- `.send(value)`

17.3.1 3.1. The methods of the generator interface

In this section, we will explore what each of the aforementioned methods does, how it works, and how it is expected to be used. By understanding how to use these methods, we will be able to make use of simple coroutines.

Later on, we will explore more advanced uses of coroutines, and how to delegate to sub- generators (coroutines) in order to refactor code, and how to orchestrate different coroutines.

3.1.1. close()

When calling this method, the generator will receive the `GeneratorExit` exception. If it's not handled, then the generator will finish without producing any more values, and its iteration will stop.

This exception can be used to handle a finishing status. In general, if our coroutine does some sort of resource management, we want to catch this exception and use that control block to release all resources being held by the coroutine. In general, it is similar to using a context manager or placing the code in the finally block of an exception control, but handling this exception specifically makes it more explicit.

In the following example, we have a coroutine that makes use of a database handler object that holds a connection to a database, and runs queries over it, streaming data by pages of a fixed length (instead of reading everything that is available at once):

```
def stream_db_records(db_handler):
    try:
        while True:
            yield db_handler.read_n_records(10)
    except GeneratorExit:
        db_handler.close()
```

At each call to the generator, it will return 10 rows obtained from the database handler, but when we decide to explicitly finish the iteration and call `close()`, we also want to close the connection to the database:

```
>>> streamer = stream_db_records(DBHandler("testdb"))
>>> next(streamer)
[(0, 'row 0'), (1, 'row 1'), (2, 'row 2'), (3, 'row 3'), ...]
>>> next(streamer)
[(0, 'row 0'), (1, 'row 1'), (2, 'row 2'), (3, 'row 3'), ...]
```

(continues on next page)

(continued from previous page)

```
>>> streamer.close()
INFO:...:closing connection to database 'testdb'
```

Use the `close()` method on generators to perform finishing-up tasks when needed.

3.1.2. throw(ex_type[, ex_value[, ex_traceback]])

This method will throw the exception at the line where the generator is currently suspended. If the generator handles the exception that was sent, the code in that particular except clause will be called, otherwise, the exception will propagate to the caller.

Here, we are modifying the previous example slightly to show the difference when we use this method for an exception that is handled by the coroutine, and when it's not:

```
class CustomException(Exception):
    pass

def stream_data(db_handler):
    while True:
        try:
            yield db_handler.read_n_records(10)
        except CustomException as e:
            logger.info(f"controlled error {e}, continuing")
        except Exception as e:
            logger.info(f"unhandled error {e}, stopping")
            db_handler.close()
        break
```

Now, it is a part of the control flow to receive a `CustomException`, and, in such a case, the generator will log an informative message (of course, we can adapt this according to our business logic on each case), and move on to the next `yield` statement, which is the line where the coroutine reads from the database and returns that data.

This particular example handles all exceptions, but if the last block (`except Exception:`) wasn't there, the result would be that the generator is raised at the line where the generator is paused (again, the `yield`), and it will propagate from there to the caller:

```
>>> streamer = stream_data(DBHandler("testdb"))
>>> next(streamer)
[(0, 'row 0'), (1, 'row 1'), (2, 'row 2'), (3, 'row 3'), (4, 'row 4'), ...]
>>> next(streamer)
[(0, 'row 0'), (1, 'row 1'), (2, 'row 2'), (3, 'row 3'), (4, 'row 4'), ...]
>>> streamer.throw(CustomException)
WARNING:controlled error CustomException(), continuing
[(0, 'row 0'), (1, 'row 1'), (2, 'row 2'), (3, 'row 3'), (4, 'row 4'), ...]
>>> streamer.throw(RuntimeError)
ERROR:unhandled error RuntimeError(), stopping
INFO:closing connection to database 'testdb'
Traceback (most recent call last):
...
StopIteration
```

When our exception from the domain was received, the generator continued. However, when it received another exception that was not expected, the default block caught where we closed the connection to the database and finished the iteration, which resulted in the generator being stopped. As we can see from the `StopIteration` that was raised, this generator can't be iterated further.

3.1.3. send(value)

In the previous example, we created a simple generator that reads rows from a database, and when we wished to finish its iteration, this generator released the resources linked to the database. This is a good example of using one of the methods that generators provide (`close`), but there is more we can do.

An obvious of such a generator is that it was reading a fixed number of rows from the database.

We would like to parametrize that number so that we can change it throughout different calls. Unfortunately, the `next()` function does not provide us with options for that. But luckily, we have `send()`:

```
def stream_db_records(db_handler):
    retrieved_data = None
    previous_page_size = 10

    try:
        while True:
            page_size = yield retrieved_data
            if page_size is None:
                page_size = previous_page_size

            previous_page_size = page_size
            retrieved_data = db_handler.read_n_records(page_size)

    except GeneratorExit:
        db_handler.close()
```

The idea is that we have now made the coroutine able to receive values from the caller by means of the `send()` method. This method is the one that actually distinguishes a generator from a coroutine because when it's used, it means that the `yield` keyword will appear on the right-hand side of the statement, and its return value will be assigned to something else.

In coroutines, we generally find the `yield` keyword to be used in the following form: `receive = yield produced`

The `yield`, in this case, will do two things. It will send `produced` back to the caller, which will pick it up on the next round of iteration (after calling `next()`, for example), and it will suspend there. At a later point, the caller will want to send a value back to the coroutine by using the `send()` method. This value will become the result of the `yield` statement, assigned in this case to the variable named `receive`.

Sending values to the coroutine only works when this one is suspended at a `yield` statement, waiting for something to produce. For this to happen, the coroutine will have to be advanced to that status. The only way to do this is by calling `next()` on it. This means that before sending anything to the coroutine, this has to be advanced at least once via the `next()` method. Failure to do so will result in an exception:

```
>>> c = coro()
>>> c.send(1)
Traceback (most recent call last):
...
TypeError: can't send non-None value to a just-started generator
```

Important: Always remember to advance a coroutine by calling `next()` before sending any values to it.

Back to our example. We are changing the way elements are produced or streamed to make it able to receive the length of the records it expects to read from the database.

The first time we call `next()`, the generator will advance up to the line containing `yield`; it will provide a value to the caller (`None`, as set in the variable), and it will suspend there).

From here, we have two options. If we choose to advance the generator by calling `next()`, the default value of 10 will be used, and it will go on with this as usual. This is because `next()` is technically the same as `send(None)`, but this is covered in the if statement that will handle the value that we previously set.

If, on the other hand, we decide to provide an explicit value via `send(<value>)`, this one will become the result of the `yield` statement, which will be assigned to the variable containing the length of the page to use, which, in turn, will be used to read from the database.

Successive calls will have this logic, but the important point is that now we can dynamically change the length of the data to read in the middle of the iteration, at any point.

Now that we understand how the previous code works, most Pythonistas would expect a simplified version of it (after all, Python is also about brevity and clean and compact code):

```
def stream_db_records(db_handler):
    retrieved_data = None
    page_size = 10
    try:
        while True:
            page_size = (yield retrieved_data) or page_size
            retrieved_data = db_handler.read_n_records(page_size)
    except GeneratorExit:
        db_handler.close()
```

This version is not only more compact, but it also illustrates the idea better. The parenthesis around the `yield` makes it clearer that it's a statement (think of it as if it were a function call), and that we are using the result of it to compare it against the previous value.

This works as we expect it does, but we always have to remember to advance the coroutine before sending any data to it. If we forget to call the first `next()`, we'll get a `TypeError`. This call could be ignored for our purposes because it doesn't return anything we'll use.

It would be good if we could use the coroutine directly, right after it is created without having to remember to call `next()` the first time, every time we are going to use it. Some authors devised an interesting decorator to achieve this. The idea of this decorator is to advance the coroutine, so the following definition works automatically:

```
@prepare_coroutine
def stream_db_records(db_handler):
    retrieved_data = None
    page_size = 10
    try:
        while True:
            page_size = (yield retrieved_data) or page_size
            retrieved_data = db_handler.read_n_records(page_size)
    except GeneratorExit:
    db_handler.close()

>>> streamer = stream_db_records(DBHandler("testdb"))
>>> len(streamer.send(5))
5
```

17.3.2 3.2. More advanced coroutines

So far, we have a better understanding of coroutines, and we are able to create simple ones to handle small tasks. We can say that these coroutines are, in fact, just more advanced generators (and that would be right, coroutines are just fancy generators), but, if we actually want to start supporting more complex scenarios, we usually have to go for a design that handles many coroutines concurrently, and that requires more features.

When handling many coroutines, we find new problems. As the control flow of our application becomes more complex, we want to pass values up and down the stack (as well as exceptions), be able to capture values from sub-coroutines we might call at any level, and finally schedule multiple coroutines to run toward a common goal.

To make things simpler, generators had to be extended once again. This is addressed by changing the semantic of generators so that they are able to return values, and introducing the new yield from construction.

3.2.1. Returning values in coroutines

As introduced at the beginning, the iteration is a mechanism that calls `next()` on an iterable object many times until a `StopIteration` exception is raised.

So far, we have been exploring the iterative nature of generators: we produce values one at a time, and, in general, we only care about each value as it's being produced at every step of the `for` loop. This is a very logical way of thinking about generators, but coroutines have a different idea; even though they are technically generators, they weren't conceived with the idea of iteration in mind, but with the goal of suspending the execution of a code until it's resumed later on.

This is an interesting challenge; when we design a coroutine, we usually care more about suspending the state rather than iterating (and iterating a coroutine would be an odd case). The challenge lies in that it is easy to mix them both. This is because of a technical implementation detail; the support for coroutines in Python was built upon generators.

If we want to use coroutines to process some information and suspend its execution, it would make sense to think of them as lightweight threads (or green threads, as they are called in other platforms). In such a case, it would make sense if they could return values, much like calling any other regular function.

But let's remember that generators are not regular functions, so in a generator, the construction `value = generator()` will do nothing other than create a generator object. What would be the semantics for making a generator return a value? It will have to be after the iteration is done.

When a generator returns a value, its iteration is immediately stopped (it can't be iterated any further). To preserve the semantics, the `StopIteration` exception is still raised, and the value to be returned is stored inside the exception object. It's the responsibility of the caller to catch it.

In the following example, we are creating a simple generator that produces two values and then returns a third. Notice how we have to catch the exception in order to get this value, and how it's stored precisely inside the exception under the attribute named `value`:

```
>>> def generator():
...     yield 1
...     yield 2
...     return 3
...
>>> value = generator()
>>> next(value)
1
>>> next(value)
2
>>> try:
...     next(value)
... except StopIteration as e:
...     print(f">>>>>> returned value {e.value}")
...
>>>>>> returned value 3
```

3.2.2. Delegating into smaller coroutines: the yield from syntax

The previous feature is interesting in the sense that it opens up a lot of new possibilities with coroutines (generators), now that they can return values. But this feature, by itself, would not be so useful without proper syntax support, because catching the returned value this way is a bit cumbersome.

This is one of the main features of the yield from syntax. Among other things (that we'll review in detail), it can collect the value returned by a sub-generator. Remember that we said that returning data in a generator was nice, but that, unfortunately, writing statements as `value = generator()` wouldn't work. Well, writing it as `value = yield from generator()` would.

3.2.2.1. The simplest use of yield from

In its most basic form, the new `yield from` syntax can be used to chain generators from nested for loops into a single one, which will end up with a single string of all the values in a continuous stream.

The canonical example is about creating a function similar to `itertools.chain()` from the standard library. This is a very nice function because it allows you to pass any number of iterables and will return them all together in one stream.

The naive implementation might look like this:

```
def chain(*iterables):
    for it in iterables:
        for value in it:
            yield value
```

It receives a variable number of iterables, traverses through all of them, and since each value is iterable, it supports a `for... in..` construction, so we have another for loop to get every value inside each particular iterable, which is produced by the caller function. This might be helpful in multiple cases, such as chaining generators together or trying to iterate things that it wouldn't normally be possible to compare in one go (such as lists with tuples, and so on).

However, the `yield from` syntax allows us to go further and avoid the nested loop because it's able to produce the values from a sub-generator directly. In this case, we could simplify the code like this:

```
def chain(*iterables):
    for it in iterables:
        yield from it
```

Notice that for both implementations, the behavior of the generator is exactly the same:

```
>>> list(chain("hello", ["world"], ("tuple", " of ", "values.")))
['h', 'e', 'l', 'l', 'o', 'world', 'tuple', ' of ', 'values.']
```

This means that we can use `yield from` over any other iterable, and it will work as if the top-level generator (the one the `yield from` is using) were generating those values itself.

This works with any iterable, and even generator expressions aren't the exception. Now that we're familiar with its syntax, let's see how we could write a simple generator function that will produce all the powers of a number (for instance, if provided with `all_powers(2, 3)`, it will have to produce 2^0, 2^1,... 2^3):

```
def all_powers(n, pow):
    yield from (n ** i for i in range(pow + 1))
```

While this simplifies the syntax a bit, saving one line of a for statement isn't a big advantage, and it wouldn't justify adding such a change to the language.

Indeed, this is actually just a side effect and the real raison d'être of the `yield from` construction is what we are going to explore in the following two sections.

3.2.2.2. Capturing the value returned by a sub-generator

In the following example, we have a generator that calls another two nested generators, producing values in a sequence. Each one of these nested generators returns a value, and we will see how the top-level generator is able to effectively capture the return value since it's calling the internal generators through `yield from`:

```
def sequence(name, start, end):
    logger.info(f"{name} started at {start}")
    yield from range(start, end)
    logger.info(f"{name} finished at {end}")
    return end
```

(continues on next page)

(continued from previous page)

```
def main():
    step1 = yield from sequence("first", 0, 5)
    step2 = yield from sequence("second", step1, 10)
    return step1 + step2
```

This is a possible execution of the code in main while it's being iterated:

```
>>> g = main()
>>> next(g)
INFO:generators_yieldfrom_2:first started at 0
0
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)
4
>>> next(g)
INFO:generators_yieldfrom_2:first finished at 5
INFO:generators_yieldfrom_2:second started at 5
5
>>> next(g)
6
>>> next(g)
7
>>> next(g)
8
>>> next(g)
9
>>> next(g)
INFO:generators_yieldfrom_2:second finished at 10
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration: 15
```

The first line of main delegates into the internal generator, and produces the values, extracting them directly from it. This is nothing new, as we have already seen. Notice, though, how the `sequence()` generator function returns the end value, which is assigned in the first line to the variable named `step1`, and how this value is correctly used at the start of the following instance of that generator.

In the end, this other generator also returns the second `end` value, and the main generator, in turn, returns the sum of them, which is the value we see once the iteration has stopped.

Tip: We can use `yield from` to capture the last value of a coroutine after it has finished its processing.

3.2.2.3. Sending and receiving data to and from a sub-generator

Now, we will see the other nice feature of the `yield from` syntax, which is probably what gives it its full power. As we have already introduced when we explored generators acting as coroutines, we know that we can send values and throw exceptions at them, and, in such cases, the coroutine will either receive the value for its internal processing, or it will have to handle the exception accordingly.

If we now have a coroutine that delegates into other ones (such as in the previous example), we would also like to preserve this logic. Having to do so manually would be quite complex if we didn't have this handled by `yield from` automatically.

In order to illustrate this, let's keep the same top-level generator (main) unmodified with respect to the previous example (calling other internal generators), but let's modify the internal generators to make them able to receive values and handle exceptions. The code is probably not idiomatic, only for the purposes of showing how this mechanism works:

```
def sequence(name, start, end):
    value = start
    logger.info("%s started at %i", name, value)

    while value < end:
        try:
            received = yield value
            logger.info("%s received %r", name, received)
            value += 1

        except CustomException as e:
            logger.info("%s is handling %s", name, e)
            received = yield "OK"

    return end
```

Now, we will call the main coroutine, not only by iterating it, but also by passing values and throwing exceptions at it to see how they are handled inside sequence :

```
>>> g = main()
>>> next(g)
INFO: first started at 0
0
>>> next(g)
INFO: first received None
1
>>> g.send("value for 1")
INFO: first received 'value for 1'
2
>>> g.throw(CustomException("controlled error"))
INFO: first is handling controlled error
'OK'
... # advance more times
INFO:second started at 5
5
>>> g.throw(CustomException("exception at second generator"))
INFO: second is handling exception at second generator
'OK'
```

This example is showing us a lot of different things. Notice how we never send values to `sequence`, but only to main, and even so, the code that is receiving those values is the nested generators. Even though we never explicitly send anything to `sequence`, it's receiving the data as it's being passed along by yield from .

The main coroutine calls two other coroutines internally, producing their values, and it will be suspended at a particular point in time in any of those. When it's stopped at the first one, we can see the logs telling us that it is that instance of the coroutine that received the value we sent. The same happens when we throw an exception to it. When the first coroutine finishes, it returns the value that was assigned in the variable named `step1`, and

passed as input for the second coroutine, which will do the same (it will handle the `send()` and `throw()` calls, accordingly).

The same happens for the values that each coroutine produces. When we are at any given step, the return from calling `send()` corresponds to the value that the subcoroutine (the one that main is currently suspended at) has produced. When we throw an exception that is being handled, the sequence coroutine produces the value OK, which is propagated to the called (`main`), and which in turn will end up at main's caller.

17.4 4. Asynchronous programming

With the constructions we have seen so far, we are able to create asynchronous programs in Python. This means that we can create programs that have many coroutines, schedule them to work in a particular order, and switch between them when they're suspended after a `yield from` has been called on each of them.

The main advantage that we can take out of this is the possibility of parallelizing I/O operations in a non-blocking way. What we would need is a low-level generator (usually implemented by a third-party library) that knows how to handle the actual I/O while the coroutine is suspended. The idea is for the coroutine to effect suspension so that our program can handle another task in the meantime. The way the application would retrieve the control back is by means of the `yield from` statement, which will suspend and produce a value to the caller (as in the examples we saw previously when we used this syntax to alter the control flow of the program).

This is roughly the way asynchronous programming had been working in Python for quite a few years, until it was decided that better syntactic support was needed.

The fact that coroutines and generators are technically the same causes some confusion. Syntactically (and technically), they are the same, but semantically, they are different. We create generators when we want to achieve efficient iteration. We typically create coroutines with the goal of running non-blocking I/O operations.

While this difference is clear, the dynamic nature of Python would still allow developers to mix these different type of objects, ending up with a runtime error at a very late stage of the program. Remember that in the simplest and most basic form of the `yield from` syntax, we used this construction over iterables (we created a sort of chain function applied over strings, lists, and so on). None of these objects were coroutines, and it still worked. Then, we saw that we can have multiple coroutines, use `yield from` to send the value (or exceptions), and get some results back. These are clearly two very different use cases, however, if we write something along the lines of the following statement: `result = yield from iterable_or_awaitable()`

It's not clear what `iterable_or_awaitable returns`. It can be a simple iterable such as a string, and it might still be syntactically correct. Or, it might be an actual coroutine. The cost of this mistake will be paid much later.

For this reason, the typing system in Python had to be extended. Before Python 3.5, coroutines were just generators with a @coroutine decorator applied, and they were to be called with the yield from syntax. Now, there is a specific type of object, that is, a coroutine.

This change heralded, syntax changes as well. The `await` and `async def` syntax were introduced. The former is intended to be used instead of `yield from`, and it only works with awaitable objects (which coroutines conveniently happen to be). Trying to call `await` with something that doesn't respect the interface of an awaitable will raise an exception. The `async def` is the new way of defining coroutines, replacing the aforementioned decorator, and this actually creates an object that, when called, will return an instance of a coroutine.

Without going into all the details and possibilities of asynchronous programming in Python, we can say that despite the new syntax and the new types, this is not doing anything fundamentally different from concepts we have covered.

The idea of programming asynchronously in Python is that there is an event loop (typically `asyncio` because it's the one that is included in the standard library, but there are many others that will work just the same) that manages a series of coroutines. These coroutines belong to the event loop, which is going to call them according to its scheduling mechanism. When each one of these runs, it will call our code (according to the logic we have defined inside the coroutine we programmed), and when we want to get control back to the event loop, we call `await <coroutine>`, which will process a task asynchronously. The event loop will resume and another coroutine will take place while that operation is left running.

In practice, there are more particularities and edge cases that are beyond the scope. It is, however, worth mentioning that these concepts are related to the ideas introduced in this chapter and that this arena is another place where generators demonstrate being a core concept of the language, as there are many things constructed on top of them.

CHAPTER

EIGHTEEN

MRO AND ACCESSING METHODS FROM SUPERCLASSES

`super` is a built-in class that can be used to access an attribute belonging to an object's superclass.

Important: The Python official documentation lists super as a built-in function, but, it's a built-in class, even if it is used like a function:

```
>>> super
<class 'super'>
>>> isinstance(super, type)
```

Its usage is a bit confusing if you are used to accessing a class attribute or method by calling the parent class directly and passing self as the first argument. This is a really old pattern, but still can be found in some code bases (especially in legacy projects). See the following code:

```
class Mama:
    def says(self):
        print('do your homework')

class Sister(Mama):
    def says(self):
        Mama.says(self)
        print('and clean your bedroom')
```

Look particularly at the `Mama.says(self)` line. You can see here an explicit use of parent class. This means that the `says()` method belonging to `Mama` will be called. But, the instance on which it will be called is provided as the self argument, which is an instance of `Sister` in this case.

Instead, the super usage would be as follows:

```
class Sister(Mama):
    def says(self):
        super(Sister, self).says()
        print('and clean your bedroom')
```

Alternatively, you can also use the shorter form of the `super()` call:

```
class Sister(Mama):
    def says(self):
        super().says()
        print('and clean your bedroom')
```

The shorter form of `super` (without passing any arguments) is allowed inside the methods, but the usage of `super` is not limited to the body of methods. It can be used in any code area where the explicit call to the method of superclass implementation is required. Still, if super is not used inside the body of the method, then, all of its arguments are mandatory:

```
>>> anita = Sister()
>>> super(anita.__class__, anita).says()
do your homework
```

The final and most important thing that should be noted about `super` is that its second argument is optional. When only the first argument is provided, then super returns an unbounded type. This is especially useful when working with `classmethod`:

```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings

    def __repr__(self):
        return "Pizza with " + " and ".join(self.toppings)

    @classmethod
    def recommend(cls):
        """Recommend some pizza with arbitrary toppings,"""
        return cls(['spam', 'ham', 'eggs'])

class VikingPizza(Pizza):
    @classmethod
    def recommend(cls):
        """Use same recommendation as super but add extra spam"""
        recommended = super(VikingPizza).recommend()
        recommended.toppings += ['spam'] * 5
        return recommended
```

Note that the zero-argument `super()` form is also allowed for methods decorated with the classmethod decorator. `super()`, if called without arguments in such methods, is treated as having only the first argument defined.

The use cases presented earlier are very simple to follow and understand, but when you face a multiple inheritance schema, it becomes hard to use `super`. Before explaining these problems, you need to first understand when `super` should be avoided and how the **Method Resolution Order (MRO)** works in Python.

18.1 1. Old-style classes and super in Python 2

`super()` in Python 2 works almost exactly the same as in Python 3. The only difference in its call signature is that the shorter, zero-argument form is not available, so at least one of the expected arguments must always be provided.

Another important thing for programmers to note who want to write cross-version compatible code is that super in Python 2 works only for new-style classes. The earlier versions of Python did not have a common ancestor for all classes in the form of an object type. The old behavior was left in every Python 2.x branch release for backward compatibility, so, in those versions, if the class definition has no ancestor specified, it is interpreted as an old-style class, and it cannot use `super`:

```
class OldStyle1:
    pass

class OldStyle2(OldStyle1):
    pass
```

The new-style class in Python 2 must explicitly inherit from the object type or other new- style class:

```
class NewStyleClass(object):
    pass

class NewStyleClassToo(NewStyleClass):
    pass
```

Python 3 no longer maintains the concept of old-style classes, so any class that does not inherit from any other class implicitly inherits from `object`. This means that explicitly stating that a class inherits from object may seem redundant. Standard good practice is to not include redundant code, but removing such redundancy in this case is a good approach only for projects that no longer target any of the Python 2 versions. Code that aims for cross-version compatibility of Python must always include `object` as an ancestor of base classes, even if this is redundant in Python 3. Not doing so will result in such classes being interpreted as old-style, and this will eventually lead to issues that are very hard to diagnose.

18.2 2. Understanding Python's Method Resolution Order

Python MRO is based on C3, the MRO built for the Dylan programming language (http://opendylan.org). The reference document, written by Michele Simionato, can be found at http://www.python.org/download/releases/2.3/mro . It describes how C3 builds the **linearization** of a class, also called **precedence**, which is an ordered list of the ancestors. This list is used to seek an attribute. The C3 algorithm is described in more detail later in this section.

The MRO change was made to resolve an issue introduced with the creation of a common base type (that is, `object` type). Before the change to the C3 linearization method, if a class had two ancestors, the order in which methods were resolved was quite simple to compute and track only for simple cases that didn't use multiple inheritance model in a cascading way.

Here is an example of code, which, under Python 2, would not use C3 as an MRO:

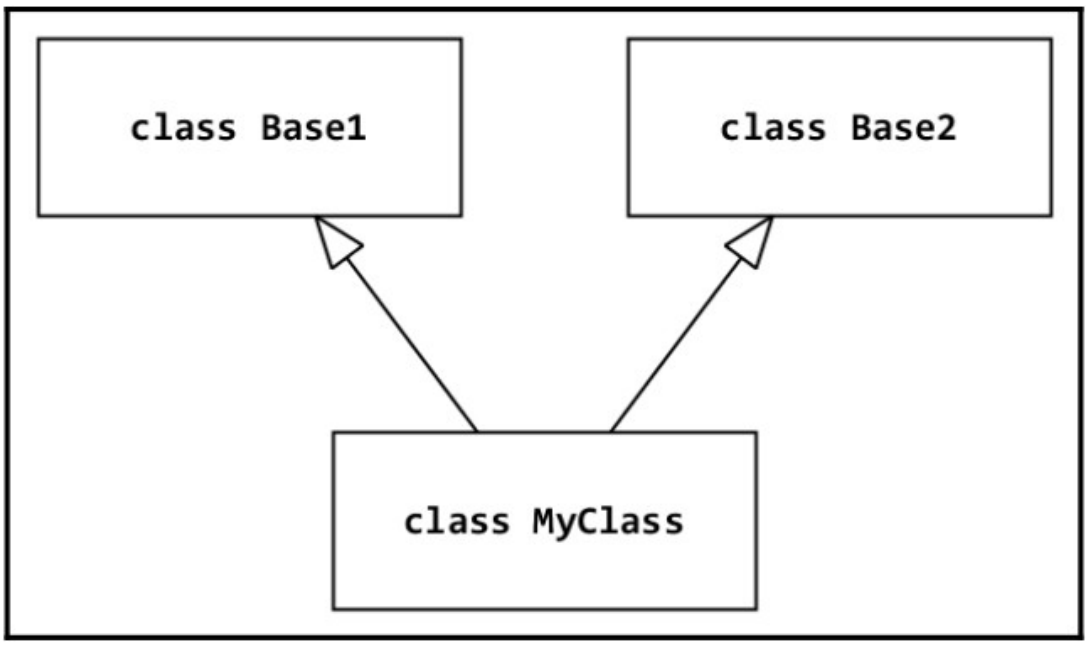
```
class Base1:
    pass

class Base2:
    def method(self):
        print('Base2')

class MyClass(Base1, Base2):
    pass

>>> MyClass().method()
Base2
```

When `MyClass().method()` is called, the interpreter looks for the method in `MyClass`, then `Base1`, and then eventually finds it in `Base2`:



When we introduce some `CommonBase` class at the top of our class hierarchy (both `Base1` and `Base2` will inherit from it), things will get more complicated. As a result, the simple resolution order that behaves according to the **left-to- right depth first** rule is getting back to the top through the `Base1` class before looking into the `Base2` class. This algorithm results in a counterintuitive output. In some cases, the method that is executed may not be the one that is the closest in the inheritance tree.

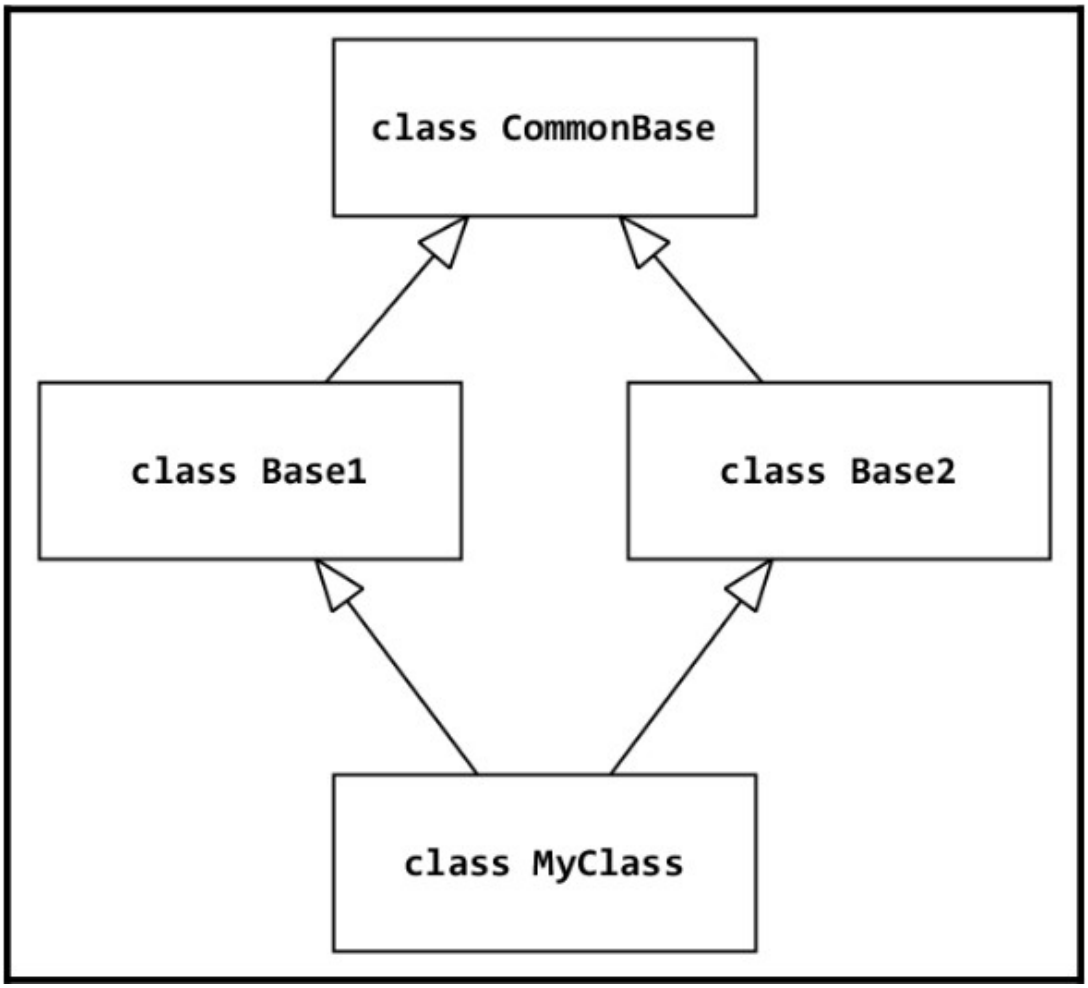
Such an algorithm is still available in Python 2 for old-style classes. Here is an example of the old method resolution in Python 2 using old-style classes:

```
class CommonBase:
    def method(self):
    print('CommonBase')

class Base1(CommonBase):
    pass

class Base2(CommonBase):
    def method(self):
        print('Base2')

class MyClass(Base1, Base2):
    pass
```



The following transcript from the interactive session shows that `Base2.method()` will not be called despite the `Base2` class being closer in the class hierarchy to `MyClass` than `CommonBase`:

```
>>> MyClass().method()
CommonBase
```

Such an inheritance scenario is extremely uncommon, so this is more a problem of theory than practice. The standard library does not structure the inheritance hierarchies in this way, and many developers think that it is bad practice. But, with the introduction of object at the top of the types hierarchy, the multiple inheritance problem pops up on the C side of the language, resulting in conflicts when doing subtyping. You should also note that every class in Python 3 has now got the same common ancestor. Since making it work properly with the existing MRO involved too much work, a new MRO was a simpler and quicker solution.

So, the same example run under Python 3 gives a different result:

```
class CommonBase:
    def method(self):
        print('CommonBase')

class Base1(CommonBase):
    pass

class Base2(CommonBase):
    def method(self):
        print('Base2')

class MyClass(Base1, Base2):
    pass
```

And here is the usage example showing that C3 serialization will pick the method of the closest ancestor:

```
>>> MyClass().method()
Base2
```

Tip: Note that the preceding behavior cannot be replicated in Python 2 without the `CommonBase` class explicitly inheriting from `object`. Reasons as to why it may be useful to specify `object` as a class ancestor in Python 3, even if this is redundant, were already mentioned.

The Python MRO is based on a recursive call over the base classes. To summarize the Michele Simionato paper referenced at the beginning of this section, the C3 symbolic notation applied to our example is as follows:

```
L[MyClass(Base1, Base2)] = MyClass + merge(L[Base1], L[Base2], Base1, Base2)
```

Here, `L[MyClass]` is the linearization of `MyClass`, and `merge` is a specific algorithm that merges several linearization results.

So, a synthetic description would be, as Simionato says:

> *"The linearization of C is the sum of C plus the merge of the linearizations of the parents and the list of the parents."*

The `merge` algorithm is responsible for removing the duplicates and preserving the correct ordering. It is described in the paper like this (adapted to our example):

> *Take the head of the first list, that is, L[Base1][0]; if this head is not in the tail of any of the other lists, then add it to the linearization of MyClass and remove it from the lists in the merge, otherwise look at the head of the next list and take it, if it is a good head.*
>
> *Then, repeat the operation until all the classes are removed or it is impossible to find good heads. In this case, it is impossible to construct the merge, Python 2.3 will refuse to create the MyClass class and will raise an exception.*

The `head` is the first element of a list and the `tail` contains the rest of the elements. For example, in (`Base1`, `Base2`, ..., `BaseN`), `Base1` is the `head` , and (`Base2`, ..., `BaseN`) is the `tail`.

In other words, C3 does a recursive depth lookup on each parent to get a sequence of lists. Then, it computes a left-to-right rule to merge all lists with a hierarchy disambiguation, when a class is involved in several lists.

So the result is as follows:

```
def L(klass):
    return [k.__name__ for k in klass.__mro__]

>>> L(MyClass)
['MyClass', 'Base1', 'Base2', 'CommonBase', 'object']
```

Tip: The `__mro__` attribute of a class (which is read-only) stores the result of the linearization computation. Computation is done when the class definition is loaded.

You can also call `MyClass.mro()` to compute and get the result. This is another reason why classes in Python 2 should be taken with an extra case. While old-style classes in Python 2 have some defined order in which methods are resolved, they do not provide the `__mro__` attribute and the `mro()` method. So, despite the order of resolution, it is wrong to say that they have MRO. In most cases, whenever someone refers to MRO in Python, it means that they are referring to the C3 algorithm described in this section.

18.3 3. Super pitfalls

Now, back to the `super()` call. If you deal with multiple inheritance hierarchy, it can become problematic. This is mainly due to the initialization of classes. In Python, the initialization methods (that is, the `__init__()` methods) of base classes are not implicitly called in ancestor classes if ancestor classes override `__init__()`. In such cases, you need to call superclass methods explicitly, and this can sometimes lead to initialization problems.

18.3.1 3.1. Mixing super and explicit class calls

In the following example, taken from James Knight's website (http://fuhm.net/super-harmful), a `C` class that calls initialization methods of its parent classes using the `super().__init__()` method will make the call to the `B.__init__()` class to be called twice:

```
class A:
    def __init__(self):
        print("A", end=" ")
        super().__init__()

class B:
    def __init__(self):
        print("B", end=" ")
        super().__init__()

class C(A, B):
    def __init__(self):
        print("C", end=" ")
        A.__init__(self)
        B.__init__(self)
```

Here is the output:

```
>>> print("MRO:", [x.__name__ for x in C.__mro__])
MRO: ['C', 'A', 'B', 'object']
>>> C()
C A B B <__main__.C object at 0x0000000001217C50>
```

In the preceding transcript we see that initialization of class `C` invokes the `B.__init__()` method twice. To avoid such issues, super should be used in the whole class hierarchy. The problem is that sometimes, a part of such complex hierarchy may be located in a third- party code. Many other related pitfalls on the hierarchy calls introduced by multiple inheritances can be found on James's page.

Unfortunately, you cannot be sure that external packages use `super()` in their code. Whenever you need to subclass some third-party class, it is always a good approach to take a look inside its code and the code of other classes in the MRO. This may be tedious, but, as a bonus, you get some information about the quality of code provided by such a package and more understanding of its code. You may learn something new that way.

18.3.2 3.2. Heterogeneous arguments

Another issue with `super` usage occurs if methods of classes within the class hierarchy use inconsistent argument sets. How can a class call its base class an `__init__()` code if it doesn't have the same signature? This leads to the following problem:

```
class CommonBase:
    def __init__(self):
        print('CommonBase')
        super().__init__()

class Base1(CommonBase):
    def __init__(self):
        print('Base1')
        super().__init__()

class Base2(CommonBase):
    def __init__(self, arg):
        print('base2')
        super().__init__()

class MyClass(Base1 , Base2):
    def __init__(self, arg):
        print('my base')
        super().__init__(arg)
```

An attempt to create a `MyClass` instance will raise `TypeError` due to a mismatch of the parent classes' `__init__()` signatures:

```
>>> MyClass(10)
my base
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "<stdin>", line 4, in __init__
TypeError: __init__() takes 1 positional argument but 2 were given
```

One solution would be to use arguments and keyword arguments packing with `*args` and `**kwargs` magic so that all constructors pass along all the parameters, even if they do not use them:

```
class CommonBase:
    def __init__(self, *args, **kwargs):
        print('CommonBase')
        super().__init__()
class Base1(CommonBase):
    def __init__(self, *args, **kwargs):
        print('Base1')
        super().__init__(*args, **kwargs)

class Base2(CommonBase):
    def __init__(self, *args, **kwargs):
        print('base2')
```

(continues on next page)

(continued from previous page)

```
        super().__init__(*args, **kwargs)

class MyClass(Base1 , Base2):
    def __init__(self, arg):
        print('my base')
        super().__init__(arg)
```

With this approach, the parent class signatures will always match:

```
>>> _ = MyClass(10)
my base
Base1
base2
CommonBase
```

This is an awful fix though, because it makes all constructors accept any kind of parameters. It leads to weak code, since anything can be passed and gone through. Another solution is to use the explicit `__init__()` calls of specific classes in `MyClass`, but this would lead to the first pitfall.

18.4 4. Best practices

To avoid all the aforementioned problems, and until Python evolves in this field, we need to take into consideration the following points:

- **Multiple inheritance should be avoided**: It can be replaced with some design patterns.
- **Super usage has to be consistent**: In a class hierarchy, super should be used everywhere or nowhere. Mixing super and classic calls is a confusing practice. People tend to avoid super to render their code more explicit.
- **Explicitly inherit from an object in Python 3 if you target Python 2 too**: Classes without any ancestor specified are recognized as old-style classes in Python 2. Mixing old-style classes with new-style classes should be avoided in Python 2.
- **Class hierarchy has to be looked over when a parent class method is called**: To avoid any problems, every time a parent class method is called, a quick glance at the MRO involved (with __mro__) is necessary.

CHAPTER

NINETEEN

METAPROGRAMMING

Metaprogramming is one of the most complex and powerful approaches to programming in Python. Metaprogramming tools and techniques have evolved with Python; so, before we dive into this topic, it is important for you to know all the elements of modern Python syntax well.

19.1 1. What is metaprogramming

Maybe there is a good academic definition of metaprogramming that we can cite here, but this is more about good software craftsmanship than about computer science theory. This is why we will use the following simple definition:

"Metaprogramming is a technique of writing computer programs that can treat themselves as data, so they can introspect, generate, and/or modify itself while running."

Using this definition, we can distinguish between two major approaches to metaprogramming in Python.

The first approach concentrates on the language's ability to introspect its basic elements, such as functions, classes, or types, and to create or modify them on the fly. Python really provides a lot of tools in this area. This feature of the Python language is used by IDEs (such as PyCharm) to provide real-time code analysis and name suggestions. The easiest possible metaprogramming tools in Python that utilized language introspection are decorators that allow for adding extra functionality to the existing functions, methods, or classes. Next are special methods of classes that allow you to interfere with class instance process creation. The most powerful are metaclasses, which allow programmers to even completely redesign Python's implementation of object-oriented programming.

The second approach allows programmers to work directly with code, either in its raw (plain text) format or in more programmatically accessible abstract syntax tree (AST) form. This second approach is, of course, more complicated and difficult to work with but allows for really extraordinary things, such as extending Python's language syntax or even creating your own **domain-specific language (DSL)**.

19.2 2. Decorators

The decorator syntax was already explained, as a syntactic sugar for the following simple pattern:

```
def decorated_function():
    pass

decorated_function = some_decorator(decorated_function)
```

This verbose form of function decoration clearly shows what the decorator does. It takes a function object and modifies it at runtime. As a result, a new function (or anything else) is created based on the previous function object with the same name. This decoration may be a complex operation that performs some code introspection or decorated function to give different results depending on how the original function was implemented. All this means is that decorators can be considered as a metaprogramming tool.

This is good news. The basics of decorators are relatively easy to grasp and in most cases make code shorter, easier to read, and also cheaper to maintain. Other metaprogramming tools that are available in Python are more difficult to understand and master. Also, they might not make the code simple at all.

19.2.1 2.1. Class decorators

One of the lesser known syntax features of Python are the class decorators. Their syntax and implementation is exactly the same as function decorators. The only difference is that they are expected to return a class instead of the function object. Here is an example class decorator that modifies the `__repr__()` method to return the printable object representation, which is shortened to some arbitrary number of characters:

```
def short_repr(cls):
    cls.__repr__ = lambda self: super(cls, self).__repr__()[:8]
    return cls

@short_repr
class ClassWithRelativelyLongName:
    pass
```

The following is what you will see in the output:

```
>>> ClassWithRelativelyLongName()
<ClassWi
```

Of course, the preceding snippet is not an example of good code by any means. Still, it shows how multiple language features that are explained in the previous chapter can be used together, for example:

- Not only instances but also class objects can be modified at runtime
- Functions are descriptors too, so they can be added to the class at runtime because the actual method binding is performed on the attribute lookup as part of the descriptor protocol
- The `super()` call can be used outside of a class definition scope as long as proper arguments are provided
- Finally, class decorators can be used on class definitions

The other aspects of writing function decorators apply to the class decorators as well. Most importantly, they can use closures and be parametrized. Taking advantage of these facts, the previous example can be rewritten into the following more readable and maintainable form:

```
def parametrized_short_repr(max_width=8):
    """Parametrized decorator that shortens representation"""

    def parametrized(cls):
        """Inner wrapper function that is actual decorator"""

        class ShortlyRepresented(cls):
            """Subclass that provides decorated behavior"""
            def __repr__(self):
                return super().__repr__()[:max_width]

        return ShortlyRepresented

    return parametrized
```

The major drawback of using closures in class decorators this way is that the resulting objects are no longer instances of the class that was decorated but instances of the subclass that was created dynamically in the decorator function. Among others, this will affect the class's `__name__` and `__doc__` attributes, as follows:

```
@parametrized_short_repr(10)
class ClassWithLittleBitLongerLongName:
    pass
```

Such usage of class decorators will result in the following changes to the class metadata:

```
>>> ClassWithLittleBitLongerLongName().__class__
<class 'ShortlyRepresented'>
>>> ClassWithLittleBitLongerLongName().__doc__
'Subclass that provides decorated behavior'
```

Unfortunately, this cannot be fixed as simply as we explained before. In class decorators, you can't simply use the additional `wraps` decorator to preserve the original class type and metadata. This makes use of the class decorators in this form limited in some circumstances. They can, for instance, break results of automated documentation generation tools.

Still, despite this single caveat, class decorators are a simple and lightweight alternative to the popular mixin class pattern. Mixin in Python is a class that is not meant to be instantiated, but is instead used to provide some reusable API or functionality to other existing classes. Mixin classes are almost always added using multiple inheritance. Their usage usually takes the following form:

```
class SomeConcreteClass(MixinClass, SomeBaseClass):
    pass
```

Mixins classes form a useful design pattern that is utilized in many libraries and frameworks. To name one, Django is an example framework that uses them extensively. While useful and popular, mixins can cause some trouble if not designed well, because, in most cases, they require the developer to rely on multiple inheritance. As we stated earlier, Python handles multiple inheritance relatively well, thanks to its clear MRO implementation. Anyway, try to avoid subclassing multiple classes if you can. Multiple inheritance makes code more complex and hard to reason about. This is why class decorators may be a good replacement for mixin classes.

19.3 4. Using __new__() for overriding instantiation

The special method `__new__()` is a static method that's responsible for creating class instances. It is special-cased, so there is no need to declare it as static using the `staticmethod` decorator. This `__new__(cls, [,...])` method is called prior to the `__init__()` initialization method. Typically, the implementation of overridden `__new__()` invokes its superclass version using `super().__new__()` with suitable arguments and modifies the instance before returning it.

The following is an example class with the overridden `__new__()` method implementation in order to count the number of class instances:

```
class InstanceCountingClass:

    instances_created = 0

    def __new__(cls, *args, **kwargs):
        print('__new__() called with:', cls, args, kwargs)
        instance = super().__new__(cls)
        instance.number = cls.instances_created
        cls.instances_created += 1

        return instance

    def __init__(self, attribute):
        print('__init__() called with:', self, attribute)
        self.attribute = attribute
```

Here is the log of the example interactive session that shows how our InstanceCountingClass implementation works:

```
>>> from instance_counting import InstanceCountingClass
>>> instance1 = InstanceCountingClass('abc')
```

(continues on next page)

(continued from previous page)

```
__new__() called with: <class '__main__.InstanceCountingClass'> ('abc',) {}
__init__() called with: <__main__.InstanceCountingClass object at
0x101259e10> abc
>>> instance2 = InstanceCountingClass('xyz')
__new__() called with: <class '__main__.InstanceCountingClass'> ('xyz',) {}
__init__() called with: <__main__.InstanceCountingClass object at
0x101259dd8> xyz
>>> instance1.number, instance1.instances_created
(0, 2)
>>> instance2.number, instance2.instances_created
(1, 2)
```

The `__new__()` method should usually return an instance of the featured class, but it is also possible for it to return other class instances. If this does happen (a different class instance is returned), then the call to the `__init__()` method is skipped. This fact is useful when there is a need to modify creation/initialization behavior of immutable class instances like some of Python's built-in types, as shown in the following code:

```
class NonZero(int):
    def __new__(cls, value):
        return super().__new__(cls, value) if value != 0 else None

    def __init__(self, skipped_value):
        # implementation of __init__ could be skipped in this case
        # but it is left to present how it may be not called
        print("__init__() called")
        super().__init__()
```

Let's review these in the following interactive session:

```
>>> type(NonZero(-12))
__init__() called
<class '__main__.NonZero'>
>>> type(NonZero(0))
<class 'NoneType'>
>>> NonZero(-3.123)
__init__() called
-3
```

So, when should we use `__new__()`? The answer is simple: only when `__init__()` is not enough. One such case was already mentioned, that is, subclassing immutable built-in Python types such as `int`, `str`, `float`, `frozenset`, and so on. This is because there was no way to modify such an immutable object instance in the `__init__()` method once it was created.

Some programmers can argue that `__new__()` may be useful for performing important object initialization that may be missed if the user forgets to use the `super().__init__()` call in the overridden initialization method. While it sounds reasonable, this has a major drawback. With such an approach, it becomes harder for the programmer to explicitly skip previous initialization steps if this is the already desired behavior. It also breaks an unspoken rule of all initializations performed in `__init__()`.

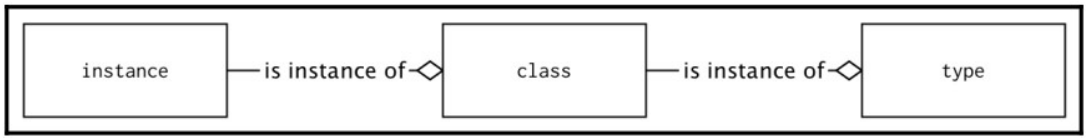
Because `__new__()` is not constrained to return the same class instance, it can be easily abused. Irresponsible usage of this method might do a lot of harm to code readability, so it should always be used carefully and backed with extensive documentation. Generally, it is better to search for other solutions that may be available for the given problem, instead of affecting object creation in a way that will break a basic programmers' expectations. Even overridden initialization of immutable types can be replaced with more predictable and well-established design patterns like the Factory Method.

There is at least one aspect of Python programming where extensive usage of the `__new__()` method is well justified. These are metaclasses.

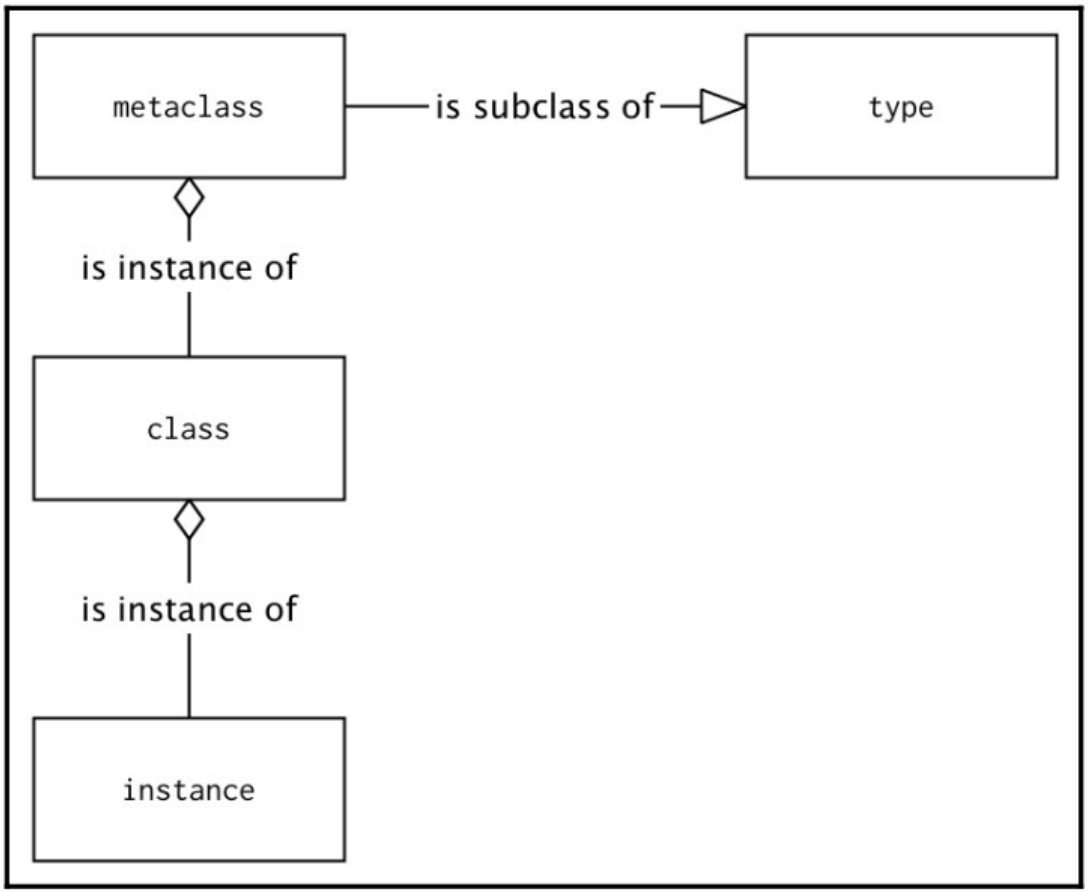
19.4 5. Metaclasses

Metaclass is a Python feature that is considered by many as one of the most difficult things to understand in this language and thus avoided by a great number of developers. In reality, it is not as complicated as it sounds once you understand a few basic concepts. As a reward, knowing how to use metaclasses grants you the ability to do things that are not possible without them.

Metaclass is a type (class) that defines other types (classes). The most important thing to know in order to understand how they work is that classes that define object instances are objects too. So, if they are objects, then they have an associated class. The basic type of every class definition is simply the built-in `type` class. Here is a simple diagram that should make this clear:



In Python, it is possible to substitute the metaclass for a class object with our own type. Usually, the new metaclass is still the subclass of the `type` class because not doing so would make the resulting classes highly incompatible with other classes in terms of inheritance:



19.4.1 5.1. The general syntax

The call to the built-in `type()` class can be used as a dynamic equivalent of the class statement. The following is an example of a class definition with the `type()` call:

```
def method(self):
    return 1

MyClass = type('MyClass', (object,), {'method': method})
```

This is equivalent to the explicit definition of the class with the `class` keyword:

```
class MyClass:
    def method(self):
        return 1
```

Every class that's created with the `class` statement implicitly uses `type` as its metaclass. This default behavior can be changed by providing the `metaclass` keyword argument to the class statement, as follows:

```
class ClassWithAMetaclass(metaclass=type):
    pass
```

The value that's provided as a `metaclass` argument is usually another class object, but it can be any other callable that accepts the same arguments as the `type` class and is expected to return another class object. The call signature is `type(name, bases, namespace)` and the meaning of the arguments are as follows:

- `name`: This is the name of the class that will be stored in the `__name__` attribute
- `bases`: This is the list of parent classes that will become the `__bases__` attribute and will be used to construct the MRO of a newly created class
- `namespace`: This is a namespace (mapping) with definitions for the class body that will become the `__dict__` attribute

One way of thinking about metaclasses is the `__new__()` method, but at a higher level of class definition.

Despite the fact that functions that explicitly call `type()` can be used in place of metaclasses, the usual approach is to use a different class that inherits from type for this purpose. The common template for a metaclass is as follows:

```
class Metaclass(type):
    def __new__(mcs, name, bases, namespace):
        return super().__new__(mcs, name, bases, namespace)

    @classmethod
    def __prepare__(mcs, name, bases, **kwargs):
        return super().__prepare__(name, bases, **kwargs)

    def __init__(cls, name, bases, namespace, **kwargs):
        super().__init__(name, bases, namespace)

    def __call__(cls, *args, **kwargs):
        return super().__call__(*args, **kwargs)
```

The `name`, `bases`, and `namespace` arguments have the same meaning as in the `type()` call we explained earlier, but each of these four methods can have the following different purposes:

- `__new__(mcs, name, bases, namespace)`: This is responsible for the actual creation of the class object in the same way as it does for ordinary classes. The first positional argument is a metaclass object. In the preceding example, it would simply be a `Metaclass`. Note that `mcs` is the popular naming convention for this argument.
- `__prepare__(mcs, name, bases, **kwargs)`: This creates an empty namespace object. By default, it returns an empty `dict`, but it can be overridden to return any other mapping type. Note that it

does not accept `namespace` as an argument because, before calling it, the namespace does not exist.

- `__init__(cls, name, bases, namespace, **kwargs)`: This is not seen popularly in metaclass implementations but has the same meaning as in ordinary classes. It can perform additional class object initialization once it is created with `__new__()`. The first positional argument is now named `cls` by convention to mark that this is already a created class object (metaclass instance) and not a metaclass object. When `__init__()` was called, the class was already constructed and so this method can do less things than the `__new__()` method. Implementing such a method is very similar to using class decorators, but the main difference is that `__init__()` will be called for every subclass, while class decorators are not called for subclasses.
- `__call__(cls, *args, **kwargs)`: This is called when an instance of a metaclass is called. The instance of a metaclass is a class object; it is invoked when you create new instances of a class. This can be used to override the default way of how class instances are created and initialized.

Each of the preceding methods can accept additional extra keyword arguments, all of which are represented by `**kwargs`. These arguments can be passed to the metaclass object using extra keyword arguments in the class definition in the form of the following code:

```
class Klass(metaclass=Metaclass, extra="value"):
    pass
```

This amount of information can be overwhelming at the beginning without proper examples, so let's trace the creation of metaclasses, classes, and instances with some `print()` calls:

```
class RevealingMeta(type):
    def __new__(mcs, name, bases, namespace, **kwargs):
        print(mcs, "__new__ called")
        return super().__new__(mcs, name, bases, namespace)

    @classmethod
    def __prepare__(mcs, name, bases, **kwargs):
        print(mcs, "__prepare__ called")
        return super().__prepare__(name, bases, **kwargs)

    def __init__(cls, name, bases, namespace, **kwargs):
        print(cls, "__init__ called")
        super().__init__(name, bases, namespace)

    def __call__(cls, *args, **kwargs):
        print(cls, "__call__ called")
        return super().__call__(*args, **kwargs)
```

Using `RevealingMeta` as a metaclass to create a new class definition will give the following output in the Python interactive session:

```
>>> class RevealingClass(metaclass=RevealingMeta):
...     def __new__(cls):
...         print(cls, "__new__ called")
...         return super().__new__(cls)
...     def __init__(self):
...         print(self, "__init__ called")
...         super().__init__()
...
<class 'RevealingMeta'> __prepare__ called
<class 'RevealingMeta'> __new__ called
<class 'RevealingClass'> __init__ called
>>> instance = RevealingClass()
<class 'RevealingClass'> __call__ called <class 'RevealingClass'> __new__
called <RevealingClass object at 0x1032b9fd0> __init__ called
```

19.4.2 5.2. Metaclass usage

Metaclasses, once mastered, are a powerful feature, but always complicate the code. Metaclasses also do not compose well and you'll quickly run into problems if you try to mix multiple metaclasses through inheritance.

For simple things, like changing the read/write attributes or adding new ones, metaclasses can be avoided in favor of simpler solutions, such as properties, descriptors, or class decorators.

But there are situations where things cannot be easily done without them. For instance, it is hard to imagine Django's ORM implementation built without extensive use of metaclasses. It could be possible, but it is rather unlikely that the resulting solution would be similarly easy to use. Frameworks are the place where metaclasses really shine. They usually have a lot of complex internal code that is not easy to understand and follow, but eventually allow other programmers to write more condensed and readable code that operates on a higher level of abstraction.

19.4.3 5.3. Metaclass pitfalls

Like some other advanced Python features, the metaclasses are very elastic and can be easily abused. While the call signature of the class is rather strict, Python does not enforce the type of the return parameter. It can be anything as long as it accepts incoming arguments on calls and has the required attributes whenever it is needed.

One such object that can be *anything-anywhere* is the instance of the `Mock` class that's provided in the `unittest.mock` module. `Mock` is not a metaclass and also does not inherit from the `type` class. It also does not return the class object on instantiating. Still, it can be included as a metaclass keyword argument in the class definition, and this will not raise any syntax errors. Using `Mock` as a metaclass is, of course, complete nonsense, but let's consider the following example:

```
>>> from unittest.mock import Mock
>>> class Nonsense(metaclass=Mock):
...     pass
...
>>> Nonsense
<Mock spec='str' id='4327214664'>
# pointless, but illustrative
```

It's not hard to predict that any attempt to instantiate our Nonsense pseudo-class will fail. What is really interesting is the following exception and traceback you'll get trying to do so:

```
>>> Nonsense()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File
"/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/unittest/mock.py",
↪ line 917, in __call__
    return _mock_self._mock_call(*args, **kwargs)
File
"/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/unittest/mock.py",
↪ line 976, in _mock_call
    result = next(effect)
StopIteration
```

Does the `StopIteration` exception give you any clue that there may be a problem with our class definition on the metaclass level? Obviously not. This example illustrates how hard it may be to debug metaclass code if you don't know where to look for errors.

19.5 6. Code generation

As we already mentioned, the dynamic code generation is the most difficult approach to metaprogramming. There are tools in Python that allow you to generate and execute code or even do some modifications to the already compiled code objects.

Various projects such as **Hy** show that even whole languages can be reimplemented in Python using code generation techniques. This proves that the possibilities are practically limitless. Knowing how vast this topic is and how badly it is riddled with various pitfalls, I won't even try to give detailed suggestions on how to create code this way, or to provide useful code samples.

Anyway, knowing what is possible may be useful for you if you plan to study this field deeper by yourself. So, treat this section only as a short summary of possible starting points for further learning.

19.5.1 6.1. exec, eval and compile

Python provides the following three built-in functions to manually execute, evaluate, and compile arbitrary Python code:

- `exec(object, globals, locals)`: This allows you to dynamically execute the Python code. `object` should be a string or code object (see the `compile()` function) representing a single statement or sequence of multiple statements. The `globals` and `locals` arguments provide global and local namespaces for the executed code and are optional. If they are not provided, then the code is executed in the current scope. If provided, `globals` must be a dictionary, while `locals` might be any mapping object; it always returns `None`.
- `eval(expression, globals, locals)`: This is used to evaluate the given expression by returning its value. It is similar to `exec()`, but it expects `expression` to be a single Python expression and not a sequence of statements. It returns the value of the evaluated expression.
- `compile(source, filename, mode)`: This compiles the source into the code object or AST object. The source code is provided as a string value in the `source` argument. The filename should be the file from which the code was read. If it has no file associated (for example, because it was created dynamically), then `<string>` is the value that is commonly used. Mode should be either `exec` (sequence of statements), `eval` (single expression), or `single` (a single interactive statement, such as in a Python interactive session).

The `exec()` and `eval()` functions are the easiest to start with when trying to dynamically generate code because they can operate on strings. If you already know how to program in Python, then you may already know how to correctly generate working source code programmatically.

The most useful in the context of metaprogramming is obviously `exec()` because it allows you to execute any sequence of Python statements. The word *any* should be alarming for you. Even `eval()`, which allows only evaluation of expressions in the hands of a skillful programmer (when fed with the user input), can lead to serious security holes. Note that crashing the Python interpreter is the scenario you should be least afraid of. Introducing vulnerability to remote execution exploits due to irresponsible use of `exec()` and `eval()` can cost you your image as a professional developer, or even your job.

Even if used with a trusted input, there is a list of little details about `exec()` and `eval()` that is too long to be included here, but might affect how your application works in ways you would not expect. Armin Ronacher has a good article that lists the most important of them, titled "Be careful with exec and eval" in Python (refer to http://lucumr.pocoo.org/2011/2/1/exec-in-python/).

Despite all these frightening warnings, there are natural situations where the usage of `exec()` and `eval()` is really justified. Still, in the case of even the tiniest doubt, you should not use them and try to find a different solution.

Tip: The signature of the `eval()` function might make you think that if you provide empty `globals` and `locals` namespaces and wrap it with proper `try ... except` statements, then it will be reasonably safe. There could be nothing more wrong. Ned Batcheler has written a very good article in which he shows how to cause an interpreter segmentation fault in the `eval()` call, even with erased access to all Python built-ins (see

http://nedbatchelder.com/blog/201206/eval_really_is_dangerous.html). This is single proof that both `exec()` and `eval()` should never be used with untrusted input.

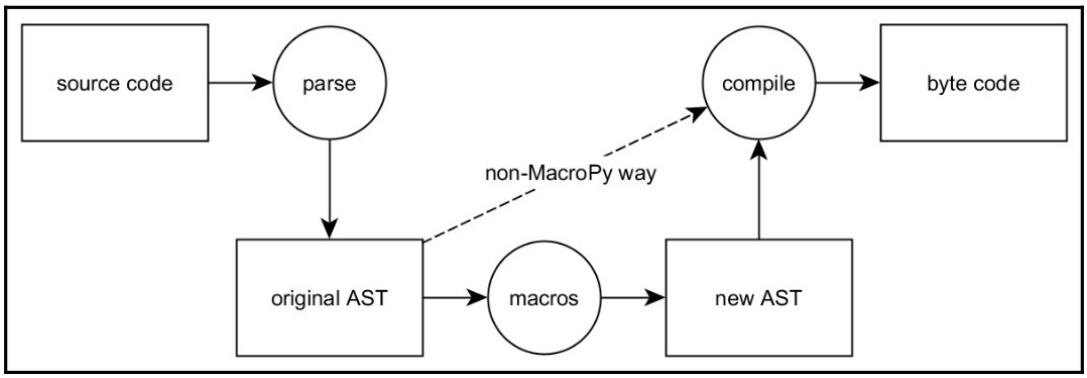
19.5.2 6.2. Abstract syntax tree (AST)

The Python syntax is converted into AST before it is compiled into byte code. This is a tree representation of the abstract syntactic structure of the source code. Processing of Python grammar is available thanks to the built-in `ast` module. Raw ASTs of Python code can be created using the `compile()` function with the `ast.PyCF_ONLY_AST` flag, or by using the `ast.parse()` helper. Direct translation in reverse is not that simple and there is no function provided in the standard library that can do so. Some projects, such as PyPy, do such things though.

The `ast` module provides some helper functions that allow you to work with the AST, for example:

```
>>> tree = ast.parse('def hello_world(): print("hello world!")')
>>> tree
<_ast.Module object at 0x00000000038E9588>
>>> ast.dump(tree)
"Module(
    body=[
        FunctionDef(
            name='hello_world',
            args=arguments(
                args=[],
                vararg=None,
                kwonlyargs=[],
                kw_defaults=[],
                kwarg=None,
                defaults=[]
            ),
            body=[
                Expr(
                    value=Call(
                        func=Name(id='print', ctx=Load()),
                        args=[Str(s='hello world!')],
                        keywords=[]
                    )
                )
            ],
            decorator_list=[],
            returns=None
        )
    ]
)"
```

The output of `ast.dump()` in the preceding example was reformatted to increase the readability and better show the tree-like structure of the AST. It is important to know that the AST can be modified before being passed to `compile()`. This gives you many new possibilities. For instance, new syntax nodes can be used for additional instrumentation, such as test coverage measurement. It is also possible to modify the existing code tree in order to add new semantics to the existing syntax. Such a technique is used by the MacroPy project (https://github.com/lihaoyi/macropy) to add syntactic macros to Python using the already existing syntax:



AST can also be created in a purely artificial manner, and there is no need to parse any source at all. This gives Python programmers the ability to create Python bytecode for custom domain-specific languages, or even completely implement other programming languages on top of Python VMs.

6.2.1. Import hooks

Taking advantage of MacroPy's ability to modify original ASTs would be as easy as using the `import macropy.activate` statement if it could somehow override the Python import behavior. Fortunately, Python provides a way to intercept imports using the following two kinds of import hooks:

- **Meta hooks**: These are called before any other `import` processing has occurred. Using meta hooks, you can override the way in which `sys.path` is processed for even frozen and built-in modules. To add a new meta hook, a new **meta path finder** object must be added to the `sys.meta_path` list.
- **Import path hooks**: These are called as part of `sys.path` processing. They are used if the path item associated with the given hook is encountered. The import path hooks are added by extending the `sys.path_hooks` list with a new **path finder** object.

The details of implementing both path finders and meta path finders are extensively implemented in the official Python documentation (see https://docs.python.org/3/reference/import.html). The official documentation should be your primary resource if you want to interact with imports on that level. This is so because import machinery in Python is rather complex and any attempt to summarize it in a few paragraphs would inevitably fail. Here, we just noted that such things are possible.

19.5.3 6.3. Projects that use code generation patterns

It is hard to find a really usable implementation of the library that relies on code generation patterns that is not only an experiment or simple proof of concept. The reasons for that situation are fairly obvious:

- Deserved fear of the `exec()` and `eval()` functions because, if used irresponsibly, they can cause real disasters
- Successful code generation is very difficult to develop and maintain because it requires a deep understanding of the language and exceptional programming skills in general

Despite these difficulties, there are some projects that successfully take this approach either to improve performance or achieve things that would be impossible by other means.

6.3.1. Falcon's compiled router

Falcon (http://falconframework.org/) is a minimalist Python WSGI web framework for building fast and lightweight APIs. It strongly encourages the REST architectural style that is currently very popular around the web. It is a good alternative to other rather heavy frameworks, such as Django or Pyramid. It is also a strong competitor to other micro- frameworks that aim for simplicity, such as Flask, Bottle, or web2py.

One of its features is it's very simple routing mechanism. It is not as complex as the routing provided by Django `urlconf` and does not provide as many features, but in most cases is just enough for any API that follows the REST architectural design. What is most interesting about Falcon's routing is the internal construction of that router. Falcon's router is implemented using the code generated from the list of routes, and code changes every time a new route is registered. This is the effort that's needed to make routing fast.

Consider this very short API example, taken from Falcon's web documentation:

```
import falcon
import json


class QuoteResource:
    def on_get(self, req, resp):
        """Handles GET requests"""

        quote = {
            'quote': 'I\'ve always been more interested in '
                     'the future than in the past.',
            'author': 'Grace Hopper'
        }

        resp.body = json.dumps(quote)

    api = falcon.API()
    api.add_route('/quote', QuoteResource())
```

In short, the highlighted call to the `api.add_route()` method updates dynamically the whole generated code tree for Falcon's request router. It also compiles it using the `compile()` function and generates the new route-finding function using `eval()`. Let's take a closer look at the following `__code__` attribute of the `api._router._find()` function:

```
>>> api._router._find.__code__
<code object find at 0x00000000033C29C0, file "<string>", line 1>
>>> api.add_route('/none', None)
>>> api._router._find.__code__
<code object find at 0x00000000033C2810, file "<string>", line 1>
```

This transcript shows that the code of this function was generated from the string and not from the real source code file (the "`<string>`" file). It also shows that the actual code object changes with every call to the `api.add_route()` method (the object's address in memory changes).

6.3.2. Hy

Hy (http://docs.hylang.org/) is the dialect of Lisp, and is written entirely in Python. Many similar projects that implement other code in Python usually try only to tokenize the plain form of code that's provided either as a file-like object or string and interpret it as a series of explicit Python calls. Unlike others, Hy can be considered as a language that runs fully in the Python runtime environment, just like Python does. Code written in Hy can use the existing built-in modules and external packages and vice-versa. Code written with Hy can be imported back into Python.

To embed Lisp in Python, Hy translates Lisp code directly into Python AST. Import interoperability is achieved using the import hook that is registered once the Hy module is imported into Python. Every module with the `.hy`

extension is treated as the Hy module and can be imported like the ordinary Python module. The following is a *hello world* program written in this Lisp dialect:

```
;; hyllo.hy
(defn hello [] (print "hello world"))
```

It can be imported and executed with the following Python code:

```
>>> import hy
>>> import hyllo
>>> hyllo.hello()
hello world
```

If we dig deeper and try to disassemble `hyllo.hello` using the built-in `dis` module, we will notice that the byte code of the Hy function does not differ significantly from its pure Python counterpart, as shown in the following code:

```
>>> import dis
>>> dis.dis(hyllo.hello)
2           0 LOAD_GLOBAL    0 (print)
            3 LOAD_CONST     1 ('hello world!')
            6 CALL_FUNCTION  1 (1 positional, 0 keyword pair)
            9 RETURN_VALUE
>>> def hello(): print("hello world!")
...
>>> dis.dis(hello)
1           0 LOAD_GLOBAL    0 (print)
            3 LOAD_CONST     1 ('hello world!')
            6 CALL_FUNCTION  1 (1 positional, 0 keyword pair)
            9 POP_TOP       10 LOAD_CONST
            0 (None)        13 RETURN_VALUE
```

CHAPTER

TWENTY

NAMING

Most of the standard library was built keeping usability in mind. Python, in this case, can be compared to the pseudocode you might think about when working on a program. Most of the code can be read out loud. For instance, this snippet could be understood even by someone that is not a programmer:

```
my_list = []
if 'd' not in my_list:
    my_list.append('d')
```

The fact that Python code is so close to natural language is one of the reasons why Python is so easy to learn and use. When you are writing a program, the flow of your thoughts is quickly translated into lines of code.

20.1 1. PEP 8

PEP 8 (http://www.python.org/dev/peps/pep-0008) provides a style guide for writing Python code. Besides some basic rules, such as indentation, maximum line length, and other details concerning the code layout, PEP 8 also provides a section on naming conventions that most of the code bases follow.

This section provides only a quick summary of PEP 8, and a handy naming guide for each kind of Python syntax element. You should still consider reading the PEP 8 document as mandatory.

20.1.1 1.1. Why and when to follow PEP 8?

If you are creating a new software package that is intended to be open sourced, you should always follow PEP 8 because it is a widely accepted standard and is used in most of the open source projects written in Python. If you want to foster any collaboration with other programmers, then you should definitely stick to PEP 8, even if you have different views on the best code style guidelines. Doing so has the benefit of making it a lot easier for other developers to jump straight into your project. Code will be easier to read for newcomers because it will be consistent in style with most of the other Python open source packages.

Also, starting with full PEP 8 compliance saves you time and trouble in the future. If you want to release your code to the public, you will eventually face suggestions from fellow programmers to switch to PEP 8. Arguments as to whether it is really necessary to do so for a particular project tend to be never-ending flame wars that are impossible to win. This is the sad truth, but you may be eventually forced to be consistent with it or risk losing valuable contributors.

Also, restyling of the whole project's code base if it is in a mature state of development might require a tremendous amount of work. In some cases, such restyling might require changing almost every line of code. While most of the changes can be automated (indentation, newlines, and trailing whitespaces), such massive code overhaul usually introduces a lot of conflicts in every version control workflow that is based on branching. It is also very hard to review so many changes at once. These are the reasons why many open source projects have a rule that style-fixing changes should always be included in separate pull/merge requests or patches that do not affect any feature or bug.

20.1.2 1.2. Team-specific style guidelines

Despite providing a comprehensive set of style guidelines, PEP 8 still leaves some freedom for the developers. Especially in terms of nested data literals and multiline function calls that require long lists of arguments. Some teams may decide that they require additional styling rules and the best option is to formalize them in some kind of document that is available for every team member.

Also, in some situations, it may be impossible or economically infeasible to be strictly consistent with PEP 8 in some old projects that had no style guide defined. Such projects will still benefit from formalization of the actual coding conventions even if they do not reflect the official set of PEP 8 rules. Remember, what is more important than consistency with PEP 8 is consistency within the project. If rules are formalized and available as a reference for every programmer, then it is way easier to keep consistency within a project and organization.

20.2 2. Naming styles

The different naming styles used in Python are:

- CamelCase
- mixedCase
- UPPERCASE and UPPER_CASE_WITH_UNDERSCORES
- lowercase and lower_case_with_underscores
- _leading and trailing_ underscores, and sometimes __doubled__ underscores

Lowercase and uppercase elements are often a single word, and sometimes a few words concatenated. With underscores, they are usually abbreviated phrases. Using a single word is better. The leading and trailing underscores are used to mark the privacy and special elements.

These styles are applied to the following:

- Variables
- Functions and methods
- Properties
- Classes
- Modules
- Packages

20.2.1 2.1. Variables

There are the following two kinds of variables in Python:

- **Constants**: These define values that are not supposed to change during program execution
- **Public and private variables**: These hold the state of applications that can change during program execution

2.1.1. Constants

For constant global variables, an uppercase with an underscore is used. It informs the developer that the given variable represents a constant value.

Note: There are no real constants in Python like those in C++, where const can be used. You can change the value of any variable. That's why Python uses a naming convention to mark a variable as a constant.

For example, the `doctest` module provides a list of option flags and directives (http://docs.python.org/lib/doctest-options.html) that are small sentences, clearly defining what each option is intended for, for example:

```
from doctest import IGNORE_EXCEPTION_DETAIL
from doctest import REPORT_ONLY_FIRST_FAILURE
```

These variable names seem rather long, but it is important to clearly describe them. Their usage is mostly located in the initialization code rather than in the body of the code itself, so this verbosity is not annoying.

Note: Abbreviated names obfuscate the code most of the time. Don't be afraid of using complete words when an abbreviation seems unclear.

Some constants' names are also driven by the underlying technology. For instance, the `os` module uses some constants that are defined on the C side, such as the `EX_XXX` series, that defines UNIX exit code numbers. Same name code can be found, as in the following example, in the system's `sysexits.h` C headers files:

```
import os
import sys


sys.exit(os.EX_SOFTWARE)
```

Another good practice when using constants is to gather all of them at the top of a module that uses them. It is also common to combine them under new variables if they are flags or enumerations that allow for such operations, for example:

```
import doctest
TEST_OPTIONS = (doctest.ELLIPSIS | doctest.NORMALIZE_WHITESPACE | doctest.REPORT_
↪ONLY_FIRST_FAILURE)
```

2.1.1.1. Naming and usage

Constants are used to define a set of values the program relies on, such as the default configuration filename.

A good practice is to gather all the constants in a single file in the package. That is how Django works, for instance. A module named `settings.py` provides all the constants as follows:

```
SQL_USER = 'tarek'
SQL_PASSWORD = 'secret'
SQL_URI = 'postgres://%s:%s@localhost/db' % (SQL_USER, SQL_PASSWORD)
MAX_THREADS = 4
```

Another approach is to use a configuration file that can be parsed with the `ConfigParser` module, or another configuration parsing tool. But some people argue that it is rather an overkill to use another file format in a language such as Python, where a source file can be edited and changed as easily as a text file.

For options that act like flags, a common practice is to combine them with Boolean operations, as the `doctest` and `re` modules do. The pattern taken from `doctest` is quite simple, as shown in the following code:

```
OPTIONS = {}

def register_option(name):
    return OPTIONS.setdefault(name, 1 << len(OPTIONS))

def has_option(options, name):
    return bool(options & name)

# now defining options
BLUE = register_option('BLUE')
RED = register_option('RED')
WHITE = register_option('WHITE')
```

This code allows for the following usage:

```
>>> # let's try them
>>> SET = BLUE | RED
>>> has_option(SET, BLUE)
True
>>> has_option(SET, WHITE)
False
```

When you define a new set of constants, avoid using a common prefix for them, unless the module has several independent sets of options. The module name itself is a common prefix.

Another good solution for option-like constants would be to use the `Enum` class from the built-in `enum` module and simply rely on the `set` collection instead of the binary operators.

Important: Using binary bit-wise operations to combine options is common in Python. The inclusive OR (`|`) operator will let you combine several options in a single integer, and the AND (`&`) operator will let you check that the option is present in the integer (refer to the `has_option` function).

2.1.2. Public and private variables

For global variables that are mutable and freely available through imports, a lowercase letter with an underscore should be used when they do not need to be protected. If a variable shouldn't be used and modified outside of its origin module we consider it a private member of that module. A leading underscore, in that case, can mark the variable as a private element of the package, as shown in the following code:

```
_observers = []

def add_observer(observer):
    _observers.append(observer)

def get_observers():
    """Makes sure _observers cannot be modified."""
    return tuple(_observers)
```

Variables that are located in functions, and methods, follow the same rules as public variables and are never marked as private since they are local to the function context.

For class or instance variables, you should use the private marker (the leading underscore) if making the variable a part of the public signature does not bring any useful information, or is redundant. In other words, if the variable is used only internally for the purpose of some other method that provides an actual public feature, it is better to make it private.

For instance, the attributes that are powering a property are good private citizens, as shown in the following code:

```
class Citizen(object):

    def __init__(self, first_name, last_name):
        self._first_name = first_name
        self._last_name = last_name

    @property
    def full_name(self):
        return f"{self._first_name} {self._last_name}"
```

Another example would be a variable that keeps some internal state that should not be disclosed to other classes. This value is not useful for the rest of the code, but participates in the behavior of the class:

```
class UnforgivingElephant(object):

    def __init__(self, name):
        self.name = name
        self._people_to_stomp_on = []

    def get_slapped_by(self, name):
        self._people_to_stomp_on.append(name)
        print('Ouch!')

    def revenge(self):
        print('10 years later...')
        for person in self._people_to_stomp_on:
            print('%s stomps on %s' % (self.name, person))
```

Here is what you'll see in an interactive session:

```
>>> joe = UnforgivingElephant('Joe')
>>> joe.get_slapped_by('Tarek')
Ouch!
>>> joe.get_slapped_by('Bill')
Ouch!
>>> joe.revenge()
10 years later...
Joe stomps on Tarek
Joe stomps on Bill
```

20.2.2 2.2. Functions and methods

Functions and methods should be in lowercase with underscores. This rule was not always true in the old standard library modules. Python 3 did a lot of reorganization of the standard library, so most of the functions and methods have a consistent letter case. Still, for some modules such as `threading`, you can access the old function names that used `mixedCase` (for example, `currentThread`). This was left to allow easier backward compatibility, but if you don't need to run your code in older versions of Python, then you should avoid using these old names.

This way of writing methods was common before the lowercase norm became the standard, and some frameworks, such as Zope and Twisted, are also still using `mixedCase` for methods. The community of developers working with them is still quite large. So the choice between `mixedCase` and lowercase with an underscore is definitely driven by the libraries you are using.

As a Zope developer, it is not easy to stay consistent because building an application that mixes pure Python modules and modules that import Zope code is difficult. In Zope, some classes mix both conventions because the code base is still evolving and Zope developers try to adopt the common conventions accepted by so many.

A decent practice in this kind of library environment is to use `mixedCase` only for elements that are exposed in the framework, and to keep the rest of the code in PEP 8 style.

It is also worth noting that developers of the Twisted project took a completely different approach to this problem.

The Twisted project, same as Zope, predates the PEP 8 document. It was started when there were no official guidelines for Python code style, so it had its own guidelines. Stylistic rules about the indentation, docstrings, line lengths, and so on could be easily adopted. On the other hand, updating all the code to match naming conventions from PEP 8 would result in completely broken backward compatibility. And doing that for such a large project as Twisted is infeasible. So Twisted adopted as much of PEP 8 as possible and left things such as `mixedCase` for variables, functions, and methods as part of its own coding standard. And this is completely compatible with the PEP 8 suggestion because it exactly says that consistency within a project is more important than consistency with PEP 8's style guide.

2.2.1 The private controversy

For private methods and functions, we usually use a single leading underscore. This is only a naming convention and has no syntactical meaning. But it doesn't mean that leading underscores have no syntactical meaning at all. When a method has two leading underscores, it is renamed on the fly by the interpreter to prevent a name collision with a method from any subclass. This feature of Python is called **name mangling**.

So some people tend to use a double leading underscore for their private attributes to avoid name collision in the subclasses, for example:

```
class Base(object):
    def __secret(self):
        print("don't tell")
    def public(self):
        self.__secret()

class Derived(Base):
    def __secret(self):
        print("never ever")
```

From this you will see the following output:

```
>>> Base.__secret
Traceback (most recent call last):
    File "<input>", line 1, in <module>
AttributeError: type object 'Base' has no attribute '__secret'
>>> dir(Base)
['_Base__secret', ..., 'public']
>>> Base().public()
don't tell
>>> Derived().public()
don't tell
```

The original motivation for name mangling in Python was not to provide the same isolation primitive as a private keyword in C++ but to make sure that some base classes implicitly avoid collisions in subclasses, especially if they are intended to be used in multiple inheritance contexts (for example, as mixin classes). But using it for every attribute that isn't public obfuscates the code and makes it extremely hard to extend. This is not Pythonic at all.

For more information on this topic, an interesting thread occurred in the Python-Dev mailing list many years ago, where people argued on the utility of name mangling and its fate in the language. It can be found at

http://mail.python.org/pipermail/python-dev/2005-December/058555.html .

2.2.2. Special methods

Special methods (https://docs.python.org/3/reference/datamodel.html#special-method-names) start and end with a double underscore and form so-called protocols of the language. Some developers used to call them *dunder* methods as a portmanteau of double underscore. They are used for operator overloading, container definitions, and so on. For the sake of readability, they should be gathered at the beginning of class definitions, as shown in the following code:

```
class WeirdInt(int):

    def __add__(self, other):
        return int.__add__(self, other) + 1

    def __repr__(self):
        return '<weirdo %d>' % self

    # public API
    def do_this(self):
        print('this')

    def do_that(self):
        print('that')
```

No user-defined method should use this convention unless it explicitly has to implement one of the Python object protocols. So don't invent your own dunder methods such as this:

```
class BadHabits:
    def __my_method__(self):
        print('ok')
```

20.2.3 2.3. Arguments

Arguments are in lowercase, with underscores if needed. They follow the same naming rules as variables because arguments are simply local variables that get their value as function input values. In the following example, `text` and `separator` are arguments of `one_line()` function:

```
def one_line(text, separator=" "):
"""Convert possibly multiline text to single line"""
    return separator.join(text.split())
```

20.2.4 2.4. Properties

The names of properties are in lowercase, or in lowercase with underscores. Most of the time they represent an object's state, which can be a noun or an adjective, or a small phrase when needed. In the following code example, the `Container` class is a simple data structure that can return copies of its contents through `unique_items` and `ordered_items` properties:

```
class Container:
    _contents = []
    def append(self, item):
        self._contents.append(item)

    @property
    def unique_items(self):
        return set(self._contents)
    @property
    def ordered_items(self):
        return list(self._contents)
```

20.2.5 2.5. Classes

The names of classes are always in `CamelCase`, and may have a leading underscore when they are private to a module.

In object-oriented programming classes are used to encapsulate the application state. Attributes of objects are record of that state. Methods are used to modify that state, convert it into meaningful values or to produce side effects. This is why class names are often noun phrases and form a usage logic with the method names that are verb phrases. The following code example contains a `Document` class definition with a single `save()` method:

```
class Document():
    file_name: str
    contents: str
    ...

    def save(self):
        with open(self.file_name, 'w') as file:
            file.write(self.contents)
```

Class instances often use the same noun phrases as the document but spelled with lowercase. So, actual `Document` class usage could be as follows:

```
new_document = Document()
new_document.save()
```

20.2.6 2.6. Modules and packages

Besides the special module `__init__`, the module names are in lowercase. The following are some examples from the standard library:

- `os`
- `sys`
- `shutil`

The Python standard library does not use underscores for module names to separate words but they are used commonly in many other projects. When the module is private to the package, a leading underscore is added. Compiled C or C++ modules are usually named with an underscore and imported in pure Python modules. Package names follow the same rules, since they act more like structured modules.

20.3 3. Naming guide

A common set of naming rules can be applied on variables, methods, functions, and properties. The names of classes and modules play a very important role in namespace construction and greatly affect code readability. This section contains a miniguide that will help you to define meaningful and readable names for your code elements.

20.3.1 3.1. Using the has/is prefixes for Boolean elements

When an element holds a Boolean value you can mark it with `is` and/or `has` syntax to make the variable more readable. In the following example, is_connected and `has_cache` are such identifiers that hold Boolean states of the `DB` class instances:

```
class DB:
is_connected = False
has_cache = False
```

20.3.2 3.2. Using plurals for variables that are collections

When an element is holding a sequence, it is a good idea to use a plural form. You can also do the same for various mapping variables and properties. In following example, `connected_users` and `tables` are class attributes that hold multiple values:

```
class DB:
    connected_users = ['Tarek']
    tables = {'Customer':['id', 'first_name', 'last_name']}
```

20.3.3 3.3. Using explicit names for dictionaries

When a variable holds a mapping, you should use an explicit name when possible. For example, if a `dict` holds a person's address, it can be named `persons_addresses`:

```
persons_addresses = {'Bill': '6565 Monty Road', 'Pamela': '45 Python street'}
```

20.3.4 3.4. Avoid generic names and redundancy

You should generally avoid using explicit type names `list`, `dict`, and `set` as parts of variable names even for local variables. Python now offers function and variable annotations and a typing hierarchy that allows you to easily mark an expected type for a given variable so there is no longer a need to describe object types in their names. It makes the code hard to read, understand, and use. Using a built-in name has to be avoided as well to avoid shadowing it in the current namespace. Generic verbs should also be avoided, unless they have a meaning in the namespace.

Instead, domain-specific terms should be used as follows:

```
def compute(data): # too generic
    for element in data:
        yield element ** 2

def squares(numbers): # better
    for number in numbers:
        yield number ** 2
```

There is also the following list of prefixes and suffixes that, despite being very common in programming, should be, in fact, avoided in function and class names:

- Manager
- Object
- Do, handle, or perform

The reason for this is that they are vague, ambiguous, and do not add any value to the actual name. Jeff Atwood, the co-founder of Discourse and Stack Overflow, has a very good article on this topic and it can be found on his blog at http://blog.codinghorror.com/i-shall-call-it-somethingmanager/

There is also a list of package names that should be avoided. Everything that does not give any clue about its content can do a lot of harm to the project in the long term. Names such as `misc`, `tools`, `utils`, `common`, or core have a very strong tendency to become endless bags of various unrelated code pieces of very poor quality that seem to grow in size exponentially. In most cases, the existence of such a module is a sign of laziness or lack of enough design efforts. Enthusiasts of such module names can simply forestall the future and rename them to `trash` or `dumpster` because this is exactly how their teammates will eventually treat such modules.

In most cases, it is almost always better to have more small modules even with very little content but with names that reflect well what is inside. To be honest, there is nothing inherently wrong with names such as `utils` and `common` and there is a possibility to use them responsibly. But reality shows that in many cases they instead become a stub for dangerous structural antipatterns that proliferate very fast. And if you don't act fast enough,

you may not be able get rid of them ever. So the best approach is simply to avoid such risky organizational patterns and nip them in the bud.

20.3.5 3.5. Avoiding existing names

It is a bad practice to use names that shadow other names that already exist in the same context. It makes code reading and debugging very confusing. Always try to define original names, even if they are local to the context. If you eventually have to reuse existing names or keywords, use a trailing underscore to avoid name collision, for example:

```
def xapian_query(terms, or_=True):
    """if or_ is true, terms are combined with the OR clause"""
    ...
```

Note that the `class` keyword is often replaced by `klass` or `cls`:

```
def factory(klass, *args, **kwargs):
    return klass(*args, **kwargs)
```

20.4 4. Best practices for arguments

The signatures of functions and methods are the guardians of code integrity. They drive its usage and build its APIs. Besides the naming rules that we have discussed previously, special care has to be taken for arguments. This can be done through the following three simple rules:

- Build arguments by iterative design.
- Trust the arguments and your tests.
- Use `*args` and `**kwargs` magic arguments carefully.

20.4.1 4.1. Building arguments by iterative design

Having a fixed and well-defined list of arguments for each function makes the code more robust. But this can't be done in the first version, so arguments have to be built by iterative design. They should reflect the precise use cases the element was created for, and evolve accordingly.

Consider the following example of the first versions of some `Service` class:

```
class Service: # version 1

    def _query(self, query, type):
        print('done')

    def execute(self, query):
        self._query(query, 'EXECUTE')
```

If you want to extend the signature of the `execute()` method with new arguments in a way that preserves backward compatibility, you should provide default values for these arguments as follows:

```
class Service(object): # version 2

    def _query(self, query, type, logger):
        logger('done')

    def execute(self, query, logger=logging.info):
        self._query(query, 'EXECUTE', logger)
```

The following example from an interactive session presents two styles of calling the `execute()` method of the updated `Service` class:

```
>>> Service().execute('my query')
# old-style call
>>> Service().execute('my query', logging.warning)
WARNING:root:done
```

20.4.2 4.2. Trusting the arguments and your tests

Given the dynamic typing nature of Python, some developers use assertions at the top of their functions and methods to make sure the arguments have proper content, for example:

```
def divide(dividend, divisor):
    assert isinstance(dividend, (int, float))
    assert isinstance(divisor, (int, float))
    return dividend / divisor
```

This is often done by developers who are used to static typing and feel that something is missing in Python.

This way of checking arguments is a part of the **Design by Contract (DbC)** programming style, where preconditions are checked before the code is actually run.

The two main problems in this approach are as follows:

- DbC's code explains how it should be used, making it less readable
- This can make it slower, since the assertions are made on each call

The latter can be avoided with the `-O` option of the Python interpreter. In that case, all assertions are removed from the code before the byte code is created, so that the checking is lost.

In any case, assertions have to be done carefully, and should not be used to bend Python to a statically typed language. The only use case for this is to protect the code from being called nonsensically. If you really want to have some kind of static typing in Python, you should definitely try MyPy or a similar static type checker that does not affect your code runtime and allows you to provide type definitions in a more readable form as function and variable annotations.

20.4.3 4.3. Using *args and **kwargs magic arguments carefully

The `*args` and `**kwargs` arguments can break the robustness of a function or method. They make the signature fuzzy, and the code often starts to become a small argument parser where it should not, for example:

```
def fuzzy_thing(**kwargs):
    if 'do_this' in kwargs:
        print('ok i did this')

    if 'do_that' in kwargs:
        print('that is done')

    print('ok')

>>> fuzzy_thing(do_this=1)
ok i did this
ok
>>> fuzzy_thing(do_that=1)
that is done
ok
>>> fuzzy_thing(what_about_that=1)
ok
```

If the argument list gets long and complex, it is tempting to add magic arguments. But this is more a sign of a weak function or method that should be broken into pieces or refactored.

When `*args` is used to deal with a sequence of elements that are treated the same way in the function, asking for a unique container argument such as an iterator is better, for example:

```
def sum(*args): # okay
    total = 0
    for arg in args:
        total += arg

    return total


def sum(sequence): # better!
    total = 0
    for arg in sequence:
        total += arg

    return total
```

For `**kwargs`, the same rule applies. It is better to fix the named arguments to make the method's signature meaningful, for example:

```
def make_sentence(**kwargs):
    noun = kwargs.get('noun', 'Bill')
    verb = kwargs.get('verb', 'is')
    adjective = kwargs.get('adjective', 'happy')
    return f'{noun} {verb} {adjective}'

def make_sentence(noun='Bill', verb='is', adjective='happy'):
    return f'{noun} {verb} {adjective}'
```

Another interesting approach is to create a container class that groups several related arguments to provide an execution context. This structure differs from `*args` or `**kwargs` because it can provide internals that work over the values, and can evolve independently. The code that uses it as an argument will not have to deal with its internals.

For instance, a web request passed on to a function is often represented by an instance of a class. This class is in charge of holding the data passed by the web server, as shown in the following code:

```
def log_request(request): # version 1
    print(request.get('HTTP_REFERER', 'No referer'))

def log_request(request): # version 2
    print(request.get('HTTP_REFERER', 'No referer'))
    print(request.get('HTTP_HOST', 'No host'))
```

Magic arguments cannot be avoided sometimes, especially in metaprogramming. For instance, they are indispensable in the creation of decorators that work on functions with any kind of signature.

20.5 5. Class names

The name of a class has to be concise, precise, and descriptive. A common practice is to use a suffix that informs about its type or nature, for example:

- SQLEngine
- MimeTypes
- StringWidget
- TestCase

For base or abstract classes, a Base or Abstract prefix can be used as follows:

- BaseCookie
- AbstractFormatter

The most important thing is to be consistent with the class attributes. For example, try to avoid redundancy between the class and its attributes' names as follows:

```
>>> SMTP.smtp_send()  # redundant information in the namespace
>>> SMTP.send()  # more readable and mnemonic
```

20.6 6. Modules and packages

The module and package names inform about the purpose of their content. The names are short, in lowercase, and usually without underscores, for example:

- `sqlite`
- `postgres`
- `sha1`

They are often suffixed with `lib` if they are implementing a protocol, as in the following:

```
import smtplib
import urllib
import telnetlib
```

When choosing a name for a module, always consider its content and limit the amount of redundancy within the whole namespace, for example:

```
from widgets.stringwidgets import TextWidget # bad
from widgets.strings import TextWidget # better
```

When a module is getting complex and contains a lot of classes, it is a good practice to create a package and split the module's elements into other modules.

The `__init__` module can also be used to put back some common APIs at the top level of the package. This approach allows you to organize the code into smaller components without reducing the ease of use.

20.7 7. Useful tools

Common conventions and practices used in a software project should always be documented. But having proper documentation for guidelines is often not enough to enforce that these guidelines are actually followed. Fortunately, you can use automated tools that can check sources of your code and verify if it meets specific naming conventions and style guidelines.

The following are a few popular tools:

- `pylint`: This is a very flexible source code analyzer
- `pycodestyle` and `flake8`: This is a small code style checker and a wrapper that adds to it some more useful features, such as static analysis and complexity measurement

20.7.1 7.1. Pylint

Besides some quality assurance metrics, Pylint allows for checking of whether a given source code is following a naming convention. Its default settings correspond to PEP 8 and a Pylint script provides a shell report output.

To install Pylint, you can use `pip` as follows:

```
$ pip install pylint
```

After this step, the command is available and can be run against a module, or several modules using wildcards. Let's try it on Buildout's `bootstrap.py` script as follows:

```
$ wget -O bootstrap.py https://bootstrap.pypa.io/bootstrap-buildout.py -q
$ pylint bootstrap.py
No config file found, using default configuration
************* Module bootstrap
C: 76, 0: Unnecessary parens after 'print' keyword (superfluous-parens)
C: 31, 0: Invalid constant name "tmpeggs" (invalid-name)
C: 33, 0: Invalid constant name "usage" (invalid-name)
C: 45, 0: Invalid constant name "parser" (invalid-name)
C: 74, 0: Invalid constant name "options" (invalid-name)
C: 74, 9: Invalid constant name "args" (invalid-name)
C: 84, 4: Import "from urllib.request import urlopen" should be placed at
the top of the module (wrong-import-position)
...
Global evaluation
-----------------
Your code has been rated at 6.12/10
```

Real Pylint's output is a bit longer and here it has been truncated for the sake of brevity.

Remember that Pylint can often give you false positive warnings that decrease the overall quality rating. For instance, an import statement that is not used by the code of the module itself is perfectly fine in some cases (for example, building top-level `__init__` modules in a package). Always treat Pylint's output as a hint and not an oracle.

Making calls to libraries that are using mixedCase for methods can also lower your rating. In any case, the global evaluation of your code score is not that important. Pylint is just a tool that points you to places where there is the possibility for improvements.

It is always recommended to do some tuning of Pylint. In order to do so you need to create a `.pylinrc` configuration file in your project's root directory. You can do that using the following `-generate-rcfile` option of the `pylint` command:

```
$ pylint --generate-rcfile > .pylintrc
```

This configuration file is self-documenting (every possible option is described with comment) and should already contain every available Pylint configuration option.

Besides checking for compliance with some arbitrary coding standards, Pylint can also give additional information about the overall code quality, such as:

- Code duplication metrics
- Unused variables and imports
- Missing function, method, or class docstrings
- Too long function signatures

The list of available checks that are enabled by default is very long. It is important to know that some of the rules are very arbitrary and cannot always be easily applied to every code base. Remember that consistency is always more valuable than compliance to some arbitrary rules. Fortunately, Pylint is very tunable, so if your team uses some naming and coding conventions that are different from the ones assumed by default, you can easily configure Pylint to check for consistency with your own conventions.

20.7.2 7.2. pycodestyle and flake8

`pycodestyle` (formerly `pep8`) is a tool that has only one purpose; it provides only style checking against code conventions defined in PEP 8. This is the main difference from Pylint that has many more additional features. This is the best option for programmers that are interested in automated code style checking only for the PEP 8 standard, without any additional tool configuration, as in Pylint's case.

`pycodestyle` can be installed with pip as follows:

```
$ pip install pycodestyle
```

When run on the Buildout's `bootstrap.py` script, it will give the following short list of code style violations:

```
$ wget -O bootstrap.py https://bootstrap.pypa.io/bootstrap-buildout.py -q
$ pycodestyle bootstrap.py
bootstrap.py:118:1: E402 module level import not at top of file
bootstrap.py:119:1: E402 module level import not at top of file
bootstrap.py:190:1: E402 module level import not at top of file
bootstrap.py:200:1: E402 module level import not at top of file
```

The main difference from Pylint's output is its length. `pycodestyle` concentrates only on style, so it does not provide any other warnings, such as unused variables, too long function names, or missing docstrings. It also does not give a rating. And it really makes sense because there is no such thing as partial consistency or partial conformance. Any, even the slightest, violation of style guidelines makes the code immediately inconsistent.

The code of `pycodestyle` is simpler than Pylint's and its output is easier to parse, so it may be a better choice if you want to make your code style verification part of a continuous integration process. If you are missing some static analysis features, there is the `flake8` package that is a wrapper on `pycodestyle` and a few other tools that are easily extendable and provide a more extensive suite of features. These include the following:

- McCabe complexity measurement
- Static analysis via `pyflakes`
- Disabling whole files or single lines using comments

Part II

Data structures and algorithms

Part III

Code quality

CHAPTER

TWENTYONE

UNIT TESTING AND REFACTORING

The ideas explored in this chapter are fundamental pillars because of their importance towards our ultimate goal: to write better and more maintainable software.

Unit tests (and any form of automatic tests, for that matter) are critical to software maintainability, and therefore are something that cannot be missing from any quality project. It is for that reason that this chapter is dedicated exclusively to aspects of automated testing as a key strategy, to safely modify the code, and iterate over it, in incrementally better versions.

21.1 1. Design principles and unit testing

In this section, are first going to take a look at unit testing from a conceptual point of view. We will revisit some of the software engineering principles we discussed in the previous to get an idea of how this is related to clean code.

After that, we will discuss in more detail how to put these concepts into practice (at the code level), and what frameworks and tools we can make use of.

First we quickly define what unit testing is about. Unit tests are parts of the code in charge of validating other parts of the code. Normally, anyone would be tempted to say that unit tests, validate the "core" of the application, but such definition regards unit tests to a secondary place, which is not the way they are thought of. Unit tests are core, and a critical component of the software and they should be treated with the same considerations as the business logic.

A unit tests is a piece of code that imports parts of the code with the business logic, and exercises its logic, asserting several scenarios with the idea to guarantee certain conditions. There are some traits that unit tests must have, such as:

- Isolation: unit test should be completely independent from any other external agent, and they have to focus only on the business logic. For this reason, they do not connect to a database, they don't perform HTTP requests, etc. Isolation also means that the tests are independent among themselves: they must be able to run in any order, without depending on any previous state.
- Performance: unit tests must run quickly. They are intended to be run multiple times, repeatedly.
- Self-validating: The execution of a unit tests determines its result. There should be no extra step required to interpret the unit test (much less manual).

More concretely, in Python this means that we will have new files where we are going to place our unit tests, and they are going to be called by some tool. Inside this files we program the tests themselves. Afterwards, a tool will collect our unit tests and run them, giving a result.

This last part is what self-validation actually means. When the tool calls our files, a Python process will be launched, and our tests will be running on it. If the tests fail, the process will have exited with an error code (in a Unix environment, this can be any number different than 0). The standard is that the tool runs the test, and prints a dot (`.`) for every successful test, an `F` if the test failed (the condition of the test was not satisfied), and an `E` if there was an exception.

21.1.1 1.1. A note about other forms of automated testing

Unit tests are intended to verify very small units, for example a function, or a method. We want from our unit tests to reach a very detailed level of granularity, testing as much code as possible. To test a class we would not want to use a unit tests, but rather a test suite, which is a collection of unit tests. Each one of them will be testing something more specific, like a method of that class.

This is not the only form of unit tests, and it cannot catch every possible error. There are also acceptance and integration tests, both out of the scope.

In an integration test, we will want to test multiple components at once. In this case we want to validate if collectively, they work as expected. In this case is acceptable (more than that, desirable) to have side-effects, and to forget about isolation, meaning that we will want to issue HTTP requests, connect to databases, and so on.

An acceptance test is an automated form of testing that tries to validate the system from the perspective of an user, typically executing use cases.

These two last forms of testing lose another nice trait with respect of unit tests: velocity. As you can imagine, they will take more time to run, therefore they will be run less frequently.

In a good development environment, the programmer will have the entire test suite, and will run unit tests all the time, repeatedly, while he or she is making changes to the code, iterating, refactoring, and so on. Once the changes are ready, and the pull request is open, the continuous integration service will run the build for that branch, where the unit tests will run, as long as the integration or acceptance tests that might exist. Needless to say, the status of the build should be successful (green) before merging, but the important part is the difference between the kind of tests: we want to run unit tests all the time, and less frequently those test that take longer. For this reason, we want to have a lot of small unit tests, and a few automated tests, strategically designed to cover as much as possible of where the unit tests could not reach (the database, for instance).

Finally, a word to the wise. Remember we encourage pragmatism. Besides these definitions give, and the points made about unit tests in the beginning of the section, the reader has to keep in mind that the best solution according to your criteria and context, should predominate. Nobody knows your system better than you. Which means, if for some reason you have to write an unit tests that needs to launch a Docker container to test against a database, go for it. Practicality beats purity.

21.1.2 1.2. Unit testing and agile software development

In modern software development, we want to deliver value constantly, and as quickly as possible. The rationale behind these goals is that the earlier we get feedback, the less the impact, and the easier it will be to change. These are no new ideas at all; some of them resemble manufacturing principles from decades ago, and others (such as the idea of getting feedback from stakeholders as soon as possible and iterating upon it) you can find in essays such as The Cathedral and the Bazaar (abbreviated as CatB).

Therefore, we want to be able to respond effectively to changes, and for that, the software we write will have to change. Like we mentioned in the previous chapters, we want our software to be adaptable, flexible, and extensible.

The code alone (regardless of how well written and designed it is) cannot guarantee us that it's flexible enough to be changed. Let's say we design a piece of software following the SOLID principles, and in one part we actually have a set of components that comply with the open/closed principle, meaning that we can easily extend them without affecting too much existing code. Assume further that the code is written in a way that favors refactoring, so we could change it as required. What's to say that when we make these changes, we aren't introducing any bugs? How do we know that existing functionality is preserved? Would you feel confident enough releasing that to your users? Will they believe that the new version works just as expected?

The answer to all of these questions is that we can't be sure unless we have a formal proof of it. And unit tests are just that, formal proof that the program works according to the specification.

Unit (or automated) tests, therefore, work as a safety net that gives us the confidence to work on our code. Armed with these tools, we can efficiently work on our code, and therefore this is what ultimately determines the velocity (or capacity) of the team working on the software product. The better the tests, the more likely it is we can deliver value quickly without being stopped by bugs every now and then.

21.1.3 1.3. Unit testing and software design

This is the other face of the coin when it comes to the relationship between the main code and unit testing. Besides the pragmatic reasons explored in the previous section, it comes down to the fact that good software is testable software. **Testability** (the quality attribute that determines how easy to test software is) is not just a nice to have, but a driver for clean code.

Unit tests aren't just something complementary to the main code base, but rather something that has a direct impact and real influence on how the code is written. There are many levels of this, from the very beginning when we realize that the moment we want to add unit tests for some parts of our code, we have to change it (resulting in a better version of it), to its ultimate expression (explored near the end of this chapter) when the entire code (the design) is driven by the way it's going to be tested via **test-driven design**.

Starting off with a simple example, we will show you a small use case in which tests (and the need to test our code) lead to improvements in the way our code ends up being written.

In the following example, we will simulate a process that requires sending metrics to an external system about the results obtained at each particular task (as always, details won't make any difference as long as we focus on the code). We have a `Process` object that represents some task on the domain problem, and it uses a `metrics` client (an external dependency and therefore something we don't control) to send the actual metrics to the external entity (that this could be sending data to `syslog`, or `statsd`, for instance):

```
class MetricsClient:
"""3rd-party metrics client"""
    def send(self, metric_name, metric_value):
        if not isinstance(metric_name, str):
            raise TypeError("expected type str for metric_name")

        if not isinstance(metric_value, str):
            raise TypeError("expected type str for metric_value")

        logger.info(f"sending {metric_name} = {metric_value}")

class Process:
    def __init__(self):
        self.client = MetricsClient() # A 3rd-party metrics client
    def process_iterations(self, n_iterations):
        for i in range(n_iterations):
            result = self.run_process()
            self.client.send(f"iteration.{i}", result)
```

In the simulated version of the third-party client, we put the requirement that the parameters provided must be of type string. Therefore, if the result of the `run_process` method is not a string, we might expect it to fail, and indeed it does:

```
Traceback (most recent call last):
...
raise TypeError("expected type str for metric_value")
TypeError: expected type str for metric_value
```

Remember that this validation is out of our hands and we cannot change the code, so we must provide the method with parameters of the correct type before proceeding. But since this is a bug we detected, we first want to write a unit test to make sure it will not happen again. We do this to actually prove that we fixed the issue, and to protect against this bug in the future, regardless of how many times the code is refactored.

It would be possible to test the code as is by mocking the client of the `Process` object (we will see how to do so in the section about mock objects, when we explore the tools for unit testing), but doing so runs more code than is needed (notice how the part we want to test is nested into the code). Moreover, it's good that the method is relatively small, because if it weren't, the test would have to run even more undesired parts that we might also need to mock. This is another example of good design (small, cohesive functions or methods), that relates to testability.

Finally, we decide not to go to much trouble and test just the part that we need to, so instead of interacting with the client directly on the main method, we delegate to a wrapper method, and the new class looks like this:

```
class WrappedClient:
    def __init__(self):
        self.client = MetricsClient()
    def send(self, metric_name, metric_value):
        return self.client.send(str(metric_name), str(metric_value))

class Process:
    def __init__(self):
        self.client = WrappedClient()
        ... # rest of the code remains unchanged
```

In this case, we opted for creating our own version of the client for metrics, that is, a wrapper around the third-party library one we used to have. To do this, we place a class that (with the same interface) will make the conversion of the types accordingly.

This way of using composition resembles the adapter design pattern (we'll explore design patterns in the next chapter, so, for now, it's just an informative message), and since this is a new object in our domain, it can have its respective unit tests. Having this object will make things simpler to test, but more importantly, now that we look at it, we realize that this is probably the way the code should have been written in the first place. Trying to write a unit test for our code made us realize that we were missing an important abstraction entirely!

Now that we have separated the method as it should be, let's write the actual unit test for it. The details about the unittest module used in this example will be explored in more detail in the part of the chapter where we explore testing tools and libraries, but for now reading the code will give us a first impression on how to test it, and it will make the previous concepts a little less abstract:

```
import unittest
from unittest.mock import Mock


class TestWrappedClient(unittest.TestCase):
    def test_send_converts_types(self):
        wrapped_client = WrappedClient()
        wrapped_client.client = Mock()
        wrapped_client.send("value", 1)
        wrapped_client.client.send.assert_called_with("value", "1")
```

`Mock` is a type that's available in the `unittest.mock` module, which is a quite convenient object to ask about all sort of things. For example, in this case, we're using it in place of the third-party library (mocked into the boundaries of the system, as commented on the next section) to check that it's called as expected (and once again, we're not testing the library itself, only that it is called correctly). Notice how we run a call like the one our `Process` object, but we expect the parameters to be converted to strings.

21.1.4 1.4. Defining the boundaries of what to test

Testing requires effort. And if we are not careful when deciding what to test, we will never end testing, hence wasting a lot of effort without achieving much.

We should scope the testing to the boundaries of our code. If we don't, we would have to also test the dependencies (external/third-party libraries or modules) or our code, and then their respective dependencies, and so on and so forth in a never-ending journey. It's not our responsibility to test dependencies, so we can assume that these projects have tests of their own. It would be enough just to test that the correct calls to external dependencies are done with the correct parameters (and that might even be an acceptable use of patching), but we shouldn't put more effort in than that.

This is another instance where good software design pays off. If we have been careful in our design, and clearly defined the boundaries of our system (that is, we designed towards interfaces, instead of concrete implementations that will change, hence inverting the dependencies over external components to reduce temporal coupling), then it will be much more easier to mock these interfaces when writing unit tests.

In good unit testing, we want to patch on the boundaries of our system and focus on the core functionality to be exercised. We don't test external libraries (third-party tools installed via `pip`, for instance), but instead, we check that they are called correctly. When we explore mock objects later on in this chapter, we will review techniques and tools for performing these types of assertion.

21.2 2. Frameworks and tools for testing

There are a lot of tools we can use for writing out unit tests, all of them with pros and cons and serving different purposes. But among all of them, there are two that will most likely cover almost every scenario, and therefore we limit this section to just them.

Along with testing frameworks and test running libraries, it's often common to find projects that configure code coverage, which they use as a quality metric. Since coverage (when used as a metric) is misleading, after seeing how to create unit tests we'll discuss why it's not to be taken lightly.

21.2.1 2.1. Frameworks and libraries for unit testing

In this section, we will discuss two frameworks for writing and running unit tests. The first one, `unittest`, is available in the standard library of Python, while the second one, `pytest`, has to be installed externally via `pip`.

When it comes to covering testing scenarios for our code, `unittest` alone will most likely suffice, since it has plenty of helpers. However, for more complex systems on which we have multiple dependencies, connections to external systems, and probably the need to patch objects, and define fixtures parameterize test cases, then `pytest` looks like a more complete option.

We will use a small program as an example to show you how could it be tested using both options which in the end will help us to get a better picture of how the two of them compare.

The example demonstrating testing tools is a simplified version of a version control tool that supports code reviews in merge requests. We will start with the following criteria:

- A merge request is rejected if at least one person disagrees with the changes.
- If nobody has disagreed, and the merge request is good for at least two other developers, it's approved.
- In any other case, its status is pending.

And here is what the code might look like:

```
from enum import Enum
class MergeRequestStatus(Enum):
    APPROVED = "approved"
    REJECTED = "rejected"
    PENDING = "pending"

class MergeRequest:
    def __init__(self):
        self._context = {
        "upvotes": set(),
        "downvotes": set(),
        }

    @property
    def status(self):
        if self._context["downvotes"]:
            return MergeRequestStatus.REJECTED
        elif len(self._context["upvotes"]) >= 2:
            return MergeRequestStatus.APPROVED
        return MergeRequestStatus.PENDING

    def upvote(self, by_user):
```

(continues on next page)

(continued from previous page)

```
        self._context["downvotes"].discard(by_user)
        self._context["upvotes"].add(by_user)

    def downvote(self, by_user):
        self._context["upvotes"].discard(by_user)
        self._context["downvotes"].add(by_user)
```

2.1.1 unittest

The `unittest` module is a great option with which to start writing unit tests because it provides a rich API to write all kinds of testing conditions, and since it's available in the standard library, it's quite versatile and convenient.

The `unittest` module is based on the concepts of `JUnit` (from Java), which in turn is also based on the original ideas of unit testing that come from Smalltalk, so it's object-oriented in nature. For this reason, tests are written through objects, where the checks are verified by methods, and it's common to group tests by scenarios in classes.

To start writing unit tests, we have to create a test class that inherits from `unittest.TestCase`, and define the conditions we want to stress on its methods. These methods should start with `test_*`, and can internally use any of the methods inherited from `unittest.TestCase` to check conditions that must hold true.

Some examples of conditions we might want to verify for our case are as follows:

```
class TestMergeRequestStatus(unittest.TestCase):
    def test_simple_rejected(self):
        merge_request = MergeRequest()
        merge_request.downvote("maintainer")
        self.assertEqual(merge_request.status, MergeRequestStatus.REJECTED)

    def test_just_created_is_pending(self):
        self.assertEqual(MergeRequest().status, MergeRequestStatus.PENDING)

    def test_pending_awaiting_review(self):
        merge_request = MergeRequest()
        merge_request.upvote("core-dev")
        self.assertEqual(merge_request.status, MergeRequestStatus.PENDING)

    def test_approved(self):
        merge_request = MergeRequest()
        merge_request.upvote("dev1")
        merge_request.upvote("dev2")
        self.assertEqual(merge_request.status, MergeRequestStatus.APPROVED)
```

The API for unit testing provides many useful methods for comparison, the most common one being `assertEquals(<actual>, <expected>[, message])`, which can be used to compare the result of the operation against the value we were expecting, optionally using a message that will be shown in the case of an error.

Another useful testing method allows us to check whether a certain exception was raised or not. When something exceptional happens, we raise an exception in our code to prevent continuous processing under the wrong assumptions, and also to inform the caller that something is wrong with the call as it was performed. This is the part of the logic that ought to be tested, and that's what this method is for.

Imagine that we are now extending our logic a little bit further to allow users to close their merge requests, and once this happens, we don't want any more votes to take place (it wouldn't make sense to evaluate a merge request once this was already closed). To prevent this from happening, we extend our code and we raise an exception on the unfortunate event when someone tries to cast a vote on a closed merge request.

After adding two new statuses (`OPEN` and `CLOSED`), and a new `close()` method, we modify the previous methods for the voting to handle this check first:

```
class MergeRequest:
    def __init__(self):
        self._context = {
        "upvotes": set(),
        "downvotes": set(),
        }
        self._status = MergeRequestStatus.OPEN
    def close(self):
        self._status = MergeRequestStatus.CLOSED
        ...
    def _cannot_vote_if_closed(self):
        if self._status == MergeRequestStatus.CLOSED:
            raise MergeRequestException("can't vote on a closed merge request")

    def upvote(self, by_user):
        self._cannot_vote_if_closed()
        self._context["downvotes"].discard(by_user)
        self._context["upvotes"].add(by_user)

    def downvote(self, by_user):
        self._cannot_vote_if_closed()
        self._context["upvotes"].discard(by_user)
        self._context["downvotes"].add(by_user)
```

Now, we want to check that this validation indeed works. For this, we're going to use the `asssertRaises` and `assertRaisesRegex` methods:

```
def test_cannot_upvote_on_closed_merge_request(self):
    self.merge_request.close()
    self.assertRaises(MergeRequestException, self.merge_request.upvote, "dev1")

def test_cannot_downvote_on_closed_merge_request(self):
    self.merge_request.close()
    self.assertRaisesRegex(MergeRequestException, "can't vote on a closed merge␣
↪request",
                           self.merge_request.downvote, "dev1")
```

The former will expect that the provided exception is raised when calling the callable in the second argument, with the arguments (`*args` and `**kwargs`) on the rest of the function, and if that's not the case it will fail, saying that the exception that was expected to be raised, wasn't. The latter does the same but it also checks that the exception that was raised, contains the message matching the regular expression that was provided as a parameter. Even if the exception is raised, but with a different message (not matching the regular expression), the test will fail.

Tip: Try to check for the error message, as not only will the exception, as an extra check, be more accurate and ensure that it is actually the exception we want that is being triggered, it will check whether another one of the same types got there by chance.

Now, we would like to test how the threshold acceptance for the merge request works, just by providing data samples of what the context looks like without needing the entire `MergeRequest` object. We want to test the part of the status property that is after the line that checks if it's closed, but independently.

The best way to achieve this is to separate that component into another class, use composition, and then move on to test this new abstraction with its own test suite:

```
class AcceptanceThreshold:
    def __init__(self, merge_request_context: dict) -> None:
        self._context = merge_request_context

    def status(self):
        if self._context["downvotes"]:
```

(continues on next page)

(continued from previous page)

```
            return MergeRequestStatus.REJECTED
        elif len(self._context["upvotes"]) >= 2:
            return MergeRequestStatus.APPROVED
        return MergeRequestStatus.PENDING

class MergeRequest:
    ...
    @property
    def status(self):
        if self._status == MergeRequestStatus.CLOSED:
            return self._status
        return AcceptanceThreshold(self._context).status()
```

With these changes, we can run the tests again and verify that they pass, meaning that this small refactor didn't break anything of the current functionality (unit tests ensure regression). With this, we can proceed with our goal to write tests that are specific to the new class:

```
class TestAcceptanceThreshold(unittest.TestCase):
    def setUp(self):
        self.fixture_data = (
            (
                {"downvotes": set(), "upvotes": set()},
                MergeRequestStatus.PENDING
            ),
            (
                {"downvotes": set(), "upvotes": {"dev1"}},
                MergeRequestStatus.PENDING,
            ),
            (
                {"downvotes": "dev1", "upvotes": set()},
                MergeRequestStatus.REJECTED
            ),
            (
                {"downvotes": set(), "upvotes": {"dev1", "dev2"}},
                MergeRequestStatus.APPROVED
            ),
        )
    def test_status_resolution(self):
        for context, expected in self.fixture_data:
            with self.subTest(context=context):
                status = AcceptanceThreshold(context).status()
                self.assertEqual(status, expected)
```

Here, in the `setUp()` method, we define the data fixture to be used throughout the tests. In this case, it's not actually needed, because we could have put it directly on the method, but if we expect to run some code before any test is executed, this is the place to write it, because this method is called once before every test is run.

By writing this new version of the code, the parameters under the code being tested are clearer and more compact, and at each case, it will report the results.

To simulate that we're running all of the parameters, the test iterates over all the data, and exercises the code with each instance. One interesting helper here is the use of `subTest`, which in this case we use to mark the test condition being called. If one of these iterations failed, `unittest` would report it with the corresponding value of the variables that were passed to the subTest (in this case, it was named context, but any series of keyword arguments would work just the same). For example, one error occurrence might look like this:

```
FAIL: (context={'downvotes': set(), 'upvotes': {'dev1', 'dev2'}})
----------------------------------------------------------------------

Traceback (most recent call last):
    File "" test_status_resolution
```

(continues on next page)

(continued from previous page)

```
        self.assertEqual(status, expected)
AssertionError: <MergeRequestStatus.APPROVED: 'approved'> !=
<MergeRequestStatus.REJECTED: 'rejected'>
```

Tip: If you choose to parameterize tests, try to provide the context of each instance of the parameters with as much information as possible to make debugging easier.

2.1.2. pytest

Pytest is a great testing framework, and can be installed via `pip`. A difference with respect to `unittest` is that, while it's still possible to classify test scenarios in classes and create object-oriented models of our tests, this is not actually mandatory, and it's possible to write unit tests with less boilerplate by just checking the conditions we want to verify with the `assert` statement.

By default, making comparisons with an `assert` statement will be enough for `pytest` to identify a unit test and report its result accordingly. More advanced uses such as those seen in the previous section are also possible, but they require using specific functions from the package.

A nice feature is that the command `pytest` will run all the tests that it can discover, even if they were written with `unittest`. This compatibility makes it easier to transition gradually.

2.1.2.1. Basic test cases with pytest

The conditions we tested in the previous section can be rewritten in simple functions with `pytest`.

Some examples with simple assertions are as follows:

```
def test_simple_rejected():
    merge_request = MergeRequest()
    merge_request.downvote("maintainer")
    assert merge_request.status == MergeRequestStatus.REJECTED

def test_just_created_is_pending():
    assert MergeRequest().status == MergeRequestStatus.PENDING

def test_pending_awaiting_review():
    merge_request = MergeRequest()
    merge_request.upvote("core-dev")
    assert merge_request.status == MergeRequestStatus.PENDING
```

Boolean equality comparisons don't require more than a simple `assert` statement, whereas other kinds of checks like the ones for the exceptions do require that we use some functions:

```
def test_invalid_types():
    merge_request = MergeRequest()
    pytest.raises(TypeError, merge_request.upvote, {"invalid-object"})

def test_cannot_vote_on_closed_merge_request():
    merge_request = MergeRequest()
    merge_request.close()
    pytest.raises(MergeRequestException, merge_request.upvote, "dev1")
    with pytest.raises(MergeRequestException, match="can't vote on a closed merge␣
↪request"):
        merge_request.downvote("dev1")
```

In this case, `pytest.raises` is the equivalent of `unittest.TestCase.assertRaises`, and it also accepts that it be called both as a method and as a context manager. If we want to check the message of the exception,

instead of a different method (like `assertRaisesRegex`), the same function has to be used, but as a context manager, and by providing the match parameter with the expression we would like to identify. `pytest` will also wrap the original exception into a custom one that can be expected (by checking some of its attributes such as `.value`, for instance) in case we want to check for more conditions, but this use of the function covers the vast majority of cases.

2.1.2.2. Parametrized tests

Running parametrized tests with `pytest` is better, not only because it provides a cleaner API, but also because each combination of the test with its parameters generates a new test case.

To work with this, we have to use the `pytest.mark.parametrize` decorator on our test. The first parameter of the decorator is a string indicating the names of the parameters to pass to the test function, and the second has to be iterable with the respective values for those parameters.

Notice how the body of the testing function is reduced to one line (after removing the internal for loop, and its nested context manager), and the data for each test case is correctly isolated from the body of the function, making it easier to extend and maintain:

```
@pytest.mark.parametrize("context, expected_status", [
    (
        {"downvotes": set(), "upvotes": set()},
        MergeRequestStatus.PENDING
    ),
    (
        {"downvotes": set(), "upvotes": {"dev1"}},
        MergeRequestStatus.PENDING,
    ),
    (
        {"downvotes": "dev1", "upvotes": set()},
        MergeRequestStatus.REJECTED
    ),
    (
        {"downvotes": set(), "upvotes": {"dev1", "dev2"}},
        MergeRequestStatus.APPROVED
    )
])
def test_acceptance_threshold_status_resolution(context, expected_status):
    assert AcceptanceThreshold(context).status() == expected_status
```

Use `@pytest.mark.parametrize` to eliminate repetition, keep the body of the test as cohesive as possible, and make the parameters (test inputs or scenarios) that the code must support explicitly.

2.1.2.3. Fixtures

One of the great things about `pytest` is how it facilitates creating reusable features so that we can feed our tests with data or objects in order to test more effectively and without repetition.

For example, we might want to create a `MergeRequest` object in a particular state, and use that object in multiple tests. We define our object as a fixture by creating a function and applying the `@pytest.fixture` decorator. The tests that want to use that fixture will have to have a parameter with the same name as the function that's defined, and `pytest` will make sure that it's provided:

```
@pytest.fixture
def rejected_mr():
    merge_request = MergeRequest()
    merge_request.downvote("dev1")
    merge_request.upvote("dev2")
    merge_request.upvote("dev3")
```

(continues on next page)

(continued from previous page)

```
    merge_request.downvote("dev4")
    return merge_request

def test_simple_rejected(rejected_mr):
    assert rejected_mr.status == MergeRequestStatus.REJECTED

def test_rejected_with_approvals(rejected_mr):
    rejected_mr.upvote("dev2")
    rejected_mr.upvote("dev3")
    assert rejected_mr.status == MergeRequestStatus.REJECTED

def test_rejected_to_pending(rejected_mr):
    rejected_mr.upvote("dev1")
    assert rejected_mr.status == MergeRequestStatus.PENDING

def test_rejected_to_approved(rejected_mr):
    rejected_mr.upvote("dev1")
    rejected_mr.upvote("dev2")
    assert rejected_mr.status == MergeRequestStatus.APPROVED
```

Remember that tests affect the main code as well, so the principles of clean code apply to them as well. In this case, the Don't Repeat Yourself (DRY) principle appears once again, and we can achieve it with the help of `pytest` fixtures.

Besides creating multiple objects or exposing data that will be used throughout the test suite, it's also possible to use them to set up some conditions, for example, to globally patch some functions that we don't want to be called, or when we want patch objects to be used instead.

21.2.2 2.2. Code coverage

Tests runners support coverage plugins (to be installed via `pip`) that will provide useful information about what lines in the code have been executed while the tests were running. This information is of great help so that we know which parts of the code need to be covered by tests, as well identifying improvements to be made (both in the production code and in the tests). One of the most widely used libraries for this is `coverage`.

While they are of great help (and we highly recommend that you use them and configure your project to run coverage in the CI when tests are run), they can also be misleading; particularly in Python, we can get a false impression if we don't pay close attention to the coverage report.

2.2.1. Setting up rest coverage

In the case of `pytest`, we have to install the `pytest-cov` package. Once installed, when the tests are run, we have to tell the `pytest` runner that `pytest-cov` will also run, and which package (or packages) should be covered (among other parameters and configurations).

This package supports multiple configurations, like different sorts of output formats, and it's easy to integrate it with any CI tool, but among all these features a highly recommended option is to set the flag that will tell us which lines haven't been covered by tests yet, because this is what's going to help us diagnose our code and allow us to start writing more tests.

To show you an example of what this would look like, use the following command:

```
pytest \
--cov-report term-missing \
--cov=coverage_1 \
test_coverage_1.py
```

This will produce an output similar to the following:

```
test_coverage_1.py ............... [100%]
----------- coverage: platform linux, python 3.6.5-final-0 -----------
Name
Stmts Miss Cover Missing
-------------------------------------------
coverage_1.py 38
1 97%
53
```

Here, it's telling us that there is a line that doesn't have unit tests so that we can take a look and see how to write a unit test for it. This is a common scenario where we realize that to cover those missing lines, we need to refactor the code by creating smaller methods. As a result, our code will look much better, as in the example we saw at the beginning of this chapter.

The problem lies in the inverse situation: can we trust the high coverage? Does it mean our code is correct? Unfortunately, having good test coverage is necessary but in sufficient condition for clean code. Not having tests for parts of the code is clearly something bad. Having tests is actually very good (and we can say this for the tests that do exist), and actually asserts real conditions that they are a guarantee of quality for that part of the code. However, we cannot say that is all that is required; despite having a high level of coverage, even more tests are required.

2.2.2. Caveats of test coverage

Python is interpreted and, at a very high-level, coverage tools take advantage of this to identify the lines that were interpreted (run) while the tests were running. It will then report this at the end. The fact that a line was interpreted does not mean that it was properly tested, and this is why we should be careful about reading the final coverage report and trusting what it says.

This is actually true for any language. The fact that a line was exercised does not mean at all that it was stressed with all its possible combinations. The fact that all branches run successfully with the provided data only means that the code supported that combination, but it doesn't tell us anything about any other possible combinations of parameters that would make the program crash.

Tip: Use coverage as a tool to find blind spots in the code, but not as a metric or target goal.

21.2.3 2.3. Mock objects

There are cases where our code is not the only thing that will be present in the context of our tests. After all, the systems we design and build have to do something real, and that usually means connecting to external services (databases, storage services, external APIs, cloud services, and so on). Because they need to have those side-effects, they're inevitable. As much as we abstract our code, program towards interfaces, and isolate code from external factors in order to minimize side-effects, they will be present in our tests, and we need an effective way to handle that.

`Mock` objects are one of the best tactics to defend against undesired side-effects. Our code might need to perform an HTTP request or send a notification email, but we surely don't want that to happen in our unit tests. Besides, unit tests should run quickly, as we want to run them quite often (all the time, actually), and this means we cannot afford latency. Therefore, real unit tests don't use any actual service: they don't connect to any database, they don't issue HTTP requests, and basically, they do nothing other than exercise the logic of the production code.

We need tests that do such things, but they aren't units. Integration tests are supposed to test functionality with a broader perspective, almost mimicking the behavior of a user. But they aren't fast. Because they connect to external systems and services, they take longer to run and are more expensive. In general, we would like to have lots of unit tests that run really quickly in order to run them all the time, and have integration tests run less often (for instance, on any new merge request).

While mock objects are useful, abusing their use ranges between a code smell or an anti- pattern is the first caveat we would like to mention before going into the details of it.

2.3.1. A fair warning about patching and mocks

We said before that unit tests help us write better code, because the moment we want to start testing parts of the code, we usually have to write them to be testable, which often means they are also cohesive, granular, and small. These are all good traits to have in a software component.

Another interesting gain is that testing will help us notice code smells in parts where we thought our code was correct. One of the main warnings that our code has code smells is whether we find ourselves trying to monkey-patch (or mock) a lot of different things just to cover a simple test case.

The `unittest` module provides a tool for patching our objects at `unittest.mock.patch`. Patching means that the original code (given by a string denoting its location at import time), will be replaced by something else, other than its original code, being the default a mock object. This replaces the code at run-time, and has the disadvantage that we are losing contact with the original code that was there in the first place, making our tests a little more shallow. It also carries performance considerations, because of the overhead that imposes modifying objects in the interpreter at run-time, and it's something that might end up update if we refactor our code and move things around.

Using monkey-patching or mocks in our tests might be acceptable, and by itself it doesn't represent an issue. On the other hand, abuse in monkey-patching is indeed a flag that something has to be improved in our code.

2.3.2. Using mock objects

In unit testing terminology, there are several types of object that fall into the category named **test doubles**. A test double is a type of object that will take the place of a real one in our test suite for different kinds of reasons (maybe we don't need the actual production code, but just a dummy object would work, or maybe we can't use it because it requires access to services or it has side-effects that we don't want in our unit tests, and so on).

There are different types of test double, such as dummy objects, stubs, spies, or mocks. Mocks are the most general type of object, and since they're quite flexible and versatile, they are appropriate for all cases without needing to go into much detail about the rest of them. It is for this reason that the standard library also includes an object of this kind, and it is common in most Python programs. That's the one we are going to be using here: `unittest.mock.Mock`.

A **mock** is a type of object created to a specification (usually resembling the object of a production class) and some configured responses (that is, we can tell the mock what it should return upon certain calls, and what its behavior should be). The Mock object will then record, as part of its internal status, how it was called (with what parameters, how many times, and so on), and we can use that information to verify the behavior of our application at a later stage.

In the case of Python, the Mock object that's available from the standard library provides a nice API to make all sorts of behavioral assertions, such as checking how many times the mock was called, with what parameters, and so on.

2.3.2.1 Types of mocks

The standard library provides `Mock` and `MagicMock` objects in the `unittest.mock` module. The former is a test double that can be configured to return any value and will keep track of the calls that were made to it. The latter does the same, but it also supports magic methods. This means that, if we have written idiomatic code that uses magic methods (and parts of the code we are testing will rely on that), it's likely that we will have to use a `MagicMock` instance instead of just a `Mock`.

Trying to use `Mock` when our code needs to call magic methods will result in an error. See the following code for an example of this:

```
class GitBranch:

    def __init__(self, commits: List[Dict]):
        self._commits = {c["id"]: c for c in commits}
```

(continues on next page)

(continued from previous page)

```
    def __getitem__(self, commit_id):
        return self._commits[commit_id]

    def __len__(self):
        return len(self._commits)

def author_by_id(commit_id, branch):
    return branch[commit_id]["author"]
```

We want to test this function; however, another test needs to call the `author_by_id` function. For some reason, since we're not testing that function, any value provided to that function (and returned) will be good:

```
def test_find_commit():
    branch = GitBranch([{"id": "123", "author": "dev1"}])
    assert author_by_id("123", branch) == "dev1"

def test_find_any():
    author = author_by_id("123", Mock()) is not None
    # ... rest of the tests..
```

As anticipated, this will not work:

```
def author_by_id(commit_id, branch):
    > return branch[commit_id]["author"]
    E TypeError: 'Mock' object is not subscriptable
```

Using `MagicMock` instead will work. We can even configure the magic method of this type of mock to return something we need in order to control the execution of our test:

```
def test_find_any():
    mbranch = MagicMock()
    mbranch.__getitem__.return_value = {"author": "test"}
    assert author_by_id("123", mbranch) == "test"
```

2.3.2.2. A use case for test doubles

To see a possible use of mocks, we need to add a new component to our application that will be in charge of notifying the merge request of the status of the build . When a build is finished, this object will be called with the ID of the merge request and the status of the build, and it will update the status of the merge request with this information by sending an HTTP POST request to a particular fixed endpoint:

```
from datetime import datetime
import requests
from constants import STATUS_ENDPOINT


class BuildStatus:
    @staticmethod
    def build_date() -> str:
        return datetime.utcnow().isoformat()

    @classmethod
    def notify(cls, merge_request_id, status):
        build_status = {
            "id": merge_request_id,
            "status": status,
            "built_at": cls.build_date(),
        }
        response = requests.post(STATUS_ENDPOINT, json=build_status)
```

(continues on next page)

(continued from previous page)

```
        response.raise_for_status()
        return response
```

This class has many side-effects, but one of them is an important external dependency which is hard to surmount. If we try to write a test over it without modifying anything, it will fail with a connection error as soon as it tries to perform the HTTP connection.

As a testing goal, we just want to make sure that the information is composed correctly, and that library requests are being called with the appropriate parameters. Since this is an external dependency, we don't test requests; just checking that it's called correctly will be enough.

Another problem we will face when trying to compare data being sent to the library is that the class is calculating the current timestamp, which is impossible to predict in a unit test. Patching `datetime` directly is not possible, because the module is written in C. There are some external libraries that can do that (`freezegun`, for example), but they come with a performance penalty, and for this example would be overkill. Therefore, we opt to wrapping the functionality we want in a static method that we will be able to patch.

Now that we have established the points that need to be replaced in the code, let's write the unit test:

```
from unittest import mock
from constants import STATUS_ENDPOINT
from mock_2 import BuildStatus

@mock.patch("mock_2.requests")
def test_build_notification_sent(mock_requests):
    build_date = "2018-01-01T00:00:01"
    with mock.patch("mock_2.BuildStatus.build_date", return_value=build_date):
        BuildStatus.notify(123, "OK")

    expected_payload = {"id": 123, "status": "OK", "built_at": build_date}
    mock_requests.post.assert_called_with(STATUS_ENDPOINT, json=expected_payload)
```

First, we use `mock.patch` as a decorator to replace the `requests` module. The result of this function will create a mock object that will be passed as a parameter to the test (named `mock_requests` in this example). Then, we use this function again, but this time as a context manager to change the return value of the method of the class that computes the date of the build, replacing the value with one we control, that we will use in the assertion.

Once we have all of this in place, we can call the class method with some parameters, and then we can use the `mock` object to check how it was called. In this case, we are using the method to see if `requests.post` was indeed called with the parameters as we wanted them to be composed.

This is a nice feature of mocks: not only do they put some boundaries around all external components (in this case to prevent actually sending some notifications or issuing HTTP requests), but they also provide a useful API to verify the calls and their parameters.

While, in this case, we were able to test the code by setting the respective mock objects in place, it's also true that we had to patch quite a lot in proportion to the total lines of code for the main functionality. There is no rule about the ratio of pure productive code being tested versus how many parts of that code we have to mock, but certainly, by using common sense, we can see that, if we had to patch quite a lot of things in the same parts, something is not clearly abstracted, and it looks like a code smell.

21.3 3. Refactoring

Refactoring is a critical activity in software maintenance, yet something that can't be done (at least correctly) without having unit tests. Every now and then, we need to support a new feature or use our software in unintended ways. We need to realize that the only way to accommodate such requirements is by first refactoring our code, make it more generic. Only then can we move forward.

Typically, when refactoring our code, we want to improve its structure and make it better, sometimes more generic, more readable, or more flexible. The challenge is to achieve these goals while at the same time preserving the exact same functionality it had prior to the modifications that were made. This means that, in the eyes of the clients of those components we're refactoring, it might as well be the case that nothing had happened at all.

This constraint of having to support the same functionalities as before but with a different version of the code implies that we need to run regression tests on code that was modified. The only cost-effective way of running regression tests is if those tests are automatic. The most cost-effective version of automatic tests is unit tests.

21.3.1 3.1. Evolving our code

In the previous example, we were able to separate out the side-effects from our code to make it testable by patching those parts of the code that depended on things we couldn't control on the unit test. This is a good approach since, after all, the `mock.patch` function comes in handy for these sorts of task and replaces the objects we tell it to, giving us back a `Mock` object.

The downside of that is that we have to provide the path of the object we are going to mock, including the module, as a string. This is a bit fragile, because if we refactor our code (let's say we rename the file or move it to some other location), all the places with the patch will have to be updated, or the test will break.

In the example, the fact that the `notify()` method directly depends on an implementation detail (the requests module) is a design issue, that is, it is taking its toll on the unit tests as well with the aforementioned fragility that is implied.

We still need to replace those methods with doubles (mocks), but if we refactor the code, we can do it in a better way. Let's separate these methods into smaller ones, and most importantly inject the dependency rather than keep it fixed. The code now applies the dependency inversion principle, and it expects to work with something that supports an interface (in this example, implicit one) such as the one the requests module provides:

```
from datetime import datetime
from constants import STATUS_ENDPOINT


class BuildStatus:
    endpoint = STATUS_ENDPOINT

    def __init__(self, transport):
        self.transport = transport

    @staticmethod
    def build_date() -> str:
        return datetime.now().isoformat()

    def compose_payload(self, merge_request_id, status) -> dict:
        return {
            "id": merge_request_id,
            "status": status,
            "built_at": self.build_date(),
        }

    def deliver(self, payload):
        response = self.transport.post(self.endpoint, json=payload)
        response.raise_for_status()
        return response
```

(continues on next page)

(continued from previous page)

```
    def notify(self, merge_request_id, status):
        return self.deliver(self.compose_payload(merge_request_id, status))
```

We separate the methods (not notify is now compose + deliver), make `compose_payload()` a new method (so that we can replace, without the need to patch the class), and require the transport dependency to be injected. Now that transport is a dependency, it is much easier to change that object for any double we want.

It is even possible to expose a fixture of this object with the doubles replaced as required:

```
@pytest.fixture
def build_status():
    bstatus = BuildStatus(Mock())
    bstatus.build_date = Mock(return_value="2018-01-01T00:00:01")
    return bstatus

def test_build_notification_sent(build_status):
    build_status.notify(1234, "OK")
    expected_payload = {
        "id": 1234,
        "status": "OK",
        "built_at": build_status.build_date(),
    }

    build_status.transport.post.assert_called_with(build_status.endpoint,␣
↪json=expected_payload)
```

21.3.2 3.2. Production code isn't the only thing that evolves

We keep saying that unit tests are as important as production code. And if we are careful enough with production code as to create the best possible abstraction, why wouldn't we do the same for unit tests?

If the code for unit tests is as important as the main code, then it's definitely wise to design it with extensibility in mind and make it as maintainable as possible. After all, this is the code that will have to be maintained by an engineer other than its original author, so it has to be readable.

The reason why we pay so much attention to make the code's flexibility is that we know requirements change and evolve over time, and eventually as domain business rules change, our code will have to change as well to support these new requirements. Since the production code changed to support new requirements, in turn, the testing code will have to change as well to support the newer version of the production code.

In one of the first examples we used, we created a series of tests for the merge request object, trying different combinations and checking the status at which the merge request was left. This is a good first approach, but we can do better than that.

Once we understand the problem better, we can start creating better abstractions. With this, the first idea that comes to mind is that we can create a higher-level abstraction that checks for particular conditions. For example, if we have an object that is a test suite that specifically targets the `MergeRequest` class, we know its functionality will be limited to the behavior of this class (because it should comply to the SRP), and therefore we could create specific testing methods on this testing class. These will only make sense for this class, but that will be helpful in reducing a lot of boilerplate code.

Instead of repeating assertions that follow the exact same structure, we can create a method that encapsulates this and reuse it across all of the tests:

```
class TestMergeRequestStatus(unittest.TestCase):

    def setUp(self):
        self.merge_request = MergeRequest()
```

(continues on next page)

(continued from previous page)

```
    def assert_rejected(self):
        self.assertEqual(self.merge_request.status, MergeRequestStatus.REJECTED)

    def assert_pending(self):
        self.assertEqual(self.merge_request.status, MergeRequestStatus.PENDING)

    def assert_approved(self):
        self.assertEqual(self.merge_request.status, MergeRequestStatus.APPROVED)

    def test_simple_rejected(self):
        self.merge_request.downvote("maintainer")
        self.assert_rejected()

    def test_just_created_is_pending(self):
        self.assert_pending()
```

If something changes with how we check the status of a merge request (or let's say we want to add extra checks), there is only one place (the `assert_approved()` method) that will have to be modified. More importantly, by creating these higher-level abstractions, the code that started as merely unit tests starts to evolve into what could end up being a testing framework with its own API or domain language, making testing more declarative.

21.4 4. More about unit testing

With the concepts we have revisited so far, we know how to test our code, think about our design in terms of how it is going to be tested, and configure the tools in our project to run the automated tests that will give us some degree of confidence over the quality of the software we have written.

If our confidence in the code is determined by the unit tests written on it, how do we know that they are enough? How could we be sure that we have been through enough on the test scenarios and that we are not missing some tests? Who says that these tests are correct? Meaning, who tests the tests?

The first part of the question, about being thorough on the tests we wrote, is answered by going beyond in our testing efforts through property-based testing.

The second part of the question might have multiple answers from different points of view, but we are going to briefly mention mutation testing as a means of determining that our tests are indeed correct. In this sense, we are thinking that the unit tests check our main productive code, and this works as a control for the unit tests as well.

21.4.1 4.1. Property-based testing

Property-based testing consists of generating data for tests cases with the goal of finding scenarios that will make the code fail, which weren't covered by our previous unit tests.

The main library for this is `hypothesis` which, configured along with our unit tests, will help us find problematic data that will make our code fail.

We can imagine that what this library does is find counter examples for our code. We write our production code (and unit tests for it!), and we claim it's correct. Now, with this library, we define some hypothesis that must hold for our code, and if there are some cases where our assertions don't hold, the `hypothesis` will provide a set of data that causes the error.

The best thing about unit tests is that they make us think harder about our production code. The best thing about `hypothesis` is that it makes us think harder about our unit tests.

21.4.2 4.2. Mutation testing

We know that tests are the formal verification method we have to ensure that our code is correct. And what makes sure that the test is correct? The production code, you might think, and yes, in a way this is correct, we can think of the main code as a counter balance for our tests.

The point in writing unit tests is that we are protecting ourselves against bugs, and testing for failure scenarios we really don't want to happen in production. It's good that the tests pass, but it would be bad if they pass for the wrong reasons. That is, we can use unit tests as an automatic regression tool: if someone introduces a bug in the code, later on, we expect at least one of our tests to catch it and fail. If this doesn't happen, either there is a test missing, or the ones we had are not doing the right checks.

This is the idea behind mutation testing. With a mutation testing tool, the code will be modified to new versions (called mutants), that are variations of the original code but with some of its logic altered (for example, operators are swapped, conditions are inverted, and so on). A good test suite should catch these mutants and kill them, in which case it means we can rely on the tests. If some mutants survive the experiment, it's usually a bad sign. Of course, this is not entirely precise, so there are intermediate states we might want to ignore.

To quickly show you how this works and to allow you to get a practical idea of this, we are going to use a different version of the code that computes the status of a merge request based on the number of approvals and rejections. This time, we have changed the code for a simple version that, based on these numbers, returns the result. We have moved the enumeration with the constants for the statuses to a separate module so that it now looks more compact:

```
from mrstatus import MergeRequestStatus as Status

def evaluate_merge_request(upvote_count, downvotes_count):
    if downvotes_count > 0:
        return Status.REJECTED
    if upvote_count >= 2:
        return Status.APPROVED
    return Status.PENDING
```

And now will we add a simple unit test, checking one of the conditions and its expected result :

```
class TestMergeRequestEvaluation(unittest.TestCase):
    def test_approved(self):
        result = evaluate_merge_request(3, 0)
        self.assertEqual(result, Status.APPROVED)
```

Now, we will install `mutpy`, a mutation testing tool for Python and tell it to run the mutation testing for this module with these tests:

```
$ mut.py \
--target mutation_testing_$N \
--unit-test test_mutation_testing_$N \
--operator AOD `# delete arithmetic operator` \
--operator AOR `# replace arithmetic operator` \
--operator COD `# delete conditional operator` \
--operator COI `# insert conditional operator` \
--operator CRP `# replace constant` \
--operator ROR `# replace relational operator` \
--show-mutants
```

The result is going to look something similar to this:

```
[*] Mutation score [0.04649 s]: 100.0%
- all: 4
- killed: 4 (100.0%)
- survived: 0 (0.0%)
- incompetent: 0 (0.0%)
- timeout: 0 (0.0%)
```

This is a good sign. Let's take a particular instance to analyze what happened. One of the lines on the output shows the following mutant:

```
- [# 1] ROR mutation_testing_1:11 :
--------------------------------------------------------
7: from mrstatus import MergeRequestStatus as Status
8:
9:
10: def evaluate_merge_request(upvote_count, downvotes_count):
~11:
if downvotes_count < 0:
12:
return Status.REJECTED
13:
if upvote_count >= 2:
14:
return Status.APPROVED
15:
return Status.PENDING
--------------------------------------------------------
[0.00401 s] killed by test_approved
(test_mutation_testing_1.TestMergeRequestEvaluation)
```

Notice that this mutant consists of the original version with the operator changed in line 11 (> for <), and the result is telling us that this mutant was killed by the tests. This means that with this version of the code (let's imagine that someone by mistakes makes this change), then the result of the function would have been APPROVED, and since the test expects it to be REJECTED, it fails, which is a good sign (the test caught the bug that was introduced).

Mutation testing is a good way to assure the quality of the unit tests, but it requires some effort and careful analysis. By using this tool in complex environments, we will have to take some time analyzing each scenario. It is also true that it is expensive to run these tests because it requires multiples runs of different versions of the code, which might take up too many resources and may take longer to complete. However, it would be even more expensive to have to make these checks manually and will require much more effort. Not doing these checks at all might be even riskier, because we would be jeopardizing the quality of the tests.

21.5 5. A brief introduction to test-driven development

There are entire books dedicated only to TDD, so it would not be realistic to try and cover this topic comprehensively. However, it's such an important topic that it has to be mentioned.

The idea behind TDD is that tests should be written before production code in a way that the production code is only written to respond to tests that are failing due to that missing implementation of the functionality.

There are multiple reasons why we would like to write the tests first and then the code. From a pragmatic point of view, we would be covering our production code quite accurately. Since all of the production code was written to respond to a unit test, it would be highly unlikely that there are tests missing for functionality (that doesn't mean that there is 100% of coverage of course, but at least all main functions, methods, or components will have their respective tests, even if they aren't completely covered).

The workflow is simple and at a high-level consist of three steps. First, we write a unit test that describes something we need to be implemented. When we run this test, it will fail, because that functionality has not been implemented yet. Then, we move onto implementing the minimal required code that satisfies that condition, and we run the test again. This time, the test should pass. Now, we can improve (refactor) the code.

This cycle has been popularized as the famous **red-green-refactor**, meaning that in the beginning, the tests fail (red), then we make them pass (green), and then we proceed to refactor the code and iterate it.

Part IV

Code optimization

Part V

Technical architecture

CHAPTER

TWENTYTWO

DESIGN PATTERNS

Design patterns have been a widespread topic in software engineering since their original inception in the famous **Gang of Four (GoF) book**, "Design Patterns: Elements of Reusable Object-Oriented Software". Design patterns help to solve common problems with abstractions that work for certain scenarios. When they are implemented properly, the general design of the solution can benefit from them.

In this chapter we take a look at some of the most common design patterns, but not from the perspective of tools to apply under certain conditions (once the patterns have been devised), but rather we analyze how design patterns contribute to clean code. After presenting a solution that implements a design pattern, we analyze how the final implementation is comparatively better as if we had chosen a different path.

As part of this analysis, we will see how to concretely implement design patterns in Python. As a result of that, we will see that the dynamic nature of Python implies some differences of implementation, with respect to other static typed languages, for which many of the design patterns were originally thought of. This means that there are some particularities about design patterns that you should bear in mind when it comes to Python, and, in some cases, trying to apply a design pattern where it doesn't really fit is non-Pythonic.

22.1 1. Considerations for design patterns in Python

Object-oriented design patterns are ideas of software construction that appear in different scenarios when we deal with models of the problem we're solving. Because they're high- level ideas, it's hard to think of them as being tied to particular programming languages. They are instead more general concepts about how objects will interact in the application. Of course, they will have their implementation details, varying from language to language, but that doesn't form the essence of a design pattern.

That's the theoretical aspect of a design pattern, the fact that it is an abstract idea that expresses concepts about the layout of the objects in the solution. There are plenty of other books and several other resources about object-oriented design, and design patterns in particular, so we are going to focus on those implementation details for Python.

Given the nature of Python, some of the classical design patterns aren't actually needed. That means that Python already supports features that render those patterns invisible. Some argue that they don't exist in Python, but keep in mind that invisible doesn't mean non-existing. They are there, just embedded in Python itself, so it's likely that we won't even notice them.

Others have a much simpler implementation, again thanks to the dynamic nature of the language, and the rest of them are practically the same as they are in other platforms, with small differences.

In any case, the important goal for achieving clean code in Python is knowing what patterns to implement and how. That means recognizing some of the patterns that Python already abstracts and how we can leverage them. For instance, it would be completely non- Pythonic to try to implement the standard definition of the iterator pattern (as we would do in different languages), because (as we have already covered) iteration is deeply embedded in Python, and the fact that we can create objects that will directly work in a for loop makes this the right way to proceed.

Something similar happens with some of the creational patterns. Classes are regular objects in Python, and so are functions. As we have seen in several examples so far, they can be passed around, decorated, reassigned, and so on. That means that whatever kind of customization we would like to make to our objects, we can most likely

do it without needing any particular setup of factory classes. In addition, there is no special syntax for creating objects in Python (no new keyword, for example). This is another reason why, most of the time, a simple function call will just work as a factory.

Other patterns are still needed, and we will see how, with some small adaptations, we can make them more Pythonic, taking full advantage of the features that the language provides (magic methods or the standard library).

Out of all the patterns available, not all of them are equally frequent, nor useful, so we will focus on the main ones, those that we would expect to see the most in our applications, and we will do so by following a pragmatic approach.

22.2 2. Design patterns in action

The canonical reference in this subject, as written by the GoF, introduces 23 design patterns, each falling under one of the creational, structural, and behavioral categories. There are even more patterns or variations of existing ones, but rather than learning all of these patterns off by heart, we should focus on keeping two things in mind. Some of the patterns are invisible in Python, and we use them probably without even noticing. Secondly, not all patterns are equally common; some of them are tremendously useful, and so they are found very frequently, while others are for more specific cases.

In this section, we will revisit the most common patterns, those that are most likely to emerge from our design. Note the use of the word emerge here. It is important. We should not force the application of a design pattern to the solution we are building, but rather evolve, refactor, and improve our solution until a pattern emerges.

Design patterns are therefore not invented but discovered. When a situation that occurs repeatedly in our code reveals itself, the general and more abstract layout of classes, objects, and related components appears under a name by which we identify a pattern.

Thinking the same thing, but now backward, we realize that the name of a design pattern wraps up a lot of concepts. This is probably the best thing about design patterns; they provide a language. Through design patterns, it's easier to communicate design ideas effectively. When two or more software engineers share the same vocabulary, and one of them mentions builder, the rest of them can immediately think about all the classes, and how they would be related, what their mechanics would be, and so on, without having to repeat this explanation all over again.

The reader will notice that the code shown in this chapter is different from the canonical or original envisioning of the design pattern in question. There is more than one reason for this. The first reason is that the examples take a more pragmatic approach, aimed at solutions for particular scenarios rather than exploring general design theory. The second reason is that the patterns are implemented with the particularities of Python, which in some cases are very subtle, but in other cases, the differences are noticeable, generally simplifying the code.

22.2.1 2.1. Creational patterns

In software engineering, creational patterns are those that deal with object instantiation, trying to abstract away much of the complexity (like determining the parameters to initialize an object, all the related objects that might be needed, etc.), in order to leave the user with a simpler interface, that should be safer to use. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

Out of the five patterns for creating objects, we will discuss mainly the variants that are used to avoid the singleton pattern, and replace it with the Borg pattern (most commonly used in Python applications), discussing their differences and advantages.

2.1.1. Factories

As was mentioned in the introduction, one of the core features of Python is that everything is an object, and as such, they can all be treated equally. This means that there are no special distinctions of things that we can or cannot do with classes, functions, or custom objects. They can all be passed by parameter, assigned, and so on.

It is for this reason that many of the factory patterns are not really needed. We could just simply define a function that will construct a set of objects, and we can even pass the class that we want to create by a parameter.

2.1.2. Singleton and shared state (monostate)

The singleton pattern, on the other hand, is something not entirely abstracted away by Python. The truth is that most of the time, this pattern is either not really needed or is a bad choice. There are a lot of problems with singletons (after all, they are, in fact, a form of global variables for object-oriented software, and as such, are a bad practice). They are hard to unit test, the fact that they might be modified at any time by any object makes them hard to predict, and their side-effects can be really problematic.

As a general principle, we should avoid using singletons as much as possible. If in some extreme case, they are required, the easiest way of achieving this in Python is by using a module. We can create an object in a module, and once it's there, it will be available from every part of the module that is imported. Python itself makes sure that modules are already singletons, in the sense that no matter how many times they're imported, and from how many places, the same module is always the one that is going to be loaded into `sys.modules`.

2.1.2.1. Shared state

Rather than forcing our design to have a singleton in which only one instance is created, no matter how the object is invoked, constructed, or initialized, it is better to replicate the data across multiple instances.

The idea of the monostate pattern is that we can have many instances that are just regular objects, without having to care whether they're singletons or not (seeing as they're just objects). The good thing about this pattern is that these objects will have their information synchronized, in a completely transparent way, without us having to worry about how this works internally.

This makes this pattern a much better choice, not only for its convenience, but also because it is less error-prone, and suffers from fewer of the disadvantages of singletons (regarding their testability, creating derived classes, and so on).

We can use this pattern on many levels, depending on how much information we need to synchronize.

In its simplest form, we can assume that we only need to have one attribute to be reflected across all instances. If that is the case, the implementation is as trivial as using a class variable, and we just need to take care in providing a correct interface to update and retrieve the value of the attribute.

Let's say we have an object that has to pull a version of a code in a Git repository by the latest tag . There might be multiple instances of this object, and when every client calls the method for fetching the code, this object will use the tag version from its attribute. At any point, this tag can be updated for a newer version, and we want any other instance (new or already created) to use this new branch when the fetch operation is being called, as shown in the following code:

```
class GitFetcher:
    _current_tag = None

def __init__(self, tag):
    self.current_tag = tag

@property
def current_tag(self):
    if self._current_tag is None:
        raise AttributeError("tag was never set")
    return self._current_tag
```

(continues on next page)

(continued from previous page)

```
@current_tag.setter
def current_tag(self, new_tag):
    self.__class__._current_tag = new_tag

def pull(self):
    logger.info("pulling from %s", self.current_tag)
    return self.current_tag
```

The reader can simply verify that creating multiple objects of the GitFetcher type with different versions will result in all objects being set with the latest version at any time, as shown in the following code:

```
>>> f1 = GitFetcher(0.1)
>>> f2 = GitFetcher(0.2)
>>> f1.current_tag = 0.3
>>> f2.pull()
0.3
>>> f1.pull()
0.3
```

In the case that we need more attributes, or that we wish to encapsulate the shared attribute a bit more, to make the design cleaner, we can use a descriptor.

A descriptor, like the one shown in the following code, solves the problem, and while it's true that it requires more code, it also encapsulates a more concrete responsibility, and part of the code is actually moved away from our original class, making either one of them more cohesive and compliant with the single responsibility principle:

```
class SharedAttribute:

    def __init__(self, initial_value=None):
        self.value = initial_value
        self._name = None

    def __get__(self, instance, owner):
        if instance is None:
            return self
        if self.value is None:
            raise AttributeError(f"{self._name} was never set")
        return self.value

    def __set__(self, instance, new_value):
        self.value = new_value

    def __set_name__(self, owner, name):
        self._name = name
```

Apart from these considerations, it's also true that the pattern is now more reusable. If we want to repeat this logic, we just have to create a new descriptor object that would work (complying with the DRY principle).

If we now want to do the same, but for the current branch, we create this new class attribute, and the rest of the class is kept intact, while still having the desired logic in place, as shown in the following code:

```
class GitFetcher:
    current_tag = SharedAttribute()
    current_branch = SharedAttribute()

    def __init__(self, tag, branch=None):
        self.current_tag = tag
        self.current_branch = branch

    def pull(self):
```

(continues on next page)

(continued from previous page)

```
        logger.info("pulling from %s", self.current_tag)
        return self.current_tag
```

The balance and trade-off of this new approach should be clear by now. This new implementation uses a bit more code, but it's reusable, so it saves lines of code (and duplicated logic) in the long run. Once again, refer to the three or more instances rule to decide if you should create such an abstraction.

Another important benefit of this solution is that it also reduces the repetition of unit tests. Reusing code here will give us more confidence on the overall quality of the solution, because now we just have to write unit tests for the descriptor object, not for all the classes that use it (we can safely assume that they're correct as long as the unit tests prove the descriptor to be correct).

2.1.2.2. The borg pattern

The previous solutions should work for most cases, but if we really have to go for a singleton (and this has to be a really good exception), then there is one last better alternative to it, only this is a riskier one.

This is the actual monostate pattern, referred to as the borg pattern in Python. The idea is to create an object that is capable of replicating all of its attributes among all instances of the same class. The fact that absolutely every attribute is being replicated has to be a warning to keep in mind undesired side-effects. Still, this pattern has many advantages over the singleton.

In this case, we are going to split the previous object into two: one that works over Git tags, and the other over branches. And we are using the code that will make the borg pattern work:

```
class BaseFetcher:
    def __init__(self, source):
        self.source = source

class TagFetcher(BaseFetcher):
    _attributes = {}

    def __init__(self, source):
        self.__dict__ = self.__class__._attributes
        super().__init__(source)

    def pull(self):
        logger.info("pulling from tag %s", self.source)
        return f"Tag = {self.source}"

class BranchFetcher(BaseFetcher):
    _attributes = {}

    def __init__(self, source):
        self.__dict__ = self.__class__._attributes
        super().__init__(source)

    def pull(self):
        logger.info("pulling from branch %s", self.source)
        return f"Branch = {self.source}"
```

Both objects have a base class, sharing their initialization method. But then they have to implement it again in order to make the borg logic work. The idea is that we use a class attribute that is a dictionary to store the attributes, and then we make the dictionary of each object (at the time it's being initialized) to use this very same dictionary. This means that any update on the dictionary of an object will be reflected in the class, which will be the same for the rest of the objects because their class is the same, and dictionaries are mutable objects that are passed as a reference. In other words, when we create new objects of this type, they will all use the same dictionary, and this dictionary is constantly being updated.

Note that we cannot put the logic of the dictionary on the base class, because this will mix the values among the

objects of different classes, which is not what we want. This boilerplate solution is what would make many think it's actually an idiom rather than a pattern.

A possible way of abstracting this in a way that achieves the DRY principle would be to create a mixin class, as shown in the following code:

```
class SharedAllMixin:
    def __init__(self, *args, **kwargs):
        try:
            self.__class__._attributes
        except AttributeError:
            self.__class__._attributes = {}

        self.__dict__ = self.__class__._attributes
        super().__init__(*args, **kwargs)

class BaseFetcher:
    def __init__(self, source):
        self.source = source

class TagFetcher(SharedAllMixin, BaseFetcher):
    def pull(self):
        logger.info("pulling from tag %s", self.source)
        return f"Tag = {self.source}"

class BranchFetcher(SharedAllMixin, BaseFetcher):
    def pull(self):
        logger.info("pulling from branch %s", self.source)
        return f"Branch = {self.source}"
```

This time, we are using the mixin class to create the dictionary with the attributes in each class in case it doesn't already exist, and then continuing with the same logic.

This implementation should not have any major problems with inheritance, so it's a more viable alternative.

2.1.3. Builder

The builder pattern is an interesting pattern that abstracts away all the complex initialization of an object. This pattern does not rely on any particularity of the language, so it's as equally applicable in Python as it would be in any other language.

While it solves a valid case, it's usually also a complicated case that is more likely to appear in the design of a framework, library, or an API. Similar to the recommendations given for descriptors, we should reserve this implementation for cases where we expect to expose an API that is going to be consumed by multiple users.

The high level idea of this patter is that we need to create a complex object, that is an object that also requires many others to work with. Rather than letting the user create all those auxiliary objects, and then assign them to the main one, we would like to create an abstraction that allows all of that to be done in a single step. In order to achieve this, we will have a builder object that knows how to create all the parts and link them together, giving the user an interface (which could be a class method), to parametrize all the information about what the resulting object should look like.

22.2.2 2.2. Structural patterns

Structural patterns are useful for situations where we need to create simpler interfaces or objects that are more powerful by extending their functionality without adding complexity to their interfaces.

The best thing about these patterns is that we can create more interesting objects, with enhanced functionality, and we can achieve this in a clean way; that is, by composing multiple single objects (the clearest example of this being the composite pattern), or by gathering many simple and cohesive interfaces.

2.2.1. Adapter

The adapter pattern is probably one of the simplest design patterns there are, and one of the most useful ones at the same time. Also known as a wrapper, this pattern solves the problem of adapting interfaces of two or more objects that are not compatible.

We typically encounter the situation where part of our code works with a model or set of classes that were polymorphic with respect to a method. For example, if there were multiple objects for retrieving data with a `fetch()` method, then we want to maintain this interface so we don't have to make major changes to our code.

But then we come to a point where the need to add a new data source, and alas, this one won't have a `fetch()` method. To make things worse, not only is this type of object not compatible, but it is also not something we control (perhaps a different team decided on the API, and we cannot modify the code).

Instead of using this object directly, we adopt its interface to the one we need. There are two ways of doing this.

The first way would be to create a class that inherits from the one we want to use, and that creates an alias for the method (if required, it will also have to adapt the parameters and the signature).

By means of inheritance, we import the external class and create a new one that will define the new method, calling the one that has a different name. In this example, let's say the external dependency has a method named `search()`, which takes only one parameter for the search because it queries in a different fashion, so our `adapter` method not only calls the external one, but it also translates the parameters accordingly, as shown in the following code:

```
from _adapter_base import UsernameLookup


class UserSource(UsernameLookup):
    def fetch(self, user_id, username):
        user_namespace = self._adapt_arguments(user_id, username)
        return self.search(user_namespace)

    @staticmethod
    def _adapt_arguments(user_id, username):
        return f"{user_id}:{username}"
```

It might be the case that our class already derives from another one, in which case, this will end up as a case of multiple inheritances, which Python supports, so it shouldn't be a problem. However, as we have seen many times before, inheritance comes with more coupling (who knows how many other methods are being carried from the external library?), and it's inflexible. Conceptually, it also wouldn't be the right choice because we reserve inheritance for situations of specification (an is a kind of relationship), and in this case, it's not clear at all that our object has to be one of the kinds that are provided by a third-party library (especially since we don't fully comprehend that object).

Therefore, a better approach would be to use composition instead. Assuming that we can provide our object with an instance of UsernameLookup, the code would be as simple as just redirecting the petition prior to adopting the parameters, as shown in the following code:

```
class UserSource:
    ...
    def fetch(self, user_id, username):
```

(continues on next page)

(continued from previous page)

```
        user_namespace = self._adapt_arguments(user_id, username)
        return self.username_lookup.search(user_namespace)
```

If we need to adopt multiple methods, and we can devise a generic way of adapting their signature as well, it might be worth using the `__getattr__()` magic method to redirect requests towards the wrapped object, but as always with generic implementations, we should be careful of not adding more complexity to the solution.

2.2.2. Composite

There will be parts of our programs that require us to work with objects that are made out of other objects. We have base objects that have a well-defined logic, and then we will have other container objects that will group a bunch of base objects, and the challenge is that we want to treat both of them (the base and the container objects) without noticing any differences.

The objects are structured in a tree hierarchy, where the basic objects would be the leaves of the tree, and the composed objects intermediate nodes. A client might want to call any of them to get the result of a method that is called. The composite object, however, will act as a client; this also will pass this request along with all the objects it contains whether they are leaves or other intermediate notes until they all are processed.

Imagine a simplified version of an online store in which we have products. Say that we offer the possibility of grouping those products, and we give customers a discount per group of products. A product has a price, and this value will be asked for when the customers come to pay. But a set of grouped products also has a price that has to be computed. We will have an object that represents this group that contains the products, and that delegates the responsibility of asking the price to each particular product (which might be another group of products as well), and so on, until there is nothing else to compute. The implementation of this is shown in the following code:

```
class Product:
    def __init__(self, name, price):
        self._name = name
        self._price = price

    @property
    def price(self):
        return self._price

class ProductBundle:
    def __init__(self,
                 name,
                 perc_discount,
                 *products: Iterable[Union[Product, "ProductBundle"]]) -> None:

    self._name = name
    self._perc_discount = perc_discount
    self._products = products

    @property
    def price(self):
        total = sum(p.price for p in self._products)
        return total * (1 - self._perc_discount)
```

We expose the public interface through a property, and leave the price as a private attribute. The `ProductBundle` class uses this property to compute the value with the discount applied by first adding all the prices of all the products it contains.

The only discrepancy between these objects is that they are created with different parameters. To be fully compatible, we should have tried to mimic the same interface and then added extra methods for adding products to the bundle but using an interface that allows the creation of complete objects. Not needing these extra steps is an advantage that justifies this small difference.

2.2.3. Decorator

Don't confuse the decorator pattern with the concept of a Python decorator. There is some resemblance, but the idea of the design pattern is quite different.

This pattern allows us to dynamically extend the functionality of some objects, without needing inheritance. It's a good alternative to multiple inheritance in creating more flexible objects.

We are going to create a structure that let's a user define a set of operations (decorations) to be applied over an object, and we'll see how each step takes place in the specified order.

The following code example is a simplified version of an object that constructs a query in the form of a dictionary from parameters that are passed to it (it might be an object that we would use for running queries to elasticsearch, for instance, but the code leaves out distracting implementation details to focus on the concepts of the pattern).

In its most basic form, the query just returns the dictionary with the data it was provided when it was created. Clients expect to use the `render()` method of this object:

```
class DictQuery:
    def __init__(self, **kwargs):
        self._raw_query = kwargs

    def render(self) -> dict:
        return self._raw_query
```

Now we want to render the query in different ways by applying transformations to the data (filtering values, normalizing them, and so on). We could create decorators and apply them to the render method, but that wouldn't be flexible enough what if we want to change them at runtime? Or if we want to select some of them, but not others?

The design is to create another object, with the same interface and the capability of enhancing (decorating) the original result through many steps, but which can be combined. These objects are chained, and each one of them does what it was originally supposed to do, plus something else. This something else is the particular decoration step.

Since Python has duck typing, we don't need to create a new base class and make these new objects part of that hierarchy, along with `DictQuery`. Simply creating a new class that has a `render()` method will be enough (again, polymorphism should not require inheritance). This process is shown in the following code:

```
class QueryEnhancer:
    def __init__(self, query: DictQuery):
        self.decorated = query

    def render(self):
        return self.decorated.render()

class RemoveEmpty(QueryEnhancer):
    def render(self):
        original = super().render()
        return {k: v for k, v in original.items() if v}

class CaseInsensitive(QueryEnhancer):
    def render(self):
        original = super().render()
        return {k: v.lower() for k, v in original.items()}
```

The `QueryEnhancer` phrase has an interface that is compatible with what the clients of `DictQuery` are expecting, so they are interchangeable. This object is designed to receive a decorated one. It's going to take the values from this and convert them, returning the modified version of the code.

If we want to remove all values that evaluate to `False` and normalize them to form our original query, we would have to use the following schema:

```
>>> original = DictQuery(key="value", empty="", none=None, upper="UPPERCASE",␣
↪title="Title")
>>> new_query = CaseInsensitive(RemoveEmpty(original))
>>> original.render()
{'key': 'value', 'empty': '', 'none': None, 'upper': 'UPPERCASE', 'title':
'Title'}
>>> new_query.render()
{'key': 'value', 'upper': 'uppercase', 'title': 'title'}
```

This is a pattern that we can also implement in different ways, taking advantage of the dynamic nature of Python, and the fact that functions are objects. We could implement this pattern with functions that are provided to the base decorator object (`QueryEnhancer`), and define each decoration step as a function, as shown in the following code:

```
class QueryEnhancer:
    def __init__(self,
                 query: DictQuery,
                 *decorators: Iterable[Callable[[Dict[str, str]], Dict[str,␣
↪str]]]) -> None:
        self._decorated = query
        self._decorators = decorators

    def render(self):
        current_result = self._decorated.render()
        for deco in self._decorators:
            current_result = deco(current_result)

        return current_result
```

With respect to the client, nothing has changed because this class maintains the compatibility through its `render()` method. Internally, however, this object is used in a slightly different fashion, as shown in the following code:

```
>>> query = DictQuery(foo="bar", empty="", none=None, upper="UPPERCASE",
title="Title")
>>> QueryEnhancer(query, remove_empty, case_insensitive).render()
{'foo': 'bar', 'upper': 'uppercase', 'title': 'title'}
```

In the preceding code, `remove_empty` and `case_insensitive` are just regular functions that transform a dictionary.

In this example, the function-based approach seems easier to understand. There might be cases with more complex rules that rely on data from the object being decorated (not only its result), and in those cases, it might be worth going for the object-oriented approach, especially if we really want to create a hierarchy of objects where each class actually represents some knowledge we want to make explicit in our design.

2.2.4. Facade

Facade is an excellent pattern. It's useful in many situations where we want to simplify the interaction between objects. The pattern is applied where there is a relation of many-to-many among several objects, and we want them to interact. Instead of creating all of these connections, we place an intermediate object in front of many of them that act as a facade.

The facade works as a hub or a single point of reference in this layout. Every time a new object wants to connect to another one, instead of having to have N interfaces for all N possible objects it needs to connect to, it will instead just talk to the facade, and this will redirect the request accordingly. Everything that's behind the facade is completely opaque to the rest of the external objects.

Apart from the main and obvious benefit (the decoupling of objects), this pattern also encourages a simpler design with fewer interfaces and better encapsulation.

This is a pattern that we can use not only for improving the code of our domain problem but also to create better APIs. If we use this pattern and provide a single interface, acting as a single point of truth or entry point for our code, it will be much easier for our users to interact with the functionality exposed. Not only that, but by exposing a functionality and hiding everything behind an interface, we are free of changing or refactoring that underlying code as many times as we want, because as long as it is behind the facade, it will not break backward compatibility, and our users will not be affected.

Note how this idea of using facades is not even limited to objects and classes, but also applies to packages (technically, packages are objects in Python, but still). We can use this idea of the facade to decide the layout of a package; that is, what is visible to the user and importable, and what is internal and should not be imported directly.

When we create a directory to build a package, we place the `__init__.py` file along with the rest of the files. This is the root of the module, a sort of facade. The rest of the files define the objects to export, but they shouldn't be directly imported by clients. The init file should import them and then clients should get them from there. This creates a better interface because users only need to know a single entry point from which to get the objects, and more importantly, the package (the rest of the files) can be refactored or rearranged as many times as needed, and this will not affect clients as long as the main API on the init file is maintained. It is of utmost importance to keep principles like this one in mind in order to build maintainable software.

There is an example of this in Python itself, with the os module. This module groups an operating system's functionality, but underneath it, uses the posix module for Portable Operating System Interface (POSIX) operating systems (this is called nt in Windows platforms). The idea is that, for portability reasons, we shouldn't ever really import the posix module directly, but always the os module. It is up to this module to determine from which platform it is being called, and expose the corresponding functionality.

22.2.3 2.3. Behavioral patterns

Behavioral patterns aim to solve the problem of how objects should cooperate, how they should communicate, and what their interfaces should be at run-time.

This can be accomplished statically by means of inheritance or dynamically by using composition. Regardless of what the pattern uses, what we will see throughout the following examples is that what these patterns have in common is the fact that the resulting code is better in some significant way, whether this is because it avoids duplication or creates good abstractions that encapsulate behavior accordingly and decouple our models.

2.3.1. Chain of responsibility

Now we are going to take another look at our event systems. We want to parse information about the events that happened on the system from the log lines (text files, dumped from our HTTP application server, for example), and we want to extract this information in a convenient way.

In our previous implementation, we achieved an interesting solution that was compliant with the open/closed principle and relied on the use of the `__subclasses__()` magic method to discover all possible event types and process the data with the right event, resolving the responsibility through a method encapsulated on each class.

This solution worked for our purposes, and it was quite extensible, but as we'll see, this design pattern will bring additional benefits.

The idea here is that we are going to create the events in a slightly different way. Each event still has the logic to determine whether or not it can process a particular log line, but it will also have a successor. This successor is a new event, the next one in the line, that will continue processing the text line in case the first one was not able to do so. The logic is simple—we chain the events, and each one of them tries to process the data. If it can, then it just returns the result. If it can't, it will pass it to its successor and repeat, as shown in the following code:

```
import re


class Event:
    pattern = None
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, next_event=None):
        self.successor = next_event

    def process(self, logline: str):
        if self.can_process(logline):
            return self._process(logline)
        if self.successor is not None:
            return self.successor.process(logline)

    def _process(self, logline: str) -> dict:
        parsed_data = self._parse_data(logline)
        return {
            "type": self.__class__.__name__,
            "id": parsed_data["id"],
            "value": parsed_data["value"],
        }

    @classmethod
    def can_process(cls, logline: str) -> bool:
        return cls.pattern.match(logline) is not None

    @classmethod
    def _parse_data(cls, logline: str) -> dict:
        return cls.pattern.match(logline).groupdict()

class LoginEvent(Event):
    pattern = re.compile(r"(?P<id>\d+):\s+login\s+(?P<value>\S+)")

class LogoutEvent(Event):
    pattern = re.compile(r"(?P<id>\d+):\s+logout\s+(?P<value>\S+)")
```

With this implementation, we create the event objects, and arrange them in the particular order in which they are going to be processed. Since they all have a `process()` method, they are polymorphic for this message, so the order in which they are aligned is completely transparent to the client, and either one of them would be transparent too. Not only that, but the `process()` method has the same logic; it tries to extract the information if the data provided is correct for the type of object handling it, and if not, it moves on to the next one in the line.

This way, we could process a login event in the following way:

```
>>> chain = LogoutEvent(LoginEvent())
>>> chain.process("567: login User")
{'type': 'LoginEvent', 'id': '567', 'value': 'User'}
```

Note how `LogoutEvent` received `LoginEvent` as its successor, and when it was asked to process something that it couldn't handle, it redirected to the correct object. As we can see from the type key on the dictionary, `LoginEvent` was the one that actually created that dictionary.

This solution is flexible enough, and shares an interesting trait with our previous one: all conditions are mutually exclusive. As long as there are no collisions, and no piece of data has more than one handler, processing the events in any order will not be an issue.

But what if we cannot make such an assumption? With the previous implementation, we could still change the `__subclasses__()` call for a list that we made according to our criteria, and that would have worked just fine. And what if we wanted that order of precedence to be determined at runtime (by the user or client, for example)? That would be a shortcoming.

With the new solution, it's possible to accomplish such requirements, because we assemble the chain at runtime, so we can manipulate it dynamically as we need to.

For example, now we add a generic type that groups both the login and logout a session event, as shown in the following code:

```
class SessionEvent(Event):
    pattern = re.compile(r"(?P<id>\d+):\s+log(in|out)\s+(?P<value>\S+)")
```

If for some reason, and in some part of the application, we want to capture this before the login event, this can be done by the following chain:

```
chain = SessionEvent(LoginEvent(LogoutEvent()))
```

By changing the order, we can, for instance, say that a generic session event has a higher priority than the login, but not the logout, and so on.

The fact that this pattern works with objects makes it more flexible with respect to our previous implementation, which relied on classes (and while they are still objects in Python, they aren't excluded from some degree of rigidity).

2.3.2. The template method

The `template` method is a pattern that yields important benefits when implemented properly. Mainly, it allows us to reuse code, and it also makes our objects more flexible and easy to change while preserving polymorphism.

The idea is that there is a class hierarchy that defines some behavior, let's say an important method of its public interface. All of the classes of the hierarchy share a common template and might need to change only certain elements of it. The idea, then, is to place this generic logic in the public method of the parent class that will internally call all other (private) methods, and these methods are the ones that the derived classes are going to modify; therefore, all the logic in the template is reused.

Avid readers might have noticed that we already implemented this pattern in the previous section (as part of the chain of responsibility example). Note that the classes derived from `Event` implement only one thing their particular pattern. For the rest of the logic, the template is in the `Event` class. The process event is generic, and relies on two auxiliary methods `can_process()` and `process()` (which in turn calls `_parse_data()`).

These extra methods rely on a class attribute pattern. Therefore, in order to extend this with a new type of object, we just have to create a new derived class and place the regular expression. After that, the rest of the logic will be inherited with this new attribute changed. This reuses a lot of code because the logic for processing the log lines is defined once and only once in the parent class.

This makes the design flexible because preserving the polymorphism is also easily achievable. If we need a new event type that for some reason needs a different way of parsing data, we only override this private method in that subclass, and the compatibility will be kept, as long as it returns something of the same type as the original one (complying with Liskov's substitution and open/closed principles). This is because it is the parent class that is calling the method from the derived classes.

This pattern is also useful if we are designing our own library or framework. By arranging the logic this way, we give users the ability to change the behavior of one of the classes quite easily. They would have to create a subclass and override the particular private method, and the result will be a new object with the new behavior that is guaranteed to be compatible with previous callers of the original object.

2.3.3. Command

The command pattern provides us with the ability to separate an action that needs to be done from the moment that it is requested to its actual execution. More than that, it can also separate the original request issued by a client from its recipient, which might be a different object. In this section, we are going to focus mainly on the first aspect of the patterns; the fact that we can separate how an order has to be run from when it actually executes.

We know we can create callable objects by implementing the `__call__()` magic method, so we could just initialize the object and then call it later on. In fact, if this is the only requirement, we might even achieve this through a nested function that, by means of a closure, creates another function to achieve the effect of a delayed execution. But this pattern can be extended to ends that aren't so easily achievable.

The idea is that the command might also be modified after its definition. This means that the client specifies a command to run, and then some of its parameters might be changed, more options added, and so on, until someone finally decides to perform the action.

Examples of this can be found in libraries that interact with databases. For instance, in `psycopg2` (a PostgreSQL client library), we establish a connection. From this, we get a cursor, and to that cursor we can pass an SQL statement to run. When we call the execute method, the internal representation of the object changes, but nothing is actually run in the database. It is when we call `fetchall()` (or a similar method) that the data is actually queried and is available in the cursor.

The same happens in the popular Object Relational Mapper SQLAlchemy (ORM SQLAlchemy). A query is defined through several steps, and once we have the query object, we can still interact with it (add or remove filters, change the conditions, apply for an order, and so on), until we decide we want the results of the query. After calling each method, the query object changes its internal properties and returns self (itself).

These are examples that resemble the behavior that we would like to achieve. A very simple way of creating this structure would be to have an object that stores the parameters of the commands that are to be run. After that, it has to also provide methods for interacting with those parameters (adding or removing filters, and so on). Optionally, we can add tracing or logging capabilities to that object to audit the operations that have been taking place. Finally, we need to provide a method that will actually perform the action. This one can be just `__call__()` or a custom one. Let's call it `do()`.

2.3.4. State

The state pattern is a clear example of reification in software design, making the concept of our domain problem an explicit object rather than just a side value.

Previously, we had an object that represented a merge request, and it had a state associated with it (open, closed, and so on). We used an enum to represent those states because, at that point, they were just data holding a value the string representation of that particular state. If they had to have some behavior, or the entire merge request had to perform some actions depending on its state and transitions, this would not have been enough.

The fact that we are adding behavior, a runtime structure, to a part of the code has to make us think in terms of objects, because that's what objects are supposed to do, after all. And here comes the reification: now the state cannot just simply be an enumeration with a string; it needs to be an object.

Imagine that we have to add some rules to the merge request say, that when it moves from open to closed, all approvals are removed (they will have to review the code again), and that when a merge request is just opened, the number of approvals is set to zero (regardless of whether it's a reopened or a brand new merge request). Another rule could be that when a merge request is merged, we want to delete the source branch, and of course, we want to forbid users from performing invalid transitions (for example, a closed merge request cannot be merged, and so on).

If we were to put all that logic into a single place, namely in the MergeRequest class, we will end up with a class that has lots of responsibilities (a poor design), probably many methods, and a very large number of if statements. It would be hard to follow the code and to understand which part is supposed to represent which business rule.

It's better to distribute this into smaller objects, each one with fewer responsibilities, and the state objects are a good place for this. We create an object for each kind of state we want to represent, and, in their methods, we place the logic for the transitions with the aforementioned rules. The `MergeRequest` object will then have a state collaborator, and this, in turn, will also know about `MergeRequest` (the double-dispatching mechanism is needed to run the appropriate actions on `MergeRequest` and handle the transitions).

We define a base abstract class with the set of methods to be implemented, and then a subclass for each particular state we want to represent. Then the MergeRequest object delegates all the actions to state , as shown in the following code:

```
class InvalidTransitionError(Exception):
    """Raised when trying to move to a target state from an unreachable source␣
↪state."""

class MergeRequestState(abc.ABC):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, merge_request):
        self._merge_request = merge_request

    @abc.abstractmethod
    def open(self):
    ...

    @abc.abstractmethod
    def close(self):
    ...

    @abc.abstractmethod
    def merge(self):
    ...

    def __str__(self):
        return self.__class__.__name__

class Open(MergeRequestState):
    def open(self):
        self._merge_request.approvals = 0

    def close(self):
        self._merge_request.approvals = 0
        self._merge_request.state = Closed

    def merge(self):
        logger.info("merging %s", self._merge_request)
        logger.info("deleting branch %s",
        self._merge_request.source_branch)
        self._merge_request.state = Merged

class Closed(MergeRequestState):
    def open(self):
        logger.info("reopening closed merge request %s",
        self._merge_request)
        self._merge_request.state = Open

    def close(self):
        pass

    def merge(self):
        raise InvalidTransitionError("can't merge a closed request")

class Merged(MergeRequestState):
    def open(self):
        raise InvalidTransitionError("already merged request")
    def close(self):
        raise InvalidTransitionError("already merged request")
    def merge(self):
        pass

class MergeRequest:
    def __init__(self, source_branch: str, target_branch: str) -> None:
        self.source_branch = source_branch
        self.target_branch = target_branch
        self._state = None
        self.approvals = 0
        self.state = Open

    @property
```

(continues on next page)

(continued from previous page)

```
    def state(self):
        return self._state

    @state.setter
    def state(self, new_state_cls):
        self._state = new_state_cls(self)

    def open(self):
        return self.state.open()

    def close(self):
        return self.state.close()

    def merge(self):
        return self.state.merge()

    def __str__(self):
        return f"{self.target_branch}:{self.source_branch}"
```

The following list outlines some clarifications about implementation details and the design decisions that should be made:

- The state is a property, so not only is it public, but there is a single place with the definitions of how states are created for a merge request, passing `self` as a parameter.
- The abstract base class is not strictly needed, but there are benefits to having it. First, it makes the kind of object we are dealing with more explicit. Second, it forces every substate to implement all the methods of the interface. There are two alternatives to this:
  - We could have not put the methods, and let `AttributeError` raise when trying to perform an invalid action, but this is not correct, and it doesn't express what happened.
  - Related to this point is the fact that we could have just used a simple base class and left those methods empty, but then the default behavior of not doing anything doesn't make it any clearer what should happen. If one of the methods in the subclass should do nothing (as in the case of merge), then it's better to let the empty method just sit there and make it explicit that for that particular case, nothing should be done, as opposed to force that logic to all objects.
- `MergeRequest` and `MergeRequestState` have links to each other. The moment a transition is made, the former object will not have extra references and should be garbage-collected, so this relationship should be always 1:1. With some small and more detailed considerations, a weak reference might be used.

The following code shows some examples of how the object is used:

```
>>> mr = MergeRequest("develop", "master")
>>> mr.open()
>>> mr.approvals
0
>>> mr.approvals = 3
>>> mr.close()
>>> mr.approvals
0
>>> mr.open()
INFO:log:reopening closed merge request master:develop
>>> mr.merge()
INFO:log:merging master:develop
INFO:log:deleting branch develop
>>> mr.close()
Traceback (most recent call last):
...
InvalidTransitionError: already merged request
```

The actions for transitioning states are delegated to the state object, which `MergeRequest` holds at all times (this

can be any of the subclasses of `ABC`). They all know how to respond to the same messages (in different ways), so these objects will take the appropriate actions corresponding to each transition (deleting branches, raising exceptions, and so on), and will then move MergeRequest to the next state.

Since `MergeRequest` delegates all actions to its state object, we will find that this typically happens every time the actions that it needs to do are in the form `self.state.open()`, and so on. Can we remove some of that boilerplate?

We could, by means of `__getattr__()`, as it is portrayed in the following code:

```
class MergeRequest:
    def __init__(self, source_branch: str, target_branch: str) -> None:
        self.source_branch = source_branch
        self.target_branch = target_branch
        self._state: MergeRequestState
        self.approvals = 0
        self.state = Open

    @property
    def state(self):
        return self._state

    @state.setter
    def state(self, new_state_cls):
        self._state = new_state_cls(self)

    @property
    def status(self):
        return str(self.state)

    def __getattr__(self, method):
        return getattr(self.state, method)

    def __str__(self):
        return f"{self.target_branch}:{self.source_branch}"
```

On the one hand, it is good that we reuse some code and remove repetitive lines. This gives the abstract base class even more sense. Somewhere, we want to have all possible actions documented, listed in a single place. That place used to be the `MergeRequest` class, but now those methods are gone, so the only remaining source of that truth is in `MergeRequestState`. Luckily, the type annotation on the state attribute is really helpful for users to know where to look for the interface definition.

A user can simply take a look and see that everything that `MergeRequest` doesn't have will be asked of its state attribute. From the init definition, the annotation will tell us that this is an object of the `MergeRequestState` type, and by looking at this interface, we will see that we can safely ask for the `open()`, `close()`, and `merge()` methods on it.

22.3 3. The null object pattern

The null object pattern is an idea that relates to the good practices that were mentioned in previously. Here, we are formalizing them, and giving more context and analysis to this idea.

The principle is rather simple: functions or methods must return objects of a consistent type. If this is guaranteed, then clients of our code can use the objects that are returned with polymorphism, without having to run extra checks on them.

In the previous examples, we explored how the dynamic nature of Python made things easier for most design patterns. In some cases, they disappear entirely, and in others, they are much easier to implement. The main goal of design patterns as they were originally thought of is that methods or functions should not explicitly name the class of the object that they need in order to work. For this reason, they propose the creation of interfaces and a way of rearranging the objects to make them fit these interfaces in order to modify the design. But most of the

time, this is not needed in Python, and we can just pass different objects, and as long as they respect the methods they must have, then the solution will work.

On the other hand, the fact that objects don't necessarily have to comply with an interface requires us to be more careful as to the things that are returning from such methods and functions. In the same way that our functions didn't make any assumptions about what they were receiving, it's fair to assume that clients of our code will not make any assumptions either (it is our responsibility to provide objects that are compatible). This can be enforced or validated with design by contract. Here, we will explore a simple pattern that will help us avoid these kinds of problems.

Consider the chain or responsibility design pattern explored in the previous section. We saw how flexible it is and its many advantages, such as decoupling responsibilities into smaller objects. One of the problems it has is that we never actually know what object will end up processing the message, if any. In particular, in our example, if there was no suitable object to process the log line, then the method would simply return `None`.

We don't know how users will use the data we passed, but we do know that they are expecting a dictionary. Therefore, the following error might occur:

```
AttributeError: 'NoneType' object has no attribute 'keys'
```

In this case, the fix is rather simple: the default value of the `process()` method should be an empty dictionary rather than `None`.

Important: Ensure that you return objects of a consistent type.

But what if the method didn't return a dictionary, but a custom object of our domain?

To solve this problem, we should have a class that represents the empty state for that object and return it. If we have a class that represents users in our system, and a function that queries users by their ID, then in the case that a user is not found, it should do one of the following two things:

- Raise an exception
- Return an object of the `UserUnknown` type

But in no case should it return `None`. The phrase `None` doesn't represent what just happened, and the caller might legitimately try to ask methods to it, and it will fail with `AttributeError`.

We have discussed exceptions and their pros and cons earlier on, so we should mention that this `null` object should just have the same methods as the original user and do nothing for each one of them.

The advantage of using this structure is that not only are we avoiding an error at runtime but also that this object might be useful. It could make the code easier to test, and it can even, for instance, help in debugging (maybe we could put logging into the methods to understand why that state was reached, what data was provided to it, and so on).

By exploiting almost all of the magic methods of Python, it would be possible to create a generic null object that does absolutely nothing, no matter how it is called, but which can be called from almost any client. Such an object would slightly resemble a Mock object. It is not advisable to go down that path because of the following reasons:

- It loses meaning with the domain problem. Back in our example, having an object of the `UnknownUser` type makes sense, and gives the caller a clear idea that something went wrong with the query.
- It doesn't respect the original interface. This is problematic. Remember that the point is that an `UnknownUser` is a user, and therefore it must have the same methods. If the caller accidentally asks for a method that is not there, then, in that case, it should raise an `AttributeError` exception, and that would be good. With the generic `null` object that can do anything and respond to anything, we would be losing this information, and bugs might creep in. If we opt for creating a `Mock` object with `spec=User`, then this anomaly would be caught, but again, using a `Mock` object to represent what is actually an empty state harms the intention revealing the degree of the code.

Note: This pattern is a good practice that allows us to maintain polymorphism in our objects.

22.4 4. Final thoughts about design patterns

We have seen the world of design patterns in Python, and in doing so, we have found solutions to common problems, as well as more techniques that will help us achieve a clean design.

All of this sounds good, but it begs the question, how good are design patterns? Some people argue that they do more harm than good, that they were created for languages whose limited type system (and lack of first-class functions) makes it impossible to accomplish things we would normally do in Python. Others claim that design patterns force a design solution, creating some bias that limits a design that would have otherwise emerged, and which would have been better. Let's look at each of these points in turn.

22.4.1 4.1. The influence of patterns over the design

A design pattern, as with any other topic in software engineering, cannot be good or bad in and of itself, but rather in how it's implemented. In some cases, there is actually no need for a design pattern, and a simpler solution would do. Trying to force a pattern where it doesn't fit is a case of over-engineering, and that's clearly bad, but it doesn't mean that there is a problem with the design patterns, and most likely in these scenarios, the problem is not even related to patterns at all. Some people try to over-engineer everything because they don't understand what flexible and adaptable software really means. As we mentioned before, making good software is not about anticipating future requirements (there is no point in doing futurology), but just solving the problem that we have at hand right now, in a way that doesn't prevent us from making changes to it in the future. It doesn't have to handle those changes now; it just needs to be flexible enough so that it can be modified in the future. And when that future comes, we will still have to remember the rule of three or more instances of the same problem before coming up with a generic solution or a proper abstraction.

This is typically the point where the design patterns should emerge, once we have identified the problem correctly and are able to recognize the pattern and abstract accordingly.

Let's come back to the topic of the suitability of the patterns to the language. As we said in the introduction of the chapter, design patterns are high-level ideas. They typically refer to the relation of objects and their interactions. It's hard to think that such things might disappear from one language to another. It's true that some patterns are actually implemented manually in Python, as is the case of the iterator pattern (which, as it was heavily discussed earlier, is built in Python), or a strategy (because, instead, we would just pass functions as any other regular object; we don't need to encapsulate the strategy method into an object the function itself would be an object).

But other patterns are actually needed, and they indeed solve problems, as in the case of the decorator and composite patterns. In other cases, there are design patterns that Python itself implements, and we just don't always see them, as in the case of the facade pattern that we discussed in the section on `os`.

As to our design patterns leading our solution in a wrong direction, we have to be careful here. Once again, it's better if we start designing our solution by thinking in terms of the domain problem and creating the right abstractions, and then later see whether there is a design pattern that emerges from that design. Let's say that it does. Is that a bad thing? The fact that there is already a solution to the problem we're trying to solve cannot be a bad thing. It would be bad to reinvent the wheel, as happens many times in our field. Moreover, the fact that we are applying a pattern, something already proven and validated, should give us greater confidence in the quality of what we are building.

22.4.2 4.2. Names in our models

Should we mention that we are using a design pattern in our code?

If the design is good and the code is clean, it should speak for itself. It is not recommended that you name things after the design patterns you are using for a couple of reasons:

- Users of our code and other developers don't need to know the design pattern behind the code, as long as it works as intended.
- Stating the design pattern ruins the intention revealing principle. Adding the name of the design pattern to a class makes it lose part of its original meaning. If a class represents a query, it should be named `Query` or `EnhancedQuery`, something that reveals the intention of what that object is supposed to do.

`EnhancedQueryDecorator` doesn't mean anything meaningful, and the `Decorator` suffix creates more confusion than clarity.

Mentioning the design patterns in docstrings might be acceptable because they work as documentation, and expressing the design ideas (again, communicating) in our design is a good thing. However, this should not be needed. Most of the time, though, we do not need to know that a design pattern is there.

The best designs are those in which design patterns are completely transparent to the users. An example of this is how the facade pattern appears in the standard library, making it completely transparent to users as to how to access the os module. An even more elegant example is how the iterator design pattern is so completely abstracted by the language that we don't even have to think about it.

CHAPTER

TWENTYTHREE

CLEAN ARCHITECTURE

In this final chapter, we focus on how everything fits together in the design of a whole system. This is a more theoretical chapter. Given the nature of the topic, it would be too complex to delve down into the more low-level details. Besides, the point is precisely to escape from those details, assume that all the principles explored in previous chapters are assimilated, and focus on the design of a system at scale.

23.1 1. From clean code to clean architecture

This section is a discussion of how concepts that were emphasized in previous chapters reappear in a slightly different shape when we consider aspects of large systems. There is an interesting resemblance to how concepts that apply to more detailed design, as well as code, also apply to large systems and architectures.

The concepts explored in previously were related to single applications, generally, a project, that might be a single repository (or a few), for a source control version system (git). This is not to say that those design ideas are only applicable to code, or that they are of no use when thinking of an architecture, for two reasons: the code is the foundation of the architecture, and, if it's not written carefully, the system will fail regardless of how well thought-out the architecture is.

Second, some principles that were revisited in previous chapters do not apply to code but are instead design ideas. The clearest example comes from design patterns. They are high- level ideas. With this, we can get a quick picture of how a component in our architecture might appear, without going into the details of the code.

But large enterprise systems typically consist of many of these applications, and now it's time to start thinking in terms of a larger design, in the form of a distributed system.

In the following sections, we discuss the main topics that have been already discussed, but now from the perspective of a system.

23.1.1 1.1. Separation of concerns

Inside an application, there are multiple components. Their code is divided into other subcomponents, such as modules or packages, and the modules into classes or functions, and the classes into methods. The emphasis has been on keeping these components as small as possible, particularly in the case of functions, they should do one thing, and be small.

Several reasons were presented to justify this rationale. Small functions are easier to understand, follow, and debug. They are also easier to test. The smaller the pieces in our code, the easier it will be to write unit tests for it.

For the components of each application, we wanted different traits, mainly high cohesion, and low coupling. By dividing components into smaller units, each one with a single and well-defined responsibility, we achieve a better structure where changes are easier to manage. In the face of new requirements, there will be a single rightful place to make the changes, and the rest of the code should probably be unaffected.

When we talk about code, we say component to refer to one of these cohesive units (it might be a class, for example). When speaking in terms of an architecture, a component means anything in the system that can be treated as a working unit. The term component itself is quite vague, so there is no universally accepted definition in software architecture of what this means more concretely. The concept of a working unit is something that

can vary from project to project. A component should be able to be released or deployed with its own cycles, independently from the rest of the parts of the system. And it is precisely that, one of the parts of a system, is namely the entire application.

For Python projects, a component could be a package, but a service can also be a component. Notice how two different concepts, with different levels of granularity, can be considered under the same category. To give an example, the event systems we used in previous chapters could be considered a component. It's a working unit with a clearly defined purpose (to enrich events identified from logs), it can be deployed independently from the rest (whether as a Python package, or, if we expose its functionality, as a service), and it's a part of the entire system, but not the whole application itself.

On the examples of previous chapters we have seen an idiomatic code, and we have also highlighted the importance of good design for our code, with objects that have single well- defined responsibilities, being isolated, orthogonal, and easier to maintain. This very same criteria, which applies to detailed design (functions, classes, methods), also applies to the components of a software architecture.

It's probably undesirable for a large system to be just one component. A monolithic application will act as the single source of truth, responsible for everything in the system, and that will carry a lot of undesired consequences (harder to isolate and identify changes, to test effectively, and so on). In the same way, our code will be harder to maintain, if we are not careful and place everything in one place, the application will suffer from similar problems if its components aren't treated with the same level of attention.

The idea of creating cohesive components in a system can have more than one implementation, depending on the level of abstraction we require.

One option would be to identify common logic that is likely to be reused multiple times and place it in a Python package (we will discuss the details later in the chapter). Another alternative would be to break the application into multiple smaller services, in a microservice architecture. The idea is to have components with a single and well-defined responsibility, and achieve the same functionality as a monolithic application by making those services cooperate, and exchange information.

23.1.2 1.2. Abstractions

This is where encapsulation appears again. From our systems (as we do in relation to the code), we want to speak in terms of the domain problem, and leave the implementation details as hidden as possible.

In the same way that the code has to be expressive (almost to the point of being self- documenting), and have the right abstractions that reveal the solution to the essential problem (minimizing accidental complexity), the architecture should tell us what the system is about. Details such as the solution used to persist data on disk, the web framework of choice, the libraries used to connect to external agents, and interaction between systems, are not relevant. What is relevant is what the system does. A concept such as a scream architecture (SCREAM) reflects this idea.

The **dependency inversion principle (DIP)** is of great help in this regard; we don't want to depend upon concrete implementations but rather abstractions. In the code, we place abstractions (or interfaces) on the boundaries, the dependencies, those parts of the application that we don't control and might change in the future. We do this because we want to invert the dependencies. Let them have to adapt to our code (by having to comply with an interface), and not the other way round.

Creating abstractions and inverting dependencies are good practices, but they're not enough. We want our entire application to be independent and isolated from things that are out of our control. And this is even more than just abstracting with objects—we need layers of abstraction.

This is a subtle, but important difference with respect to the detailed design. In the DIP, it was recommended to create an interface, that could be implemented with the abc module from the standard library, for instance. Because Python works with duck typing, while using an abstract class might be helpful, it's not mandatory, as we can easily achieve the same effect with regular objects as long as they comply with the required interface. The dynamic typing nature of Python allowed us to have these alternatives. When thinking in terms of architecture, there is no such a thing. As it will become clearer with the example, we need to abstract dependencies entirely, and there is no feature of Python that can do that for us.

Some might argue "Well, the ORM is a good abstraction for a database, isn't it?" It's not enough. The ORM itself is a dependency and, as such, out of our control. It would be even better to create an intermediate layer, an adapter, between the API of the ORM and our application.

This means that we don't abstract the database just with an ORM; we use the abstraction layer we create on top of it, to define objects of our own that belong to our domain. The application then imports this component, and uses the entities provided by this layer, but not the other way round. The abstraction layer should not know about the logic of our application; it's even truer that the database should know nothing about the application itself. If that were the case, the database would be coupled to our application. The goal is to invert the dependency—this layer provides an API, and every storage component that wants to connect has to conform to this API. This is the concept of a hexagonal architecture (HEX).

23.2 2. Software components

We have a large system now, and we need to scale it. It also has to be maintainable. At this point, the concerns aren't only technical but also organizational. This means it's not just about managing software repositories; each repository will most likely belong to an application, and it will be maintained by a team who owns that part of the system.

This demands we keep in mind how a large system is divided into different components. This can have many phases, from a very simple approach about, say, creating Python packages, to more complex scenarios in a microservice architecture.

The situation could be even more complex when different languages are involved, but in this chapter, we will assume they are all Python projects.

These components need to interact, as do the teams. The only way this can work at scale is if all the parts agree on an interface, a contract.

23.2.1 2.1. Packages

A Python package is a convenient way to distribute software and reuse code in a more general way. Packages that have been built can be published to an artifact repository (such as an internal PyPi server for the company), from where it will be downloaded by the rest of the applications that require it.

The motivation behind this approach has many elements to it: it's about reusing code at large, and also achieving conceptual integrity.

Here, we discuss the basics of packaging a Python project that can be published in a repository. The default repository might be PyPi, but also internal; or custom setups will work with the same basics.

We are going to simulate that we have created a small library, and we will use that as an example to review the main points to take into consideration.

Aside from all the open source libraries available, sometimes we might need some extra functionality: perhaps our application uses a particular idiom repeatedly or relies on a function or mechanism quite heavily and the team has devised a better function for these particular needs. In order to work more effectively, we can place this abstraction into a library, and encourage all team members to use the idioms as provided by it, because doing so will help avoid mistakes and reduce bugs.

Potentially, there are infinite examples that could suit this scenario. Maybe the application needs to extract a lot of .tag.gz files (in a particular format) and has faced security problems in the past with malicious files that ended up with path traversal attacks. As a mitigation measure, the functionality for abstracting custom file formats securely was put in a library that wraps the default one and adds some extra checks. This sounds like a good idea.

Or maybe there is a configuration file that has to be written, or parsed in a particular format, and this requires many steps to be followed in order; again, creating a helper function to wrap this, and using it in all the projects that need it, constitutes a good investment, not only because it saves a lot of code repetition, but also because it makes it harder to make mistakes.

The gain is not only complying with the DRY principle (avoiding code duplication, encouraging reuse) but also that the abstracted functionality represents a single point of reference of how things should be done, hence contributing to the attainment of conceptual integrity.

In general, the minimum layout for a library would look like this:

```
.
├── Makefile
├── README.rst
├── setup.py
├── src
│   ├── apptool
│   ├── common.py
│   ├── __init__.py
│   └── parse.py
└── tests
    ├── integration
    └── unit
```

The important part is the `setup.py` file, which contains the definition for the package. In this file, all the important definitions of the project (its requirements, dependencies, name, description, and so on) are specified.

The `apptool` directory under `src` is the name of the library we're working on. This is a typical Python project, so we place here all the files we need.

An example of the setup.py file could be:

```
from setuptools import find_packages, setup


with open("README.rst", "r") as longdesc:
    long_description = longdesc.read()

setup(
    name="apptool",
    description="Description of the intention of the package",
    long_description=long_description,
    author="Dev team",
    version="0.1.0",
    packages=find_packages(where="src/"),
    package_dir={"": "src"},
)
```

This minimal example contains the key elements of the project. The name argument in the setup function is used to give the name that the package will have in the repository (under this name, we run the command to install it, in this case its `pip install apptool`). It's not strictly required that it matches the name of the project directory (`src/apptool`), but it's highly recommended, so its easier for users.

The version is important to keep different releases going on, and then the packages are specified. By using the `find_packages()` function, we automatically discover everything that's a package, in this case under the `src/` directory. Searching under this directory helps to avoid mixing up files beyond the scope of the project and, for instance, accidentally releasing tests or a broken structure of the project.

A package is built by running the following commands, assuming its run inside a virtual environment with the dependencies installed:

```
$VIRTUAL_ENV/bin/pip install -U setuptools wheel
$VIRTUAL_ENV/bin/python setup.py sdist bdist_wheel
```

This will place the artifacts in the `dist/` directory, from where they can be later published either to PyPi or to the internal package repository of the company.

The key points in packaging a Python project are:

- Test and verify that the installation is platform-independent and that it doesn't rely on any local setup (this can be achieved by placing the source files under an `src/` directory)
- Make sure that unit tests aren't shipped as part of the package being built
- Separate dependencies: what the project strictly needs to run is not the same as what developers require
- It's a good idea to create entry points for the commands that are going to be required the most

The `setup.py` file supports multiple other parameters and configurations and can be effected in a much more complicated manner. If our package requires several operating system libraries to be installed, it's a good idea to write some logic in the `setup.py` file to compile and build the extensions that are required. This way, if something is amiss, it will fail early on in the installation process, and if the package provides a helpful error message, the user will be able to fix the dependencies more quickly and continue.

Installing such dependencies represents another difficult step in making the application ubiquitous, and easy to run by any developer regardless of their platform of choice. The best way to surmount this obstacle is to abstract the platform by creating a Docker image.

23.2.2 2.2. Containers

This chapter is dedicated to architecture, so the term container refers to something completely different from a Python container (an object with a `__contains__` method). A container is a process that runs in the operating system under a group with certain restrictions and isolation considerations. Concretely we refer to Docker containers, which allow managing applications (services or processes) as independent components.

Containers represent another way of delivering software. Creating Python packages taking into account the considerations in the previous section is more suitable for libraries, or frameworks, where the goal is to reuse code and take advantage of using a single place where specific logic is gathered.

In the case of containers, the objective will not be creating libraries but applications (most of the time). However, an application or platform does not necessarily mean an entire service. The idea of building containers is to create small components that represent a service with a small and clear purpose.

In this section, we will mention Docker when we talk about containers, and we will explore the basics of how to create Docker images and containers for Python projects. Keep in mind that this is not the only technology for launching applications into containers, and also that it's completely independent of Python.

A Docker container needs an image to run on, and this image is created from other base images. But the images we create can themselves serve as base images for other containers. We will want to do that in cases where there is a common base in our application that can be shared across many containers. A potential use would be creating a base image that installs a package (or many) in the way we described in the previous section, and also all of its dependencies, including those at the operating system level. A package we create can depend not only on other Python libraries, but also on a particular platform (a specific operating system), and particular libraries preinstalled in that operating system, without which the package will simply not install and will fail.

Containers are a great portability tool for this. They can help us ensure that our application will have a canonical way of running, and it will also ease the development process a lot (reproducing scenarios across environments, replicating tests, on-boarding new team members, and so on).

As packages are the way we reuse code and unify criteria, containers represent the way we create the different services of the application. They meet the criteria behind the principle of separation of concerns (SoC) of the architecture. Each service is another kind of component that will encapsulate a set of functionalities independently of the rest of the application. These containers ought to be designed in such a way that they favor maintainability: if the responsibilities are clearly divided, a change in a service should not impact any other part of the application whatsoever.

23.3 3. Use case

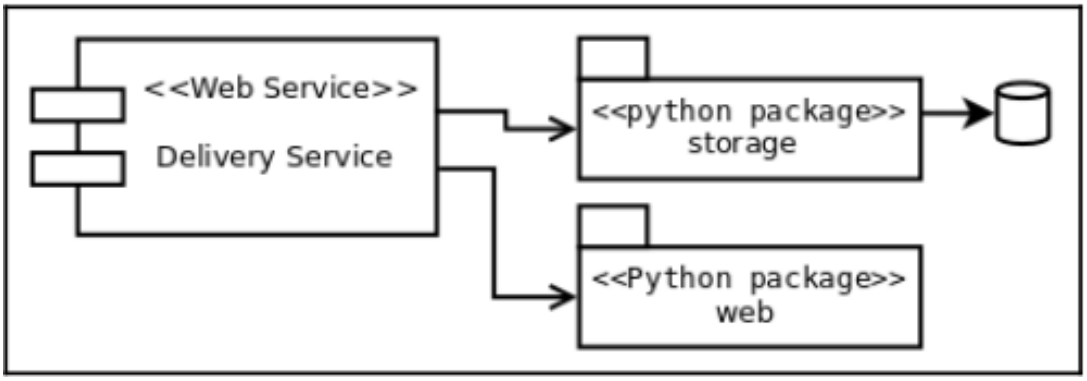
As an example of how we might organize the components of our application, and how the previous concepts might work in practice, we present the following simple example.

The use case is that there is an application for delivering food, and this application has a specific service for tracking the status of each delivery at its different stages. We are going to focus only on this particular service, regardless of how the rest of the application might appear. The service has to be really simple: a REST API that, when asked about the status of a particular order, will return a JSON response with a descriptive message.

We are going to assume that the information about each particular order is stored in a database, but this detail should not matter at all.

Our service has two main concerns for now: getting the information about a particular order (from wherever this might be stored), and presenting this information in a useful way to the clients (in this case, delivering the results in JSON format, exposed as a web service).

As the application has to be maintainable and extensible, we want to keep these two concerns as hidden as possible and focus on the main logic. Therefore, these two details are abstracted and encapsulated into Python packages that the main application with the core logic will use, as shown in the following diagram:



23.3.1 3.1. The code

The idea of creating Python packages in this example is to illustrate how abstracted and isolated components can be made, in order to work effectively. In reality, there is no actual need for these to be Python packages; we could just create the right abstractions as part of the "delivery service" project, and, while the correct isolation is preserved, it will work without any issues.

Creating packages makes more sense when there is logic that is going to be repeated and is expected to be used across many other applications (that will import from those packages) because we want to favor code reuse. In this particular case, there are no such requirements, so it might be beyond the scope of the design, but such distinction still makes more clear the idea of a "pluggable architecture" or component, something that is really a wrapper abstracting technical details we don't really want to deal with, much less depend upon.

The `storage` package is in charge of retrieving the data that is required, and presenting this to the next layer (the delivery service) in a convenient format, something that is suitable for the business rules. The main application should now know where this data came from, what its format is, and so on. This is the entire reason why we have such an abstraction in between so the application doesn't use a row or an ORM entity directly, but rather something workable.

3.1.1. Domain models

The following definitions apply to classes for business rules. Notice that they are meant to be pure business objects, not bound to anything in particular. They aren't models of an ORM, or objects of an external framework, and so on. The application should work with these objects (or objects with the same criteria).

In each case, the dosctring documents the purpose of each class, according to the business rule:

```
from typing import Union


class DispatchedOrder:
    """An order that was just created and notified to start its delivery."""

    status = "dispatched"

    def __init__(self, when):
        self._when = when

    def message(self) -> dict:
        return {
            "status": self.status,
            "msg": f"Order was dispatched on {self._when.isoformat()}"
        }


class OrderInTransit:
    """An order that is currently being sent to the customer."""

    status = "in transit"

    def __init__(self, current_location):
        self._current_location = current_location

    def message(self) -> dict:
        return {
            "status": self.status,
            "msg": f"The order is in progress (current location: {self._current_
↪location})"
        }


class OrderDelivered:
    """An order that was already delivered to the customer."""

    status = "delivered"

    def __init__(self, delivered_at):
        self._delivered_at = delivered_at

    def message(self) -> dict:
        return {
            "status": self.status,
            "msg": f"Order delivered on {self._delivered_at.isoformat()}"
        }

class DeliveryOrder:
    def __init__(self, delivery_id: str, status: Union[DispatchedOrder,␣
↪OrderInTransit, OrderDelivered]) -> None:
        self._delivery_id = delivery_id
        self._status = status

    def message(self) -> dict:
```

(continues on next page)

(continued from previous page)

```
        return {"id": self._delivery_id, **self._status.message()}
```

From this code, we can already get an idea of what the application will look like: we want to have a `DeliveryOrder` object, which will have its own status (as an internal collaborator), and once we have that, we will call its `message()` method to return this information to the user.

3.1.2. Calling from the application

Here is how these objects are going to be used in the application. Notice how this depends on the previous packages (web and storage), but not the other way round:

```
from storage import DBClient, DeliveryStatusQuery, OrderNotFoundError
from web import NotFound, View, app, register_route


class DeliveryView(View):
    async def _get(self, request, delivery_id: int):
        dsq = DeliveryStatusQuery(int(delivery_id), await DBClient())
        try:
            result = await dsq.get()
        except OrderNotFoundError as e:
            raise NotFound(str(e)) from e
        return result.message()

register_route(DeliveryView, "/status/<delivery_id:int>")
```

In the previous section, the domain objects were shown and here the code for the application is displayed. Aren't we missing something? Sure, but is it something we really need to know now? Not necessarily.

The code inside the storage and web packages was deliberately left out. Also, and this was done on purpose, the names of such packages were chosen so as not to reveal any technical detail: `storage` and `web`.

Look again at the code in the previous listing. Can you tell which frameworks are being used? Does it say whether the data comes from a text file, a database (if so, of what type? SQL? NoSQL?), or another service (the web, for instance)? Assume that it comes from a relational database. Is there any clue to how this information is retrieved? Manual SQL queries? Through an ORM? What about the web? Can we guess what frameworks are used?

The fact that we cannot answer any of those questions is probably a good sign. Those are details, and details ought to be encapsulated. We can't answer those questions unless we take a look at what's inside those packages.

There is another way of answering the previous questions, and it comes in the form of a question itself: why do we need to know that? Looking at the code, we can see that there is a `DeliveryOrder`, created with an identifier of a delivery, and that it has a `get()` method, which returns an object representing the status of the delivery. If all of this information is correct, that's all we should care about. What difference does it make how is it done?

The abstractions we created make our code declarative. In declarative programming, we declare the problem we want to solve, not how we want to solve it. It's the opposite of imperative, in which we have to make all the steps required explicit in order to get something (for instance connect to the database, run this query, parse the result, load it into this object, and so on). In this case, we are declaring that we just want to know the status of the delivery given by some identifier.

These packages are in charge of dealing with the details and presenting what the application needs in a convenient format, namely objects of the kind presented in the previous section. We just have to know that the storage package contains an object that, given an ID for a delivery and a storage client (this dependency is being injected into this example for simplicity, but other alternatives are also possible), it will retrieve `DeliveryOrder` which we can then ask to compose the message.

This architecture provides convenience and makes it easier to adapt to changes, as it protects the kernel of the business logic from the external factors that can change.

Imagine we want to change how the information is retrieved. How hard would that be? The application relies on an API, like the following one:

```
dsq = DeliveryStatusQuery(int(delivery_id), await DBClient())
```

So it would be about just changing how the `get()` method works, adapting it to the new implementation detail. All we need is for this new object to return `DeliveryOrder` on its `get()` method and that would be all. We can change the query, the ORM, the database, and so on, and, in all cases, the code in the application does not need to change!

3.1.3. Adapters

Still, without looking at the code in the packages, we can conclude that they work as interfaces for the technical details of the application.

In fact, since we are seeing the application from a high-level perspective, without needing to look at the code, we can imagine that inside those packages there must be an implementation of the adapter design pattern. One or more of these objects is adapting an external implementation to the API defined by the application. This way, dependencies that want to work with the application must conform to the API, and an adapter will have to be made.

Notice how the view is constructed. It inherits from a class named `View` that comes from our `web` package. We can deduce that this `View` is, in turn, a class derived from one of the web frameworks that might be being used, creating an adapter by inheritance. The important thing to note is that once this is done, the only object that matters is our `View` class, because, in a way, we are creating our own framework, which is based on adapting an existing one (but again changing the framework will mean just changing the adapters, not the entire application).

23.3.2 3.2. The services

To create the service, we are going to launch the Python application inside a Docker container. Starting from a base image, the container will have to install the dependencies for the application to run, which also has dependencies at the operating system level.

This is actually a choice because it depends on how the dependencies are used. If a package we use requires other libraries on the operating system to compile at installation time, we can avoid this simply by building a wheel for our platform of the library and installing this directly. If the libraries are needed at runtime, then there is no choice but to make them part of the image of the container.

Now, we discuss one of the many ways of preparing a Python application to be run inside a Docker container. This is one of numerous alternatives for packaging a Python project into a container. First, we take a look at what the structure of the directories looks like:

```
.
├── Dockerfile
├── libs
│   ├── README.rst
│   ├── storage
│   └── web
├── Makefile
├── README.rst
├── setup.py
└── statusweb
    ├── __init__.py
    └── service.py
```

The libs directory can be ignored since it's just the place where the dependencies are placed (it's displayed here to keep them in mind when they are referenced in the setup.py file, but they could be placed in a different repository and installed remotely via `pip`).

We have `Makefile` with some helper commands, then the `setup.py` file, and the application itself inside the `statusweb` directory. A common difference between packaging applications and libraries is that while the latter specify their dependencies in the `setup.py` file, the former has a `requirements.txt` file from where dependencies are installed via `pip install -r requirements.txt`. Normally, we would do this in the

`Dockerfile`, but in order to keep things simpler in this particular example, we will assume that taking the dependencies from the `setup.py` file is enough. This is because, besides this consideration, there are a lot more considerations to be taken into account when dealing with dependencies, such as freezing the version of the packages, tracking indirect dependencies, using extra tools such as `pipenv`, and more topics that are beyond the scope of the chapter. In addition, it is also customary to make the `setup.py` file read from `requirements.txt` for consistency.

Now we have the content of the `setup.py` file, which states some details of the application:

```
from setuptools import find_packages, setup


with open("README.rst", "r") as longdesc:
    long_description = longdesc.read()

install_requires = ["web", "storage"]

setup(
    name="delistatus",
    description="Check the status of a delivery order",
    long_description=long_description,
    author="Dev team",
    version="0.1.0",
    packages=find_packages(),
    install_requires=install_requires,
    entry_points={
        "console_scripts": [
            "status-service = statusweb.service:main",
        ],
    }
)
```

The first thing we notice is that the application declares its dependencies, which are the packages we created and placed under `libs/`, namely `web` and `storage`, abstracting and adapting to some external components. These packages, in turn, will have dependencies, so we will have to make sure the container installs all the required libraries when the image is being created so that they can install successfully, and then this package afterward.

The second thing we notice is the definition of the `entry_points` keyword argument passed to the `setup` function. This is not strictly mandatory, but it's a good idea to create an entry point. When the package is installed in a virtual environment, it shares this directory along with all its dependencies. A virtual environment is a structure of directories with the dependencies of a given project. It has many subdirectories, but the most important ones are:

```
<virtual-env-root>/lib/<python-version>/site-packages
<virtual-env-root>/bin
```

The first one contains all the libraries installed in that virtual environment. If we were to create a virtual environment with this project, that directory would contain the `web`, and `storage` packages, along with all its dependencies, plus some extra basic ones and the current project itself.

The second, `/bin/`, contains the binary files and commands available when that virtual environment is active. By default, it would just be the version of Python, `pip`, and some other basic commands. When we create an entry point, a binary with that declared name is placed there, and, as a result, we have that command available to run when the environment is active. When this command is called, it will run the function that is specified with all the context of the virtual environment. That means it is a binary we can call directly without having to worry about whether the virtual environment is active, or whether the dependencies are installed in the path that is currently running.

The definition is the following one: `"status-service = statusweb.service:main"`

The left-hand side of the equals sign declares the name of the entry point. In this case, we will have a command named `status-service` available. The right-hand side declares how that command should be run. It requires

the package where the function is defined, followed by the function name after `:`. In this case, it will run the main function declared in `statusweb/service.py`.

This is followed by a definition of the `Dockerfile`:

```
FROM python:3.6.6-alpine3.6
RUN apk add --update \
    python-dev \
    gcc \
    musl-dev \
    make
WORKDIR /app
ADD . /app
RUN pip install /app/libs/web /app/libs/storage
RUN pip install /app
EXPOSE 8080
CMD ["/usr/local/bin/status-service"]
```

The image is built based on a lightweight Python image, and then the operating system dependencies are installed so that our libraries can be installed. Following the previous consideration, this Dockerfile simply copies the libraries, but this might as well be installed from a `requirements.txt` file accordingly. After all the `pip install` commands are ready, it copies the application in the working directory, and the entry point from Docker (the `CMD` command, not to be confused with the Python one) calls the entry point of the package where we placed the function that launches the process.

All the configuration is passed by environment variables, so the code for our service will have to comply with this norm.

In a more complex scenario involving more services and dependencies, we will not just run the image of the created container, but instead declare a `docker-compose.yml` file with the definitions of all the services, base images, and how they are linked and interconnected.

Now that we have the container running, we can launch it and run a small test on it to get an idea of how it works:

```
$ curl http://localhost:8080/status/1
{"id":1,"status":"dispatched","msg":"Order was dispatched on
2018-08-01T22:25:12+00:00"}
```

3.3. Analysis

There are many conclusions to be drawn from the previous implementation. While it might seem like a good approach, there are cons that come with the benefits; after all, no architecture or implementation is perfect. This means that a solution such as this one cannot be good for all cases, so it will pretty much depend on the circumstances of the project, the team, the organization, and more.

While it's true that the main idea of the solution is to abstract details as much as possible, as we shall see some parts cannot be fully abstracted away, and also the contracts between the layers imply an abstraction leak.

3.3.1. The dependency flow

Notice that dependencies flow in only one direction, as they move closer to the kernel, where the business rules lie. This can be traced by looking at the import statements. The application imports everything it needs from `storage`, for example, and in no part is this inverted.

Breaking this rule would create coupling. The way the code is arranged now means that there is a weak dependency between the application and `storage`. The API is such that we need an object with a `get()` method, and any storage that wants to connect to the application needs to implement this object according to this specification. The dependencies are therefore inverted: it's up to every storage to implement this interface, in order to create an object according to what the application is expecting.

3.3.2. Limitations

Not everything can be abstracted away. In some cases, it's simply not possible, and in others, it might not be convenient. Let's start with the convenience aspect.

In this example, there is an adapter of the web framework of choice to a clean API to be presented to the application. In a more complex scenario, such a change might not be possible. Even with this abstraction, parts of the library were still visible to the application. Adapting an entire framework might not only be hard but also not possible in some cases. It's not entirely a problem to be completely isolated from the web framework because, sooner or later, we will need some of its features or technical details.

The important takeaway here is not the adapter, but the idea of hiding technical details as much as possible. That means, that the best thing that was displayed on the listing for the code of the application was not the fact that there was an adapter between our version of the web framework and the actual one, but instead the fact that the latter was not mentioned by name in any part of the visible code. The service was made clear that `web` was just a dependency (a detail being imported), and revealed the intention behind what it was supposed to do. The goal is to reveal the intention (as in the code) and to defer details as much as possible.

As to what things cannot be isolated, those are the elements that are closest to the code. In this case, the web application was using the objects operating within them in an asynchronous fashion. That is a hard constraint we cannot circumvent. It's true that whatever is inside the storage package can be changed, refactored, and modified, but whatever these modifications might be, it still needs to preserve the interface, and that includes the asynchronous interface.

3.3.3. Testability

Again, much like with the code, the architecture can benefit from separating pieces into smaller components. The fact that dependencies are now isolated and controlled by separate components leaves us with a cleaner design for the main application, and now it's easier to ignore the boundaries to focus on testing the core of the application.

We could create a patch for the dependencies, and write unit tests that are simpler (they won't need a database), or to launch an entire web service, for instance. Working with pure domain objects means it will be easier to understand the code and the unit tests. Even the adapters will not need that much testing because their logic should be very simple.

3.3.4. Intention revealing

These details included keeping functions short, concerns separated, dependencies isolated, and assigning the right meaning to abstractions in every part of the code. Intention revealing was a critical concept for our code: every name has to be wisely chosen, clearly communicating what it's supposed to do. Every function should tell a story.

A good architecture should reveal the intent of the system it entails. It should not mention the tools it's built with; those are details, and as we discussed at length, details should be hidden, encapsulated.

Part VI

Low level Python

CHAPTER

TWENTYFOUR

HOW DOES PYTHON WORK?

24.1 1. Interpreters

The reference Python interpreter implementation is called CPython and, as its name suggests, it is written entirely in the C language. It was always C and probably will be still for a very long time. That's the implementation that most Python programmers choose because it is always up to date with the language specification and is the interpreter that most libraries are tested on. But, besides C, Python interpreter was written in a few other languages. Also, there are some modified versions of CPython interpreter available under different names and tailored exactly for some niche applications. Most of them are a few milestones behind CPython, but provide a great opportunity to use and promote the language in a specific environment.

24.1.1 1.1. Why should I care?

There are plenty of alternative Python implementations available. The Python wiki page on that topic (https://wiki.python.org/moin/PythonImplementations) features dozens of different language variants, dialects, or implementations of Python interpreter built with something other than C. Some of them implement only a subset of the core language syntax, features, and built-in extensions, but there are at least a few that are almost fully compatible with CPython. The most important thing to know is that, while some of them are just toy projects or experiments, most of them were created to solve some real problems: problems that were either impossible to solve with CPython or required too much of the developer's effort.

Examples of such problems are as follows:

- Running Python code on embedded systems
- Integration with code written for runtime frameworks, such as Java or .NET, or in different languages
- Running Python code in web browsers

The following sections provide a short description of, subjectively, the most popular and up-to-date choices that are currently available for Python programmers.

24.1.2 1.2. Stackless Python

Stackless Python advertises itself as an enhanced version of Python. Stackless is named so because it avoids depending on the C call stack for its own stack. It is, in fact, a modified CPython code that also adds some new features that were missing from the core Python implementation at the time Stackless was created. The most important of these are microthreads, which are managed by the interpreter as cheap and lightweight alternatives to ordinary threads, that must depend on system kernel context switching and task scheduling.

The latest available versions are 2.7.15 and 3.6.6 and implement 2.7 and 3.6 versions of Python, respectively. All the additional features provided by Stackless are exposed as a framework within this distribution through the built-in stackless module.

Stackless isn't the most popular alternative implementation of Python, but it is worth knowing, because some of the ideas that were introduced in it had a strong impact on the language community. The core switching functionality was extracted from Stackless and published as an independent package named greenlet, which is now the basis for

many useful libraries and frameworks. Also, most of its features were re-implemented in PyPy: another Python implementation that will be featured later. The official online documentation of Stackless Python can be found at https://stackless.readthedocs.io and the project wiki can be found at https://github.com/stackless-dev/stackless.

24.1.3 1.3. Jython

Jython is a Java implementation of the language. It compiles the code into Java byte code, and allows the developers to seamlessly use Java classes within their Python modules. Jython allows people to use Python as the top-level scripting language on complex application systems, for example, J2EE. It also brings Java applications into the Python world. Making Apache Jackrabbit (which is a document repository API based on JCR; see http://jackrabbit.apache.org available in a Python program is a good example of what Jython allows.

The main differences of Jython compared to the CPython implementation are as follows:

- True Java's garbage collection instead of reference counting
- Lack of global interpreter lock (GIL) allows better utilization of multiple cores in multi-threaded applications

The main weakness of this implementation of the language is the lack of support for C Python Extension APIs, so no Python extensions written in C will work with Jython.

The latest available version of Jython is Jython 2.7, and this corresponds to the 2.7 version of the language. It is advertised as implementing nearly all of the core Python standard library and using the same regression test suite. Unfortunately, Jython 3.x was never released, and the project can be now safely considered dead. However, Jython is still worth mentioning, even if it is not developed anymore, because it was very unique implementation at the time and had meaningful impact on other alternative Python implementations.

The official project page can be found at http://www.jython.org.

24.1.4 1.4. IronPython

IronPython brings Python into the .NET Framework. The project is supported by Microsoft, where IronPython's lead developers work. It is quite an important implementation for the promotion of a language. Besides Java, the .NET community is one of the biggest developer communities out there. It is also worth noting that Microsoft provides a set of free development tools that turn Visual Studio into a full-fledged Python IDE. This is distributed as Visual Studio plugins named Python Tools for Visual Studio (PVTS), and is available as open source code on GitHub (http://microsoft.github.io/PTVS).

The latest stable release is 2.7.8, and it is compatible with Python 2.7. Unlike Jython, we can observe active development on both 2.x and 3.x branches of the interpreter, although Python 3 support still hasn't been officially released yet. Despite the fact that .NET runs primarily on Microsoft Windows, it is also possible to run IronPython on macOS and Linux. This is thanks to Mono, a cross platform, open source .NET implementation.

The main differences and advantages of IronPython compared to CPython are as follows:

- Similar to Jython, the lack of global interpreter lock (GIL) allows for better utilization of multiple cores in multi-threaded applications
- Code written in C# and other .NET languages can be easily integrated in IronPython and vice versa
- It can be run in all major web browsers through Silverlight (although Microsoft will stop supporting Silverlight in 2021)

When speaking about weaknesses, IronPython seems very similar to Jython, because it does not support the Python/C Extension APIs. This is important for developers who would like to use packages such as NumPy, which are largely based on C extensions. There were a few community attempts to bring the Python/C Extensions API support to IronPython, or at least to provide compatibility for the NumPy package, but unfortunately no project had notable success in that area.

You can learn more about IronPython from its official project page at http://ironpython.net/.

24.1.5 1.5. PyPy

PyPy is probably the most exciting alternative implementation of Python, as its goal is to rewrite Python in Python. PyPy in the Python interpreter is written in Python. We have a C code layer carrying out the nuts-and-bolts work for CPython. But in PyPy, this C code layer is rewritten in pure Python.

This means that you can change the interpreter's behavior during execution time, and implement code patterns that couldn't be easily done in CPython.

PyPy is currently fully compatible with Python version 2.7.13, while the latest PyPy3 is compatible with Python version 3.5.3.

In the past, PyPy was mostly interesting for theoretical reasons, and it interested those who enjoyed going deep into the details of the language. It was not generally used in production, but this has changed through the years. Nowadays, many benchmarks show that, surprisingly, PyPy is often way faster than the CPython implementation. This project has its own benchmarking site that tracks performance of each version measured using dozens of different benchmarks (refer to http://speed.pypy.org/). It clearly shows that PyPy with JIT enabled is usually at least few times faster than CPython. This and other features of PyPy makes more and more developers decide to use PyPy in their production environments.

The main differences of PyPy compared to CPython implementation are as follows:

- Garbage collection used instead of reference counting
- Integrated tracing JIT compiler that allows impressive improvements in performance
- Application-level Stackless features borrowed from Stackless Python

Like almost every other alternative Python implementation, PyPy lacks the full official support of C's Python Extension API. Still, it at least provides some sort of support for C extensions through its CPyExt subsystem, although it is poorly documented and still not feature complete. Also, there is an ongoing effort within the community in porting NumPy to PyPy because it is the most requested feature.

The official PyPy project page can be found at http://pypy.org.

24.1.6 1.6. MicroPython

MicroPython is one of the youngest alternative implementations on that list, as its first official version was released on May 3, 2014. It is also one of the most interesting implementations. MicroPython is a Python interpreter that was optimized for use on microcontrollers and in very constrained environments. Its small size and multiple optimizations allow it to run in just 256 kilobytes of code space and with just 16 kilobytes of RAM.

The main reference devices that you can test this interpreter on are BBC's micro:bit devices and pyboards, which are simple-to-use microcontroller development boards, that are targeted at teaching programming and electronics.

MicroPython is written in C99 (it's C language standard) and can be built for many hardware architectures, including x86, x86-64, ARM, ARM Thumb, and Xtensa. It is based on Python 3, but due to many syntax differences, it's impossible to say that it is fully compatible with any Python 3.x release. It is certainly a dialect of Python 3, with `print()` functions, `async`/`await` keywords, and many other Python 3 features, but you can't expect that your favorite Python 3 libraries will work properly under that interpreter out of the box.

You can learn more about MicroPython from its official project page at https://micropython.org.

CHAPTER

TWENTYFIVE

MODERN PYTHON DEVELOPMENT ENVIRONMENTS

A deep understanding of the programming language of choice is the most important thing in being an expert. This will always be true for any technology. Still, it is really hard to develop good software without knowing the common tools and practices that are common within the given language community. Python has no single feature that cannot be found in some other language. So, in direct comparison of syntax, expressiveness, or performance, there will always be a solution that is better in one or more fields. But, the area in which Python really stands out from the crowd is the whole ecosystem built around the language. The Python community spent years polishing standard practices and libraries that help to create more reliable software in a shorter time.

The most obvious and important part of the ecosystem is a huge collection of free and open source packages that solve a multitude of problems. Writing new software is always an expensive and time-consuming process. Being able to reuse the existing code instead of reinventing the wheel greatly reduces development times and costs. For some companies, it is the only reason why their projects are economically feasible.

Because of this, Python developers put a lot of effort into creating tools and standards to work with open source packages that have been created by others: starting from virtual isolated environments, improved interactive shells, and debuggers, to programs that help to discover, search, and analyze the huge collection of packages that are available on Python Package Index (PyPI).

25.1 1. Installing packages with pip

Nowadays, a lot of operating systems come with Python as a standard component. Most Linux distributions and UNIX-based systems, such as FreeBSD, NetBSD, OpenBSD, or macOS, come with Python either installed by default or available through system package repositories. Many of them even use it as part of some core components: Python powers the installers of Ubuntu (Ubiquity), Red Hat Linux (Anaconda), and Fedora (Anaconda again). Unfortunately, the preinstalled system version of Python is often Python 2.7, which is fairly outdated.

Due to Python's popularity as an operating system component, a lot of packages from PyPI are also available as native packages managed by the system's package management tools, such as apt-get (Debian, Ubuntu), rpm (Red Hat Linux), or emerge (Gentoo). It should be remembered, however, that the list of available libraries is very limited, and they are mostly outdated compared to PyPI. This is the reason why pip should always be used to obtain new packages in the latest version, as recommended by the Python Packaging Authority (PyPA). Although it is an independent package, starting from version 2.7.9 and 3.4 of CPython, it is bundled with every new release by default. Installing the new package is as simple as this:

```
pip install <package-name>
```

Among other features, pip allows specific versions of packages to be forced (using the `pip install package-name==version` syntax) and upgraded to the latest version available (using the `--upgrade` switch). The full usage description for most of the command-line tools presented in the book can be easily obtained simply by running the command with the `-h` or `--help` switch, but here is an example session that demonstrates the most commonly used options:

```
$ pip show pip
Name: pip
Version: 18.0
```

(continues on next page)

(continued from previous page)

```
Summary: The PyPA recommended tool for installing Python packages.
Home-page: https://pip.pypa.io/
Author: The pip developers
Author-email: pypa-dev@groups.google.com
License: MIT
Location: /Users/swistakm/.envs/epp-3rd-ed/lib/python3.7/site-packages
Requires:
Required-by:

$ pip install 'pip>=18.0'
Requirement already satisfied: pip>=18.0 in (...)/lib/python3.7/sitepackages (18.0)

$ pip install --upgrade pip
Requirement already up-to-date: pip in (...)/lib/python3.7/site-packages
(18.0)
```

In some cases, pip may not be available by default. From Python 3.4 onward (and also Python 2.7.9), it can always be bootstrapped using the ensurepip module:

```
$ python -m ensurepip
Looking in links:
/var/folders/z6/3m2r6jgd04q0m7yq29c6lbzh0000gn/T/tmp784u9bct
Requirement already satisfied: setuptools in /Users/swistakm/.envs/epp-3rd-ed/lib/
↪python3.7/site-packages (40.4.3)
Collecting pip
Installing collected packages: pip
Successfully installed pip-10.0.1
```

25.2 2. Isolating the runtime environment

`pip` may be used to install system-wide packages. On UNIX-based and Linux systems, this will require superuser privileges, so the actual invocation will be as follows:

```
sudo pip install <package-name>
```

Note that this is not required on Windows since it does not provide the Python interpreter by default, and Python on Windows is usually installed manually by the user without superuser privileges.

Installing system-wide packages directly from PyPI is not recommended, and should be avoided. This may seem like a contradiction to the previous statement that using `pip` is a PyPA recommendation, but there are some serious reasons for that. As we explained earlier, Python is often an important part of many packages that are available through operating system package repositories, and may power a lot of important services. System distribution maintainers put in a lot of effort to select the correct versions of packages to match various package dependencies. Very often, Python packages that are available from a system's package repositories contain custom patches, or are purposely kept outdated to ensure compatibility with some other system components. Forcing an update of such a package, using pip, to a version that breaks some backward compatibility, might cause bugs in some crucial system service.

Doing such things on the local computer for development purposes only is also not a good excuse. Recklessly using pip that way is almost always asking for trouble, and will eventually lead to issues that are very hard to debug. This does not mean that installing packages from PyPI is a strictly forbidden thing, but it should be always done consciously and with an understanding of the related risk.

Fortunately, there is an easy solution to this problem: environment isolation. There are various tools that allow the isolation of the Python runtime environment at different levels of system abstraction. The main idea is to isolate project dependencies from packages that are required by different projects and/or system services. The benefits of this approach are as follows:

- It solves the Project X depends on version 1.x but, Project Y needs 4.x dilemma. The developer can work on multiple projects with different dependencies that may even collide without the risk of affecting each other.
- The project is no longer constrained by versions of packages that are provided in the developer's system distribution repositories.
- There is no risk of breaking other system services that depend on certain package versions, because new package versions are only available inside such an environment.
- A list of packages that are project dependencies can be easily frozen, so it is very easy to reproduce such an environment on another computer.

If you're working on multiple projects in parallel, you'll quickly find that is impossible to maintain their dependencies without any kind of isolation.

25.2.1 2.1 Application-level isolation versus system-level isolation

The easiest and most lightweight approach to isolation is to use application-level virtual environments. These focus on isolating the Python interpreter and the packages available within it. Such environments are very easy to set up, and are very often just enough to ensure proper isolation during the development of small projects and packages.

Unfortunately, in some cases, this may not be enough to ensure enough consistency and reproducibility. Despite the fact that software written in Python is usually considered very portable, it is still very easy to run into issues that occur only on selected systems or even specific distributions of such systems (for example, Ubuntu versus Gentoo). This is very common in large and complex projects, especially if they depend on compiled Python extensions or internal components of the hosting operating system.

In such cases, system-level isolation is a good addition to the workflow. This kind of approach usually tries to replicate and isolate complete operating systems with all of its libraries and crucial system components, either with classical system virtualization tools (for example, VMWare, Parallels, and VirtualBox) or container systems (for example, Docker and Rocket).

25.2.2 2.2. Python's venv

There are several ways to isolate Python at runtime. The simplest and most obvious, although hardest to maintain, is to manually change the `PATH` and `PYTHONPATH` environment variables and/or move the Python binary to a different, customized place where we want to store our project's dependencies, in order to affect the way that it discovers available packages. Fortunately, there are several tools available that can help in maintaining the virtual environments and packages that are installed for these environments. These are mainly `virtualenv` and `venv`. What they do under the hood is, in fact, the same that we would do manually. The actual strategy depends on the specific tool implementation, but generally they are more convenient to use and can provide additional benefits.

To create new virtual environment, you can simply use the following command:

```
python3.7 -m venv ENV
```

Here, `ENV` should be replaced by the desired name for the new environment. This will create a new `ENV` directory in the current working directory path. Inside, it will contain a few new directories:

- `bin/`: This is where the new Python executable and scripts/executables provided by other packages are stored.
- `lib/` and `include/`: These directories contain the supporting library files for new Python inside the virtual environment. The new packages will be installed in ENV/lib/pythonX.Y/site-packages/.

Once the new environment has been created, it needs to be activated in the current shell session using UNIX's source command: `source ENV/bin/activate`

This changes the state of the current shell sessions by affecting its environment variables. In order to make the user aware that they have activated the virtual environment, it will change the shell prompt by appending the (`ENV`)

string at its beginning. To illustrate this, here is an example session that creates a new environment and activates it:

```
$ python -m venv example
$ source example/bin/activate
(example) $ which python
/home/swistakm/example/bin/python
(example) $ deactivate
$ which python
/usr/local/bin/python
```

The important thing to note about `venv` is that it depends completely on its state, as stored on a filesystem. It does not provide any additional abilities to track what packages should be installed in it. These virtual environments are also not portable, and should not be moved to another system/machine. This means that the new virtual environment needs to be created from scratch for each new application deployment. Because of this, there is a good practice that's used by `venv` users to store all project dependencies in the `requirements.txt` file (this is the naming convention), as shown in the following code:

```
# lines followed by hash (#) are treated as a comments
# strict version names are best for reproducibility
eventlet==0.17.4
graceful==0.1.1
# for projects that are well tested with different
# dependency versions the relative version specifiers
# are acceptable too
falcon>=0.3.0,<0.5.0
# packages without versions should be avoided unless
# latest release is always required/desired
pytz
```

With such files, all dependencies can be easily installed using `pip`, because it accepts the requirements file as its output: `pip install -r requirements.txt`

What needs to be remembered is that the requirements file is not always the ideal solution, because it does not define the exact list of dependencies, only those that are to be installed. So, the whole project can work without problems in some development environments but will fail to start in others if the requirements file is outdated and does not reflect the actual state of the environment. There is, of course, the `pip freeze` command, which prints all packages in the current environment, but it should not be used blindly. It will output everything, even packages that are not used in the project but are installed only for testing.

Important: For Windows users, `venv` under Windows uses a different naming convention for its internal structure of directories. You need to use `Scripts/`, `Libs/`, and `Include/` instead of `bin/`, `lib/`, and `include/`, to better match development conventions on that operating system. The commands that are used for activating/deactivating the environment are also different; you need to use `ENV/Scripts/activate.bat` and `ENV/Scripts/deactivate.bat` instead of using source on activate and deactivate scripts.

Important: The Python `venv` module provides an additional `pyvenv` command-line script; since Python 3.6, it has been marked as deprecated and its usage is officially discouraged, as the `pythonX.Y -m venv` command is explicit about what version of Python will be used to create new environments, unlike the `pyvenv` script.

25.2.3 2.3. System-level environment isolation

In most cases, software implementation can iterate quickly because developers reuse a lot of existing components. Don't Repeat Yourself: this is a popular rule and motto of many programmers. Using other packages and modules to include them in the code base is only a part of that culture. What can also be considered under reused components are binary libraries, databases, system services, third-party APIs, and so on. Even whole operating systems should be considered as being reused.

The backend services of web-based applications are a great example of how complex such applications can be. The simplest software stack usually consists of a few layers (starting from the lowest):

- A database or other kind of storage
- The application code implemented in Python
- An HTTP server, such as Apache or NGINX

Of course, such stacks can be even simpler, but it is very unlikely. In fact, big applications are often so complex that it is hard to distinguish single layers. Big applications can use many different databases, be divided into multiple independent processes, and use many other system services for caching, queuing, logging, service discovery, and so on. Sadly, there are no limits for complexity, and it seems that code simply follows the second law of thermodynamics.

What is really important is that not all software stack elements can be isolated on the level of Python runtime environments. No matter whether it is an HTTP server, such as Nginx, or RDBMS, such as PostgreSQL, they are usually available in different versions on different systems. Making sure that everyone in a development team uses the same versions of every component is very hard without the proper tools. It is theoretically possible that all developers in a team working on a single project will be able to get the same versions of services on their development boxes. But all this effort is futile if they do not use the same operating system as they do in the production environment. Forcing a programmer to work on something else rather than their beloved system of choice is impossible.

The problem lies in the fact that portability is still a big challenge. Not all services will work exactly the same in production environments as they do on the developer's machines, and this is very unlikely to change. Even Python can behave differently on different systems, despite how much work is put in to make it cross-platform. Usually, this is well documented and happens only in places that depend directly on system calls, but relying on the programmer's ability to remember a long list of compatibility quirks is quite an error-prone strategy.

A popular solution to this problem is isolating whole systems as an application environment. This is usually achieved by leveraging different types of system virtualization tools. Virtualization, of course, reduces performance; but with modern computers that have hardware support for virtualization, the performance loss is usually negligible. On the other hand, the list of possible gains is very long:

- The development environment can exactly match the system version and services used in production, which helps to solve compatibility issues
- Definitions for system configuration tools, such as Puppet, Chef, or Ansible (if used), can be reused to configure the development environment
- The newly hired team members can easily hop into the project if the creation of such environments is automated
- The developers can work directly with low-level system features that may not be available on operating systems they use for work, for example, File System in User Space (FUSE), which is not available in Windows.

2.3.1. Virtual development environments using Vagrant

Vagrant currently seems to be one of the most popular tools for developers to manage virtual machines for the purpose of local development. It provides a simple and convenient way to describe development environments with all system dependencies in a way that is directly tied to the source code of your project. It is available for Windows, Mac OS, and a few popular Linux distributions (refer to https://www.vagrantup.com). It does not have any additional dependencies. Vagrant creates new development environments in the form of virtual machines or containers. The exact implementation depends on a choice of virtualization providers. VirtualBox is the default provider, and it is bundled with the Vagrant installer, but additional providers are available as well. The most notable choices are VMware, Docker, Linux Containers (LXC), and Hyper-V.

The most important configuration is provided to Vagrant in a single file named `Vagrantfile`. It should be independent for every project. The following are the most important things it provides:

- Choice of virtualization provider
- A box, which is used as a virtual machine image
- Choice of provisioning method
- Shared storage between the VM and VM's host
- Ports that need to be forwarded between VM and its host

The syntax language for Vagrantfile is Ruby. The example configuration file provides a good template to start the project and has an excellent documentation, so the knowledge of this language is not required. Template configuration can be created using a single command:

```
vagrant init
```

This will create a new file named `Vagrantfile` in the current working directory. The best place to store this file is usually the root of the related project sources. This file is already a valid configuration that will create a new VM using the default provider and base box image. The default `Vagrantfile` content that's created with the `vagrant init` command contains a lot of comments that will guide you through the complete configuration process.

The following is a minimal example of `Vagrantfile` for the Python 3.7 development environment based on the Ubuntu operating system, with some sensible defaults that, among others, enable port 80 forwarding in case you want to do some web development with Python:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :
Vagrant.configure("2") do |config|
 # Every Vagrant development environment requires a box.
 # You can search for boxes at https://vagrantcloud.com/search.
 # Here we use Bionic version Ubuntu system for x64 architecture.
 config.vm.box = "ubuntu/bionic64"
 # Create a forwarded port mapping which allows access to a specific
 # port within the machine from a port on the host machine and only
 # allow access via 127.0.0.1 to disable public access
 config.vm.network "forwarded_port", guest: 80, host: 8080, host_ip: "127.0.0.1"
 config.vm.provider "virtualbox" do |vb|
 # Display the VirtualBox GUI when booting the machine
 vb.gui = false
 # Customize the amount of memory on the VM:
 vb.memory = "1024"
 end
 # Enable provisioning with a shell script.
 config.vm.provision "shell", inline: <<-SHELL
 apt-get update
 apt-get install python3.7 -y
 SHELL
end
```

In the preceding example, we have set an additional provision of system packages with simple shell script. When you feel that `Vagrantfile` is ready, you can run your virtual machine using the following command:

```
vagrant up
```

The initial start can take a few minutes, because the actual box image must be downloaded from the web. There are also some initialization processes that may take a while every time the existing VM is brought up, and the amount of time depends on the choice of provider, image, and your system's performance. Usually, this takes only a couple of seconds. Once the new Vagrant environment is up and running, developers can connect to it through SSH using the following shorthand:

```
vagrant ssh
```

This can be done anywhere in the project source tree below the location of `Vagrantfile`. For the developers' convenience, Vagrant will traverse all directories above the user's current working directory in the filesystem tree, looking for the configuration file and matching it with the related VM instance. Then, it establishes the secure shell connection, so the development environment can be interacted with just like an ordinary remote machine. The only difference is that the whole project source tree (root defined as the location of Vagrantfile) is available on the VM's filesystem under `/vagrant/`. This directory is automatically synchronized with your host filesystem, so you can normally work in the IDE or editor of your choice run on the host, and can treat the SSH session to your Vagrant VM just like a normal local Terminal session.

2.3.2. Virtual environments using Docker

Containers are an alternative to full machine virtualization. It is a lightweight method of virtualization, where the kernel and operating system allow multiple isolated user space instances to be run. OS is shared between containers and the host, so it theoretically requires less overhead than in full virtualization. Such a container contains only application code and its system-level dependencies, but, from the perspective of processes running inside, it looks like a completely isolated system environment.

Software containers got their popularity mostly thanks to Docker, which is one of the available implementations. Docker allows to describe its container in the form of a simple text document called `Dockerfile`. Containers from such definitions can be built and stored. It also supports incremental changes, so if new things are added to the container then it does not need to be recreated from scratch.

2.3.2.1. Containerization versus virtualization

Different tools, such as Docker and Vagrant, seem to overlap in features: but the main difference between them is the reason why these tools were built. Vagrant, as we mentioned earlier, is built primarily as a tool for development. It allows us to bootstrap the whole virtual machine with a single command, but does not allow us to simply pack such an environment as a complete deliverable artifact and deploy or release it. Docker, on the other hand, is built exactly for that purpose: preparing complete containers that can be sent and deployed to production as a whole package. If implemented well, this can greatly improve the process of product deployment. Because of that, using Docker and similar solutions (Rocket for example) during development only makes more sense if such containers are also to be used in the deployment process on production.

Due to some implementation nuances, the environments that are based on containers may sometimes behave differently than environments based on virtual machines. If you decide to use containers for development, but don't decide to use them on target production environments, you'll lose some of the consistency guarantees that were the main reason for environment isolation. But, if you already use containers in your target production environments, then you should always replicate production conditions rather than using the same technique. Fortunately, Docker, which is currently the most popular container solution, provides an amazing `docker-compose` tool that makes the management of local containerized environments extremely easy.

2.3.2.2. Writing your first Dockerfile

Every Docker-based environment starts with `Dockerfile`. Dockerfile is a format description of how to create a Docker image. You can think about the Docker images in a similar way to how you would think about images of virtual machines. It is a single file (composed of many layers) that encapsulates all system libraries, files, source code, and other dependencies that are required to execute your application.

Every layer of a Docker image is described in the Dockerfile by a single instruction in the following format: `INSTRUCTION arguments`.

Docker supports plenty of instructions, but the most basic ones that you need to know in order to get started are as follows:

- `FROM <image-name>`: This describes the base image that your image will be based on.
- `COPY <src>... <dst>`: This copies files from the local build context (usually project files) and adds them to the container's filesystem.
- `ADD <src>... <dst>`: This works similarly to COPY but automatically unpacks archives and allows <src> to be URLs.
- `RUN <command>`: This runs specified commands on top of previous layers, and commits changes that this command made to the filesystem as a new image layer.
- `ENTRYPOINT ["<executable>", "<param>", ...]`: This configures the default command to be run as your container. If no entry point is specified anywhere in the image layers, then Docker defaults to /bin/sh -c.
- `CMD ["<param>", ...]`: This specifies the default parameters for image entry points. Knowing that the default entry point for Docker is /bin/sh -c, this instruction can also take the form of CMD ["<executable>", "<param>", ...], although it is recommended to define the target executable directly in the ENTRYPOINT instruction and use CMD only for default arguments.
- `WORKDIR <dir>`: This sets the current working directory for any of the following RUN, CMD, ENTRYPOINT, COPY, and ADD instructions.

To properly illustrate the typical structure of `Dockerfile`, let's assume that we want to dockerize the built-in Python web server available through the `http.server` module with some predefined static files that this server should serve. The structure of our project files could be as follows:

```
.
├── Dockerfile
├── README
└── static
    ├── index.html
    └── picture.jpg
```

Locally, you could run that Python's `http.server` on a default HTTP port with the following simple command:

```
python3.7 -m http.server --directory static/ 80
```

This example is of course, very trivial, and using Docker for it is using a sledgehammer to crack a nut. So, just for the purpose of this example, let's pretend that we have a lot of code in the project that generates these static files. We would like to deliver only these static files, and not the code that generates them. Let's also assume that the recipients of our image know how to use Docker but don't know how to use Python.

So, what we want to achieve is the following:

- Hide some complexity from the user—especially the fact that we use Python and the HTTP server that's built-in into Python
- Package Python3.7 executable with all its dependencies and all static files primarily available in our project directory
- Provide some defaults to run the server on port 80

With all these requirements, our Dockerfile could take the following form:

```
# Let's define base image.
# "python" is official Python image.
# The "slim" versions are sensible starting
# points for other lightweight Python-based images
FROM python:3.7-slim

# In order to keep image clean let's switch
# to selected working directory. "/app/" is
# commonly used for that purpose.
WORKDIR /app/

# These are our static files copied from
# project source tree to the current working
# directory.
COPY static/ static/

# We would run "python -m http.server" locally
# so lets make it an entry point.
ENTRYPOINT ["python3.7", "-m", "http.server"]

# We want to serve files from static/ directory
# on port 80 by default so set this as default arguments
# of the built-in Python HTTP server
CMD ["--directory", "static/", "80"]
```

2.3.2.3. Running containers

Before your container can be started, you'll first need to build an image defined in the `Dockerfile`. You can build the image using the following command:

```
docker build -t <name> <path>
```

The `-t <name>` argument allows us to name the image with a readable identifier. It is totally optional, but without it you won't be able to easily reference a newly created image. The `<path>` argument specifies the path to the directory where your Dockerfile is located. Let's assume that we were already running the command from the root of the project we presented in the previous section, and we want to tag our image with the name webserver. The docker build command invocation will be following, and its output may be as follows:

```
$ docker build -t webserver .

Sending build context to Docker daemon 4.608kB
Step 1/5 : FROM python:3.7-slim
3.7-slim: Pulling from library/python
802b00ed6f79: Pull complete
cf9573ca9503: Pull complete
b2182f7db2fb: Pull complete
37c0dde21a8c: Pull complete
a6c85c69b6b4: Pull complete
Digest:
sha256:b73537137f740733ef0af985d5d7e5ac5054aadebfa2b6691df5efa793f9fd6d
Status: Downloaded newer image for python:3.7-slim
 ---> a3aec6c4b7c4
Step 2/5 : WORKDIR /app/
 ---> Running in 648a5bb2d9ab
Removing intermediate container 648a5bb2d9ab
 ---> a2489d084377
Step 3/5 : COPY static/ static/
 ---> 958a04fa5fa8
```

(continues on next page)

(continued from previous page)

```
Step 4/5 : ENTRYPOINT ["python3.7", "-m", "http.server", "--bind", "80"]
 ---> Running in ec9f2a63c472
Removing intermediate container ec9f2a63c472
 ---> 991f46cf010a
Step 5/5 : CMD ["--directory", "static/"]
 ---> Running in 60322d5a9e9e
Removing intermediate container 60322d5a9e9e
 ---> 40c606a39f7a
Successfully built 40c606a39f7a
Successfully tagged webserver:latest
```

Once created, you can inspect the list of available images using the docker images command:

```
$ docker images

REPOSITORY      TAG         IMAGE ID        CREATED         SIZE
webserver       latest      40c606a39f7a    2 minutes ago   143MB
python          3.7-slim    a3aec6c4b7c4    2 weeks ago     143MB
```

Note: The 143 MB of image for a simple Python image may seem like a lot, but it isn't really anything to worry about. For the sake of brevity, we have used a base image that is simple to use. There are other images that have been crafted specially to minimize this size, but these are usually dedicated to more experienced Docker users. Also, thanks to the layered structure of Docker images, if you're using many containers, the base layers can be cached and reused, so an eventual space overhead is rarely an issue.

Once your image is built and tagged, you can run a container using the docker run command. Our container is an example of a web service, so we will have to additionally tell Docker that we want to publish the container's ports by binding them locally:

```
docker run -it --rm -p 80:80 webserver
```

Here is an explanation of the specific arguments of the preceding command:

- `-it`: These are actually two concatenated options: `-i` and `-t`. `-i` (like interactive) keeps STDIN open, even if the container process is detached, and `-t` (like tty) allocates pseudo-TTY for the container. In short, thanks to these two options, we will be able to see live logs from `http.server` and ensure that the keyboard interrupt will cause the process to exit. It will simply behave the same way as we would start Python, straight from the command line.
- `--rm`: Tells Docker to automatically remove container when it exits.
- `-p 80:80`: Tells Docker to publish the container's port 80 by binding port 80 on the host's interface.

2.3.2.4. Setting up complex environments

While the basic usage of Docker is pretty straightforward for basic setups, it can be bit overwhelming once you start to use it in multiple projects. It is really easy to forget about specific command-line options, or which ports should be published on which images. But things start to be really complicated when you have one service that needs to communicate with others. Single docker containers should only contain one running process.

This means that you really shouldn't put any additional process supervision tools, such as Supervisor and Circus, and should instead set up multiple containers that communicate with each other. Each service may use a completely different image, provide different configuration options, and expose ports that may or may not overlap.

The best tool that you can use in both simple and complex scenarios is Compose. Compose is usually distributed with Docker, but in some Linux distributions (for example, Ubuntu), it may not be available by default, and must be installed as a separate package from the packages repository. Compose provides a powerful command-line utility named `docker-compose`, and allows you to describe multi-container applications using the YAML syntax.

Compose expects the specially named `docker-compose.yml` file to be in your project directory. An example of such a file for our previous project could be as follows:

```
version: '3'
services:
     webserver:
          # this tell Compose to build image from
          # local (.) directory
          build: .
          # this is equivalent to "-p" option of
          # the "docker build" command
          ports:
              - "80:80"
          # this is equivalent to "-t" option of
          # the "docker build" command
          tty: true
```

If you create such a docker-compose.yml file in your project, then your whole application environment can be started and stopped with two simple commands:

```
docker-compose up
docker-compose down
```

2.3.2.5. Reducing the size of containers

A common concern of new Docker users is the size of their container images. It's true that containers provide a lot of space overhead compared to plain Python packages, but it is usually nothing if we compare the size of images for virtual machines. However, it is still very common to host many services on a single virtual machine, but with a container-based approach, you should definitely have a separate image for every service. This means that with a lot of services, the overhead may become noticeable.

If you want to limit the size of your images, you can use two complementary techniques:

1. Use a base image that is designed specifically for that purpose: Alpine Linux is an example of a compact Linux distribution that is specifically tailored to provide very small and lightweight Docker images. The base image is only 5 MB in size, and provides an elegant package manager that allows you to keep your images compact, too.
2. Take into consideration the characteristics of the Docker overlay filesystem: Docker images consist of layers where each layer encapsulates the difference in the root filesystem between itself and the previous layer. Once the layer is committed the size of the image cannot be reduced. This means that if you need a system package as a build dependency, and it may be later discarded from the image, then instead of using multiple RUN instructions, it may be better to do everything in a single RUN instruction with chained shell commands to avoid excessive layer commits.

These two techniques can be illustrated by the following Dockerfile:

```
# Here we use bare alpine to illustrate
# package management as it lacks Python
# by default. For Python projects in general
# the 'python:3.7-alpine' is probably better
# choice.
FROM alpine:3.7

# Add python3 package as alpine image lacks it by default
RUN apk add python3

# Run multiple commands in single RUN instruction
# so space can be reclaimed after the 'apk del py3-pip'
# command because image layer is committed only after
# whole whole instruction.
```

(continues on next page)

(continued from previous page)

```
RUN apk add py3-pip && \
 pip3 install django && \
 apk del py3-pip
```

2.3.2.6. Addressing services inside of a Compose environment

Complex applications often consist of multiple services that communicate with each other. Compose allows us to define such applications with ease. The following is an example `docker-compose.yml` file that defines the application as a composition of two services:

```
version: '3'
services:
    webserver:
        build: .
        ports:
            - "80:80"
        tty: true

    database:
        image: postgres
        restart: always
```

The preceding configuration defines two services:

- `webserver`: This is a main application service container with images built from the local Dockerfile
- `database`: This is a PostgreSQL database container from an official postgres Docker image

We assume that the `webserver` service wants to communicate with the `database` service over the network. In order to set up such communications, we need to know the service IP address or hostname so that it can be used as an application configuration. Thankfully, Compose is a tool that was designed exactly for such scenarios, so it will make it a lot more easier for us.

Whenever you start your environment with the `docker-compose up` command, Compose will create a dedicated Docker network by default, and will register all services in that network using their names as their hostnames. This means that the `webserver` service can use the `database:5432` address to communicate with its database (5432 is the default PostgreSQL port), and any other service in that Compose applicant will be able to access the HTTP endpoint of the webserver service under the `http://webserver:80` address.

Even though the service hostnames in Compose are easily predictable, it isn't good practice to hardcode any addresses in your application or its configuration. The best approach would be to provide them as environment variables that can be read by an application on startup. The following example shows how arbitrary environment variables can be defined for each service in a `docker-compose.yml` file:

```
version: '3'
services:
     webserver:
         build: .
         ports:
            - "80:80"
         tty: true
         environment:
             - DATABASE_HOSTNAME=database
             - DATABASE_PORT=5432

     database:
         image: postgres
         restart: always
```

2.3.2.7. Communicating between multiple Compose environments

If you build a system composed of multiple independent services and/or applications, you will very likely want to keep their code in multiple independent code repositories (projects). The `docker-compose.yml` files for every Compose application are usually kept in the same code repository as the application code. The default network that was created by Compose for a single application is isolated from the networks of other applications. So, what can you do if you suddenly want your multiple independent applications to communicate with each other?

Fortunately, this is another thing that is extremely easy with Compose. The syntax of the `docker-compose.yml` file allows you to define a named external Docker network as the default network for all services defined in that configuration. The following is an example configuration that defines an external network named `my-interservicenetwork`:

```
version: '3'

networks:
     default:
        external:
            name: my-interservice-network

services:
     webserver:
         build: .
         ports:
            - "80:80"
         tty: true
         environment:
             - DATABASE_HOSTNAME=database
             - DATABASE_PORT=5432

     database:
         image: postgres
         restart: always
```

Such external networks are not managed by Compose, so you'll have to create it manually with the docker network create command, as follows:

```
docker network create my-interservice-network
```

Once you have done this, you can use this external network in other `docker-compose.yml` files for all applications that should have their services registered in the same network. The following is an example configuration for other applications that will be able to communicate with both `database` and `webserver` services over `my-interservicenetwork`, even though they are not defined in the same `docker-compose.yml` file:

```
version: '3'
networks:
     default:
        external:
            name: my-interservice-network
services:
     other-service:
         build: .
         ports:
            - "80:80"
         tty: true
         environment:
             - DATABASE_HOSTNAME=database
             - DATABASE_PORT=5432
             - WEBSERVER_ADDRESS=http://webserver:80
```

25.3 3. Popular productivity tools

Productivity tool is bit of a vague term. On one hand, almost every open source code package that has been released and is available online is a kind of productivity booster: it provides ready-to-use solutions to some problem, so that no one needs to spend time on it (ideally speaking). On the other hand, you could say that the whole of Python is about productivity, and both are undoubtedly true. Almost everything in this language and community surrounding it seems to be designed in order to make software development as productive as possible.

This creates a positive feedback loop. Since writing code is fun and easy, a lot of programmers use their free time to create tools that make it even easier and fun. And this fact will be used here as a basis for a very subjective and non-scientific definition of a productivity tool: a piece of software that makes development easier and more fun.

By nature, productivity tools focus mainly on certain elements of the development process, such as testing, debugging, and managing packages, and are not core parts of products that they help to build. In some cases, they may not even be referred to anywhere in the project's codebase, despite being used on a daily basis.

The most important productivity tools, `pip` and `venv`, were already discussed earlier. Some of them have packages for specific problems, such as profiling and testing, which have their own chapters in this book. This section is dedicated to other tools that are really worth mentioning, but have no specific chapter in this book where they could be introduced.

25.3.1 3.1. Custom Python shells

Python programmers spend a lot of time in interactive interpreter sessions. It is very good for testing small code snippets, accessing documentation, or even debugging code at runtime. The default interactive Python session is very simple, and does not provide many features, such as tab completion or code introspection helpers. Fortunately, the default Python shell can be easily extended and customized.

If you use an interactive shell very often, you can easily modify the behavior of it prompt. Python at startup reads the `PYTHONSTARTUP` environment variable, looking for the path of the custom initializations script. Some operating system distributions where Python is a common system component (for example, Linux, macOS) may be already preconfigured to provide a default startup script. It is commonly found in the users, home directory under the `.pythonstartup` name. These scripts often use the `readline` module (based on the GNU readline library) together with `rlcompleter` in order to provide interactive tab completion and command history.

If you don't have a default python startup script, you can easily build your own. A basic script for command history and tab completion can be as simple as the following:

```
import atexit
import os


try:
    import readline
except ImportError:
    print("Completion unavailable: readline module not available")
else:
     import rlcompleter
     # tab completion
     readline.parse_and_bind('tab: complete')
     # Path to history file in user's home directory.
     # Can use your own path.
     history_file = os.path.join(os.environ['HOME'], '.python_shell_history')
     try:
        readline.read_history_file(history_file)
     except IOError:
        pass

     atexit.register(readline.write_history_file, history_file)
     del os, history_file, readline, rlcompleter
```

Create this file in your home directory and call it `.pythonstartup`. Then, add a `PYTHONSTARTUP` variable in your environment using the path of your file.

3.1.1. Setting up the PYTHONSTARTUP environment variable

If you are running Linux or macOS, the simplest way is to create the startup script in your home folder. Then, link it with a `PYTHONSTARTUP` environment variable that's been set in the system shell startup script. For example, Bash and Korn shell use the `.profile` file, where you can insert a line, as follows:

```
export PYTHONSTARTUP=~/.pythonstartup
```

If you are running Windows, it is easy to set a new environment variable as an administrator in the system preferences, and save the script in a common place instead of using a specific user location.

Writing on the PYTHONSTARTUP script may be a good exercise, but creating a good custom shell all alone is a challenge that only a few can find time for. Fortunately, there are a few custom Python shell implementations that immensely improve the experience of interactive sessions in Python.

3.1.2. IPython

IPython (https://ipython.readthedocs.io/en/stable/overview.html) provides an extended Python command shell. Among the features that it provides, the most interesting ones are as follows:

- Dynamic object introspection
- System shell access from the prompt
- Profiling direct support
- Debugging facilities

Now, IPython is a part of the larger project called Jupyter, which provides interactive notebooks with live code that can be written in many different languages.

3.1.3. bpython

bpython (https://bpython-interpreter.org/) advertises itself as a fancy interface to the Python interpreter. Here are some of the accented features on the projects page:

- In-line syntax highlighting
- Readline-like autocomplete with suggestions displayed as you type
- Expected parameter list for any Python function
- Auto-indentation
- Python 3 support

3.1.4. ptpython

ptpython (https://github.com/jonathanslenders/ptpython/) is another approach to the topic of advanced Python shells. What is interesting about this project is that core prompt utilities implementation is available as a separate package, called `prompt_toolkit` (from the same author). This allows us to easily create various aesthetically pleasing interactive command-line interfaces.

It is often compared to bpython in functionalities, but the main difference is that it enables compatibility mode with IPython and its syntax enables additional features, such as %pdb, %cpaste, or %profile.

25.3.2 3.2. Incorporating shells in your own scripts and programs

Sometimes, there is a need to incorporate a read-eval-print loop (REPL), similar to Python's interactive session, inside of your own software. This allows for easier experimentation with your code and inspection of its internal state. The simplest module that allows for emulating Python's interactive interpreter already comes with the standard library and is named code.

The script that starts interactive sessions consists of one import and single function call:

```
import code
code.interact()
```

You can easily do some minor tuning, such as modify a prompt value or add banner and exit messages, but anything more fancy will require a lot more work. If you want to have more features, such as code highlighting, completion, or direct access to the system shell, it is always better to use something that was already built by someone. Fortunately, all of the interactive shells that were mentioned in the previous section can be embedded in your own program as easily as the code module.

The following are examples of how to invoke all of the previously mentioned shells inside of your code:

```
import IPython
IPython.embed()

import bpython
bpython.embed()

from ptpython.repl import embed
embed(globals(), locals())
```

25.3.3 3.3. Interactive debuggers

Code debugging is an integral element of the software development process. Many programmers can spend most of their life using only extensive logging and print statements as their primary debugging tools, but most professional developers prefer to rely on some kind of debugger.

Python already ships with a built-in interactive debugger called `pdb` (refer to https://docs.python.org/3/library/pdb.html). It can be invoked from the command line on the existing script, so Python will enter post-mortem debugging if the program exits abnormally:

```
python -m pdb script.py
```

Post-mortem debugging, while useful, does not cover every scenario. It is useful only when the application exits with some exception if the bug occurs. In many cases, faulty code just behaves abnormally, but does not exit unexpectedly. In such cases, custom breakpoints can be set on a specific line of code using this single-line idiom:

```
import pdb
pdb.set_trace()
```

This will cause the Python interpreter to start the debugger session on this line during runtime.

`pdb` is very useful for tracing issues, and at first glance it may look very familiar to the wellknown GNU Debugger (GDB). Because Python is a dynamic language, the `pdb` session is very similar to an ordinary interpreter session. This means that the developer is not limited to tracing code execution, but can call any code and even perform module imports.

Sadly, because of its roots (`bdb`), your first experience with `pdb` can be a bit overwhelming due to the existence of cryptic short-letter debugger commands such as `h`, `b`, `s`, `n`, `j`, and `r`. When in doubt, the `help pdb` command which can be typed during the debugger session, will provide extensive usage and additional information.

The debugger session in `pdb` is also very simple and does not provide additional features such as tab completion or code highlighting. Fortunately, there are a few packages available on PyPI that provide such features from alternative Python shells, as mentioned in the previous section. The most notable examples are as follows:

- `ipdb`: This is a separate package based on ipython
- `ptpdb`: This is a separate package based on ptpython
- `bpdb`: This is bundled with bpython

Part VII

Code distribution

CHAPTER

TWENTYSIX

PACKAGING

This chapter focuses on a repeatable process of writing and releasing Python packages. We will see how to shorten the time needed to set up everything before starting the real work. We will also learn how to provide a standardized way to write packages and ease the use of a test-driven development approach. We will finally learn how to facilitate the release process.

It is organized into the following four parts:

- A **common pattern** for all packages that describes the similarities between all Python packages, and how distutils and setuptools play a central role the packaging process.
- What are **namespace packages** and why they can be useful?
- How to register and upload packages in the **Python Package Index (PyPI)** with emphasis on security and common pitfalls.
- The **standalone executables** as an alternative way to package and distribute Python applications.

26.1 1. Creating a package

Python packaging can be a bit overwhelming at first. The main reason for that is the confusion about proper tools for creating Python packages. Anyway, once you create your first package, you will see that this is not as hard as it looks. Also, knowing proper, state-of- the art packaging tools helps a lot.

You should know how to create packages even if you are not interested in distributing your code as open source. Knowing how to make your own packages will give you more insight in the packaging ecosystem and will help you to work with third-party code that is available on PyPI that you are probably already using.

Also, having your closed source project or its components available as source distribution packages can help you to deploy your code in different environments. The advantages of leveraging the Python packaging ecosystem in the code deployment process will be described in more detail in the next chapter. Here we will focus on proper tools and techniques to create such distributions.

26.1.1 1.1. The confusing state of Python packaging tools

The state of Python packaging was very confusing for a long time and it took many years to bring organization to this topic. Everything started with the `distutils` package introduced in 1998, which was later enhanced by setuptools in 2003. These two projects started a long and knotted story of forks, alternative projects, and complete rewrites that tried to (once and for all) fix the Python packaging ecosystem. Unfortunately, most of these attempts never succeeded. The effect was quite the opposite. Each new project that aimed to supersede `setuptools` or `distutils` only added to the already huge confusion around packaging tools. Some of such forks were merged back to their ancestors (such as `distribute` which was a fork of `setuptools`) but some were left abandoned (such as `distutils2`).

Fortunately, this state is gradually changing. An organization called the Python Packaging Authority (PyPA) was formed to bring back the order and organization to the packaging ecosystem. The Python Packaging User Guide (https://packaging.python.org), maintained by PyPA, is the authoritative source of information about the latest packaging tools and best practices. Treat that site as the best source of information about packaging and

complementary reading for this chapter. This guide also contains a detailed history of changes and new projects related to packaging. So it is worth reading it, even if you already know a bit about packaging, to make sure you still use the proper tools.

Stay away from other popular internet resources, such as "The Hitchhiker's Guide to Packaging". It is old, not maintained, and mostly obsolete. It may be interesting only for historical reasons, and the Python Packaging User Guide is in fact a fork of this old resource.

1.1.1. The current landscape of Python packaging thanks to PyPA

PyPA, besides providing an authoritative guide for packaging, also maintains packaging projects and a standardization process for new official aspects of Python packaging. All of PyPA's projects can be found under a single organization on GitHub: https://github.com/pypa

Some of them were already mentioned. The following are the most notable:

- `pip`
- `virtualenv`
- `twine`
- `warehouse`

Note that most of them were started outside of this organization and were moved under PyPA patronage when they become mature and widespread solutions.

Thanks to PyPA engagement, the progressive abandonment of the eggs format in favor of wheels for built distributions has already happened. Also thanks to the commitment of the PyPA community, the old PyPI implementation was finally totally rewritten in the form of the Warehouse project. Now, PyPI has got a modernized user interface and many long- awaited usability improvements and features.

1.1.2. Tool recommendations

The Python Packaging User Guide gives a few suggestions on recommended tools for working with packages. They can be generally divided into the following two groups:

- Tools for installing packages
- Tools for package creation and distribution

Utilities from the first group recommended are:

- Use `pip` for installing packages from PyPI.
- Use `virtualenv` or `venv` for application-level isolation of the Python runtime environment.

The Python Packaging User Guide recommendations of tools for package creation and distribution are as follows:

- Use `setuptools` to define projects and create **source distributions**.
- Use **wheels** in favor of **eggs** to create **built distributions**.
- Use `twine` to upload package distributions to PyPI.

26.1.2 1.2. Project configuration

It should be obvious that the easiest way to organize the code of big applications is to split them into several packages. This makes the code simpler, easier to understand, maintain, and change. It also maximizes the reusability of your code. Separate packages act as components that can be used in various programs.

1.2.1. setup.py

The root directory of a package that has to be distributed contains a `setup.py` script. It defines all metadata as described in the `distutils` module. Package metadata is expressed as arguments in a call to the standard `setup()` function. Despite `distutils` being the standard library module provided for the purpose of code packaging, it is actually recommended to use the `setuptools` instead. The `setuptools` package provides several enhancements over the standard `distutils` module.

Therefore, the minimum content for this file is as follows:

```
from setuptools import setup

setup(
    name='mypackage'
)
```

`name` gives the full name of the package. From there, the script provides several commands that can be listed with the `--help-commands` option, as shown in the following code:

```
$ python3 setup.py --help-commands
Standard commands:
    build           build everything needed to install
    clean           clean up temporary files from 'build' command
    install         install everything from build directory
    sdist           create a source distribution (tarball, zip file, etc.)
    register        register the distribution with the Python package index
    bdist           create a built (binary) distribution
    check           perform some checks on the package
    upload          upload binary package to PyPI

Extra commands:
    bdist_wheel     create a wheel distribution
    alias           define a shortcut to invoke one or more commands
    develop         install package in 'development mode'

usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
or: setup.py --help [cmd1 cmd2 ...]
or: setup.py --help-commands
or: setup.py cmd --help
```

The actual list of commands is longer and can vary depending on the available `setuptools` extensions. It was truncated to show only those that are most important and relevant to this chapter. **Standard commands** are the built-in commands provided by `distutils`, whereas **extra commands** are the ones provided by third-party packages, such as `setuptools` or any other package that defines and registers a new command. Here, one such extra command registered by another package is `bdist_wheel`, provided by the `wheel` package.

1.2.2. setup.cfg

The `setup.cfg` file contains default options for commands of the `setup.py` script. This is very useful if the process for building and distributing the package is more complex and requires many optional arguments to be passed to the `setup.py` script commands. This `setup.cfg` file allows you to store such default parameters together with your source code on a per project basis. This will make your distribution flow independent from the project and also provides transparency about how your package was built/distributed to the users and other team members.

The syntax for the `setup.cfg` file is the same as provided by the built-in `configparser` module so it is similar to the popular Microsoft Windows INI files. Here is an example of the `setup.cfg` configuration file that provides some `global`, `sdist`, and `bdist_wheel` commands' defaults:

```
[global]
quiet=1

[sdist]
formats=zip,tar

[bdist_wheel]
universal=1
```

This example configuration will ensure that source distributions (`sdist` section) will always be created in two formats (ZIP and TAR) and the built `wheel` distributions (`bdist_wheel` section) will be created as universal wheels that are independent from the Python version. Also most of the output will be suppressed on every command by the global `--quiet` switch. Note that this option is included here only for demonstration purposes and it may not be a reasonable choice to suppress the output for every command by default.

1.2.3. MANIFEST.in

When building a distribution with the `sdist` command, the `distutils` module browses the package directory looking for files to include in the archive. By default `distutils` will include the following:

- All Python source files implied by the `py_modules`, `packages`, and `scripts` arguments
- All C source files listed in the `ext_modules` argument
- Files that match the glob pattern `test/test*.py`
- Files named `README`, `README.txt`, `setup.py`, and `setup.cfg`

Besides that, if your package is versioned with a version control system such as Subversion, Mercurial, or Git, there is the possibility to auto-include all version controlled files using additional `setuptools` extensions such as `setuptools-svn`, `setuptools-hg`, and `setuptools-git`. Integration with other version control systems is also possible through other custom extensions. No matter if it is the default built-in collection strategy or one defined by custom extension, the `sdist` will create a `MANIFEST` file that lists all files and will include them in the final archive.

Let's say you are not using any extra extensions, and you need to include in your package distribution some files that are not captured by default. You can define a template called `MANIFEST.in` in your package root directory (the same directory as `setup.py` file). This template directs the `sdist` command on which files to include.

This `MANIFEST.in` template defines one inclusion or exclusion rule per line:

```
include HISTORY.txt
include README.txt
include CHANGES.txt
include CONTRIBUTORS.txt
include LICENSE
recursive-include *.txt *.py
```

The full list of the `MANIFEST.in` commands can be found in the official `distutils` documentation.

1.2.4. Most important metadata

Besides the name and the version of the package being distributed, the most important arguments that the `setup()` function can receive are as follows:

- `description`: This includes a few sentences to describe the package.
- `long_description`: This includes a full description that can be in reStructuredText (default) or other supported markup languages.
- `long_description_content_type`: this defines MIME type of long description; it is used to tell the package repository what kind of markup language is used for the package description.
- `keywords`: This is a list of keywords that define the package and allow for better indexing in the package repository.
- `author`: This is the name of the package author or organization that takes care of it.
- `author_email`: This is the contact email address.
- `url`: This is the URL of the project.
- `license`: This is the name of the license (GPL, LGPL, and so on) under which the package is distributed.
- `packages`: This is a list of all package names in the package distribution; `setuptools` provides a small function called `find_packages` that can automatically find package names to include.
- `namespace_packages`: This is a list of namespace packages within package distribution.

1.2.5. Trove classifiers

PyPI and `distutils` provide a solution for categorizing applications with the set of classifiers called trove classifiers. All trove classifiers form a tree-like structure. Each classifier string defines a list of nested namespaces where every namespace is separated by the :: substring. Their list is provided to the package definition as a classifiers argument of the `setup()` function.

Here is an example list of classifiers taken from `solrq` project available on PyPI:

```
from setuptools import setup


setup(
    name="solrq",
    # (...)
    classifiers=[
    'Development Status :: 4 - Beta',
    'Intended Audience :: Developers',
    'License :: OSI Approved :: BSD License',
    'Operating System :: OS Independent',
    'Programming Language :: Python',
    'Programming Language :: Python :: 2',
    'Programming Language :: Python :: 2.6',
    'Programming Language :: Python :: 2.7',
    'Programming Language :: Python :: 3',
    'Programming Language :: Python :: 3.2',
    'Programming Language :: Python :: 3.3',
    'Programming Language :: Python :: 3.4',
    'Programming Language :: Python :: Implementation :: PyPy',
    'Topic :: Internet :: WWW/HTTP :: Indexing/Search',
    ]
)
```

Trove classifiers are completely optional in the package definition but provide a useful extension to the basic metadata available in the `setup()` interface. Among others, trove classifiers may provide information about supported Python versions, supported operating systems, the development stage of the project, or the license

under which the code is released. Many PyPI users search and browse the available packages by categories so a proper classification helps packages to reach their target.

Trove classifiers serve an important role in the whole packaging ecosystem and should never be ignored. There is no organization that verifies packages classification, so it is your responsibility to provide proper classifiers for your packages and not introduce chaos to the whole package index.

At the time of writing this section, there are 667 classifiers available on PyPI that are grouped into the following nine major categories:

- Development status
- Environment
- Framework
- Intended audience
- License
- Natural language
- Operating system
- Programming language
- Topic

This list is ever-growing, and new classifiers are added from time to time. It is thus possible that the total count of them will be different at the time you read this. The full list of currently available trove classifiers is available at https://pypi.org/classifiers.

1.2.6. Common patterns

Creating a package for distribution can be a tedious task for unexperienced developers. Most of the metadata that `setuptools` or `distutils` accept in their `setup()` function call can be provided manually ignoring the fact that this metadata may be also available in other parts of the project. Here is an example:

```
from setuptools import setup


setup(
    name="myproject",
    version="0.0.1",
    description="mypackage project short description",
    long_description="""
        Longer description of mypackage project
        possibly with some documentation and/or
        usage examples
    """,
    install_requires=[
        'dependency1',
        'dependency2',
        'etc'
    ]
)
```

Some of the metadata elements are often found in different places in a typical Python project. For instance, content of long description is commonly included in the project's `README` file, and it is a good convention to put a version specifier in the `__init__` module of the package. Hardcoding such package metadata as `setup()` function arguments redundancy to the project that allows for easy mistakes and inconsistencies in future. Both `setuptools` and `distutils` cannot automatically pick metadata information from the project sources, so you need to provide it yourself. There are some common patterns among the Python community for solving the most popular problems such as dependency management, version/readme inclusion, and so on. It is worth knowing at least a few of them because they are so popular that they could be considered as packaging idioms.

1.2.6.1. Automated inclusion of version string from package

The PEP 440 "Version Identification and Dependency Specification" document specifies a standard for version and dependency specification. It is a long document that covers accepted version specification schemes and defines how version matching and comparison in Python packaging tools should work. If you are using or plan to use a complex project version numbering scheme, then you should definitely read this document carefully. If you are using a simple scheme that consists just of one, two, three, or more numbers separated by dots, then you don't have to dig into the details of PEP 440. If you don't know how to choose the proper versioning scheme, I greatly recommend following the semantic versioning scheme.

The other problem related to code versioning is where to include that version specifier for a package or module. There is PEP 396 (Module Version Numbers) that deals exactly with this problem. PEP 396 is only an informational document and has a deferred status, so it is not a part of the official Python standards track. Anyway, it describes what seems to be a *de facto* standard now. According to PEP 396, if a package or module has a specific version defined, the version specifier should be included as a `__version__` attribute of package root `__init__.py` INI file or distributed module file. Another *de facto* standard is to also include the `VERSION` attribute that contains the tuple of the version specifier parts. This helps users to write compatibility code because such version tuples can be easily compared if the versioning scheme is simple enough.

So many packages available on PyPI follow both conventions. Their `__init__.py` files contain version attributes that look like the following:

```
VERSION = (0, 1, 1)
__version__ = ".".join([str(x) for x in VERSION])
```

The other suggestion of PEP 396 is that the version argument provided in the `setup()` function of the `setup.py` script should be derived from `__version__`, or the other way around. The Python Packaging User Guide features multiple patterns for single- sourcing project versioning, and each of them has its own advantages and limitations. My personal favorite is rather long and is not included in the PyPA's guide, but has the advantage of limiting the complexity only to the `setup.py` script. This boilerplate assumes that the version specifier is provided by the `VERSION` attribute of the package's `__init__` module and extracts this data for inclusion in the `setup()` call. Here is an excerpt from some imaginary package's setup.py script that illustrates this approach:

```
from setuptools import setup
import os

def get_version(version_tuple):
    if not isinstance(version_tuple[-1], int):
        return '.'.join(map(str, version_tuple[:-1])) + version_tuple[-1]
    return '.'.join(map(str, version_tuple))

init = os.path.join(os.path.dirname(__file__), 'src', 'some_package', '__init__.py
↪')
version_line = list(filter(lambda l: l.startswith('VERSION'), open(init)))[0]
PKG_VERSION = get_version(eval(version_line.split('=')[-1]))

setup(
    name='some-package',
    version=PKG_VERSION,
    # ...
)
```

1.2.6.2. README file

The Python Package Index can display the project's README file or the value of `long_description` on the package page in the PyPI portal. PyPI is able to interpret the markup used in the `long_description` content and render it as HTML on the package page. The type of markup language is controlled through the `long_description_content_type` argument of the `setup()` call. For now, there are the following three choices for markup available:

- Plain text with `long_description_content_type='text/plain'`
- reStructuredText with `long_description_content_type='text/x-rst'`
- Markdown with `long_description_content_type='text/markdown'`

Markdown and reStructuredText are the most popular choices among Python developers, but some might still want to use different markup languages for various reasons. If you want to use something different as your markup language for your project's README, you can still provide it as a project description on the PyPI page in a readable form. The trick lies in using the `pypandoc` package to translate your other markup language into reStructuredText (or Markdown) while uploading the package to the Python Package Index. It is important to do it with a fallback to plain content of your README file, so the installation won't fail if the user has no `pypandoc` installed. The following is an example of a `setup.py` script that is able to read the content of the README file written in AsciiDoc markup language and translate it to reStructuredText before including a `long_description` argument:

```
from setuptools import setup


try:
    from pypandoc import convert

    def read_md(file_path):
        return convert(file_path, to='rst', format='asciidoc')

except ImportError:
    convert = None
    print("warning: pypandoc module not found, could not convert Asciidoc to RST")

    def read_md(file_path):
        with open(file_path, 'r') as f:
        return f.read()

README = os.path.join(os.path.dirname(__file__), 'README')

setup(
    name='some-package',
    long_description=read_md(README),
    long_description_content_type='text/x-rst',
    # ...
)
```

1.2.6.3. Managing dependencies

Many projects require some external packages to be installed in order to work properly. When the list of dependencies is very long, there comes a question as to how to manage it. The answer in most cases is very simple. Do not over-engineer it. Keep it simple and provide the list of dependencies explicitly in your setup.py script as follows:

```
from setuptools import setup
```

(continues on next page)

(continued from previous page)

```
setup(
    name='some-package',
    install_requires=['falcon', 'requests', 'delorean']
    # ...
)
```

Some Python developers like to use `requirements.txt` files for tracking lists of dependencies for their packages. In some situations, you might find some reason for doing that, but in most cases, this is a relic of times where the code of that project was not properly packaged. Anyway, even such notable projects as Celery still stick to this convention. So if you are not willing to change your habits or you are somehow forced to use requirement files, then at least do it properly. Here is one of the popular idioms for reading the list of dependencies from the `requirements.txt` file:

```
from setuptools import setup
import os

def strip_comments(l):
    return l.split('#', 1)[0].strip()

def reqs(*f):
    return list(filter(None, [strip_comments(l)
                              for l in open(os.path.join(os.getcwd(), *f)).
↪readlines()]))
setup(
    name='some-package',
    install_requires=reqs('requirements.txt')
    # ...
)
```

26.1.3 1.2.7. The custom setup command

`distutils` allows you to create new commands. A new command can be registered with an entry point, which was introduced by `setuptools` as a simple way to define packages as plugins.

An entry point is a named link to a class or a function that is made available through some APIs in `setuptools`. Any application can scan for all registered packages and use the linked code as a plugin.

To link the new command, the entry_points metadata can be used in the setup call as follows:

```
setup(
    name="my.command",
    entry_points="""
        [distutils.commands]
        my_command = my.command.module.Class
    """
)
```

All named links are gathered in named sections. When `distutils` is loaded, it scans for links that were registered under `distutils.commands`.

This mechanism is used by numerous Python applications that provide extensibility.

26.1.4 1.3. Working with packages during development

Working with `setuptools` is mostly about building and distributing packages. However, you still need to use `setuptools` to install packages directly from project sources. And the reason for that is simple. It is a good habit to test if our packaging code works properly before submitting your package to PyPI. And the simplest way to test it is by installing it. If you send a broken package to the repository, then in order to re-upload it, you need to increase the version number.

Testing if your code is packaged properly before the final distribution saves you from unnecessary version number inflation and obviously from wasting your time. Also, installation directly from your own sources using `setuptools` may be essential when working on multiple related packages at the same time.

1.3.1. setup.py install

The `install` command installs the package in your current Python environment. It will try to build the package if no previous build was made and then inject the result into the filesystem directory where Python is looking for installed packages. If you have an archive with a source distribution of some package, you can decompress it in a temporary folder and then install it with this command. The `install` command will also install dependencies that are defined in the `install_requires` argument. Dependencies will be installed from the Python Package Index.

An alternative to the bare `setup.py` script when installing a package is to use `pip`. Since it is a tool that is recommended by PyPA, you should use it even when installing a package in your local environment just for development purposes. In order to install a package from local sources, run the following command:

```
pip install <project-path>
```

1.3.2. Uninstalling packages

Amazingly, `setuptools` and `distutils` lack the `uninstall` command. Fortunately, it is possible to uninstall any Python package using `pip` as follows:

```
pip uninstall <package-name>
```

Uninstalling can be a dangerous operation when attempted on system-wide packages. This is another reason why it is so important to use virtual environments for any development.

1.3.3. setup.py develop or pip -e

Packages installed with `setup.py` install are copied to the site-packages directory of your current Python environment. This means that whenever you make a change to the sources of that package, you are required to reinstall it. This is often a problem during intensive development because it is very easy to forget about the need to perform installation again. This is why `setuptools` provides an extra `develop` command that allows you to install packages in the development mode. This command creates a special link to project sources in the deployment directory (site-packages) instead of copying the whole package there. Package sources can be edited without the need for reinstallation and are available in the `sys.path` as if they were installed normally.

`pip` also allows you to install packages in such a mode. This installation option is called editable mode and can be enabled with the `-e` parameter in the `install` command as follows:

```
pip install -e <project-path>
```

Once you install the package in your environment in editable mode, you can freely modify the installed package in place and all the changes will be immediately visible without the need to reinstall the package.

26.2 2. Namespace packages

The Zen of Python that you can read after writing `import this` in the interpreter session says the following about namespaces:

"Namespaces are one honking great idea-let's do more of those!"

And this can be understood in at least two ways. The first is a namespace in context of the language. We all use the following namespaces without even knowing:

- The global namespace of a module
- The local namespace of the function or method invocation
- The class namespace

The other kind of namespaces can be provided at the packaging level. These are **namespace packages**. This is often an overlooked feature of Python packaging that can be very useful in structuring the package ecosystem in your organization or in a very large project.

Namespace packages can be understood as a way of grouping related packages, where each of these packages can be installed independently.

Namespace packages are especially useful if you have components of your application developed, packaged, and versioned independently but you still want to access them from the same namespace. This also helps to make clear to which organization or project every package belongs. For instance, for some imaginary Acme company, the common namespace could be `acme`. Therefore this organization could create the general `acme` namespace package that could serve as a container for other packages from this organization. For example, if someone from Acme wants to contribute to this namespace with, for example, an SQL-related library, they can create a new `acme.sql` package that registers itself in the acme namespace.

It is important to know what's the difference between normal and namespace packages and what problem they solve. Normally (without namespace packages), you would create a package called `acme` with an `sql` subpackage/submodule with the following file structure:

```
$ tree acme/
acme/
├── acme
│   ├── __init__.py
│   └── sql
│       └── __init__.py
└── setup.py

2 directories, 3 files
```

Whenever you want to add a new subpackage, let's say `templating`, you are forced to include it in the source tree of `acme` as follows:

```
$ tree acme/
acme/
├── acme
│   ├── __init__.py
│   ├── sql
│   │   └── __init__.py
│   └── templating
│       └── __init__.py
└── setup.py

3 directories, 4 files
```

Such an approach makes independent development of `acme.sql` and `acme.templating` almost impossible. The `setup.py` script will also have to specify all dependencies for every subpackage. So it is impossible (or at least very hard) to have an installation of some of the `acme` components optional. Also, with enough subpackages it is practically impossible to avoid dependency conflicts.

With namespace packages, you can store the source tree for each of these subpackages independently as follows:

```
$ tree acme.sql/
acme.sql/
├── acme
│   └── sql
│       └── __init__.py
└── setup.py

2 directories, 2 files


$ tree acme.templating/
acme.templating/
├── acme
│   └── templating
│       └── __init__.py
└── setup.py

2 directories, 2 files
```

And you can also register them independently in PyPI or any package index you use. Users can choose which of the subpackages they want to install from the acme namespace as follows, but they never install the general `acme` package (it doesn't even have to exist):

```
$ pip install acme.sql acme.templating
```

Note that independent source trees are not enough to create namespace packages in Python. You need a bit of additional work if you don't want your packages to not overwrite each other. Also proper handling may be different depending on the Python language version you target. Details of that are described in the next two sections.

26.2.1 2.1. Implicit namespace packages

If you use and target only Python 3, then there is good news for you. PEP 420 (Implicit Namespace Packages) introduced a new way to define namespace packages. It is part of the standards track and became an official part of the language since version 3.3. In short, every directory that contains Python packages or modules (including namespace packages too) is considered a namespace package if it does not contain the `__init__.py` file. So, the following are examples of file structures presented in the previous section:

```
$ tree acme.sql/
acme.sql/
├── acme
│   └── sql
│       └── __init__.py
└── setup.py

2 directories, 2 files


$ tree acme.templating/
acme.templating/
├── acme
│   └── templating
│       └── __init__.py
└── setup.py

2 directories, 2 files
```

They are enough to define that acme is a namespace package under Python 3.3 and later. Minimal `setup.py` for `acme.templating` package will look like following:

```
from setuptools import setup
setup(
    name='acme.templating',
    packages=['acme.templating'],
)
```

Unfortunately, the `setuptools.find_packages()` function does not support PEP 420 at the time of writing this section. This may change in the future. Also, a requirement to explicitly define a list of packages seems to be a very small price to pay for easy integration of namespace packages.

26.2.2 2.2. Namespace packages in previous Python versions

You can't use implicit namespace packages (PEP 420 layout) in Python versions older than 3.3. Still, the concept of namespace packages is very old and was commonly used for years in such mature projects such as Zope. It means that it is definitely possible to use namespace packages in older version of Python. Actually, there are several ways to define that the package should be treated as a namespace.

The simplest one is to create a file structure for each component that resembles an ordinary package layout without implicit namespace packages and leave everything to `setuptools`.

So, the example layout for `acme.sql` and a `cme.templating` could be the following:

```
$ tree acme.sql/
acme.sql/
├── acme
│   ├── __init__.py
│   └── sql
│       └── __init__.py
└── setup.py

2 directories, 3 files


$ tree acme.templating/
acme.templating/
├── acme
│   ├── __init__.py
│   └── templating
│       └── __init__.py
└── setup.py

2 directories, 3 files
```

Note that for both `acme.sql` and `acme.templating`, there is an additional source file, `acme/__init__.py`. This file must be left empty. The `acme` namespace package will be created if we provide its name as a value of the `namespace_packages` keyword argument of the `setuptools.setup()` function as follows:

```
from setuptools import setup


setup(
    name='acme.templating',
    packages=['acme.templating'],
    namespace_packages=['acme'],
)
```

Easiest does not mean best. The `setuptools` module in order to register a new namespace will call for the `pkg_resources.declare_namespace()` function in your `__init__.py` file. It will happen even if the `__init__.py` file is empty. Anyway, as the official documentation says, it is your own responsibility to declare namespaces in the `__init__.py` file, and this implicit behavior of `setuptools` may be dropped in the future. In order to be safe and future-proof, you need to add the following line to the `acme/__init__.py` file:

```
__import__('pkg_resources').declare_namespace(__name__)
```

This line will make your namespace package safe from potential future changes regarding namespace packages in the `setuptools` module.

26.3 3. Uploading a package

Packages would be useless without an organized way to store, upload, and download them. Python Package Index is the main source of open source packages in the Python community. Anyone can freely upload new packages and the only requirement is to register on the PyPI site: https://pypi.python.org/pypi.

You are not, of course, limited to only this index and all Python packaging tools support the usage of alternative package repositories. This is especially useful for distributing closed source code among internal organizations or for deployment purposes. Details of such packaging usage with instructions on how to create your own package index will be explained in the next chapter. Here we focus mainly on open source uploads to PyPI, with only little mention on how to specify alternative repositories.

26.3.1 3.1. PyPI: Python Package Index

Python Package Index is, as already mentioned, the official source of open source package distributions. Downloading from it does not require any account or permission. The only thing you need is a package manager that can download new distributions from PyPI. Your preferred choice should be `pip`.

3.1.1. Uploading to PyPI

Anyone can register and upload packages to PyPI provided that he or she has an account registered. Packages are bound to the user, so, by default, only the user that registered the name of the package is its admin and can upload new distributions. This could be a problem for bigger projects, so there is an option to mark other users as package maintainers so that they are able to upload new distributions too.

The easiest way to upload a package is to use the following upload command of the setup.py script:

```
$ python setup.py <dist-commands> upload
```

Here, `<dist-commands>` is a list of commands that creates distributions to upload. Only distributions created during the same `setup.py` execution will be uploaded to the repository. So, if you upload source distribution, built distribution, and `wheel` package at once, then you need to issue the following command:

```
$ python setup.py sdist bdist bdist_wheel upload
```

When uploading using `setup.py`, you cannot reuse distributions that were already built in previous command calls and are forced to rebuild them on every upload. This may be inconvenient for large or complex projects where creation of the actual distribution may take a considerable amount of time. Another problem of `setup.py` upload is that it can use plain text HTTP or unverified HTTPS connections on some Python versions. This is why Twine is recommended as a secure replacement for the `setup.py upload` command.

Twine is the utility for interacting with PyPI that currently serves only one purpose: securely uploading packages to the repository. It supports any packaging format and always ensures that the connection is secure. It also allows you to upload files that were already created, so you are able to test distributions before release. The following example usage of twine still requires invoking the `setup.py` script for building distributions:

```
$ python setup.py sdist bdist_wheel
$ twine upload dist/*
```

3.1.2. .pypirc

`.pypirc` is a configuration file that stores information about Python packages repositories. It should be located in your home directory. The format for this file is as follows:

```
[distutils]
index-servers =
    pypi
    other

[pypi]
repository: <repository-url>
username: <username>
password: <password>

[other]
repository: https://example.com/pypi
username: <username>
password: <password>
```

The `distutils` section should have the `index-servers` variable that lists all sections describing all the available repositories and credentials for them. There are only the following three variables that can be modified for each repository section:

- `repository`: This is the URL of the package repository (it defaults to https://pypi.org/).
- `username`: This is the username for authentication in the given repository.
- `password`: This is the user password for authentication in the given repository (in plain text).

Note that storing your repository password in plain text may not be the wisest security choice. You can always leave it blank and you should be prompted for it whenever it is necessary.

The `.pypirc` file should be respected by every packaging tool built for Python. While this may not be true for every packaging-related utility out there, it is supported by the most important ones, such as `pip`, `twine`, `distutils` and `setuptools`.

26.3.2 3.2. Source packages versus built packages

There are generally the following two types of distributions for Python packages:

- Source distributions
- Built (binary) distributions

Source distributions are the simplest and most platform independent. For pure Python packages, it is a no-brainer. Such a distribution contains only Python sources and these should already be highly portable.

A more complex situation is when your package introduces some extensions written, for example, in C. Source distributions will still work provided that the package user has proper development toolchain in his/her environment. This consists mostly of the compiler and proper C header files. For such cases, the build distribution format may be better suited because it can provide already built extensions for specific platforms.

26.3.3 3.2.1. sdist

The `sdist` command is the simplest command available. It creates a release tree where everything that is needed to run the package is copied to. This tree is then archived in one or many archived files (often, it just creates one tarball). The archive is basically a copy of the source tree.

This command is the easiest way to distribute a package that would be independent from the target system. It creates a `dist/` directory for storing the archives to be distributed. Before you create the first distribution, you have to provide a `setup()` call with a version number, as follows. If you don't, `setuptools` module will assume default value of `version = '0.0.0'`:

```
from setuptools import setup
setup(name='acme.sql', version='0.1.1')
```

Every time a package is released, the version number should be increased so that the target system knows the package has changed.

Let's run the following `sdist` command for `acme.sql` package in `0.1.1` version:

```
$ python setup.py sdist
running sdist
...
creating dist
tar -cf dist/acme.sql-0.1.1.tar acme.sql-0.1.1
gzip -f9 dist/acme.sql-0.1.1.tar
removing 'acme.sql-0.1.1' (and everything under it)

$ ls dist/
acme.sql-0.1.1.tar.gz
```

Note: On Windows, the default archive type will be ZIP.

The version is used to mark the name of the archive, which can be distributed and installed on any system that has Python. In the `sdist` distribution, if the package contains C libraries or extensions, the target system is responsible for compiling them. This is very common for Linux-based systems or macOS because they commonly provide a compiler. But it is less usual to have it under Windows. That's why a package should always be distributed with a prebuilt distribution as well, when it is intended to be run on several platforms.

3.2.2. bdist and wheels

To be able to distribute a prebuilt distribution, `distutils` provides the `build` command. This commands compiles the package in the following four steps:

- `build_py`: This builds pure Python modules by byte-compiling them and copying them into the build folder.
- `build_clib`: This builds C libraries, when the package contains any, using Python compiler and creating a static library in the build folder.
- `build_ext`: This builds C extensions and puts the result in the build folder like `build_clib`.
- `build_scripts`: This builds the modules that are marked as scripts. It also changes the interpreter path when the first line was set (using `!#` prefix) and fixes the file mode so that it is executable.

Each of these steps is a command that can be called independently. The result of the compilation process is a `build` folder that contains everything needed for the package to be installed. There's no cross-compiler option yet in the `distutils` package. This means that the result of the command is always specific to the system it was built on.

When some C extensions have to be created, the build process uses the default system compiler and the Python header file (`Python.h`). This include file is available from the time Python was built from the sources. For

a packaged distribution, an extra package for your system distribution is probably required. At least in popular Linux distributions, it is often named `python-dev`. It contains all the necessary header files for building Python extensions.

The C compiler used in the build process is the compiler that is default for your operating system. For a Linux-based system or macOS, this would be gcc or clang respectively. For Windows, Microsoft Visual C++ can be used (there's a free command-line version available). The open source project MinGW can be used as well. This can be configured in `distutils`.

The `build` command is used by the `bdist` command to build a binary distribution. It invokes `build` and all the dependent commands, and then creates an archive in the same way as `sdist` does.

Let's create a binary distribution for `acme.sql` on macOS as follows:

```
$ python setup.py bdist
running bdist
running bdist_dumb
running build
...
running install_scripts
tar -cf dist/acme.sql-0.1.1.macosx-10.3-fat.tar .
gzip -f9 acme.sql-0.1.1.macosx-10.3-fat.tar
removing 'build/bdist.macosx-10.3-fat/dumb' (and everything under it)

$ ls dist/
acme.sql-0.1.1.macosx-10.3-fat.tar.gz
acme.sql-0.1.1.tar.gz
```

Notice that the newly created archive's name contains the name of the system and the distribution it was built on (macOS 10.3).

The same command invoked on Windows will create a another system, specific distribution archive as follows:

```
C:\acme.sql> python.exe setup.py bdist
...

C:\acme.sql> dir dist
25/02/2008      08:18       <DIR>       .
25/02/2008      08:18       <DIR>       ..
25/02/2008      08:24                   16 055 acme.sql-0.1.1.win32.zip
                   1 File(s)   16 055 bytes
                   2 Dir(s)    22 239 752 192 bytes free
```

If a package contains C code, apart from a source distribution, it's important to release as many different binary distributions as possible. At the very least, a Windows binary distribution is important for those who most probably don't have a C compiler installed.

A binary release contains a tree that can be copied directly into the Python tree. It mainly contains a folder that is copied into Python's `site-packages` folder. It may also contain cached bytecode files (`*.pyc` files on Python 2 and `__pycache__/*.pyc` on Python 3).

The other kind of build distributions are wheels provided by the `wheel` package. When installed (for example, using `pip`), the wheel package adds a new `bdist_wheel` command to the `distutils`. It allows creating platform specific distributions (currently only for Windows, macOS, and Linux) that are better alternatives to normal bdist distributions. It was designed to replace another distribution format introduced earlier by `setuptools` called eggs. Eggs are now obsolete, so won't be featured in here. The list of advantages of using wheels is quite long. Here are the ones that are mentioned on the Python Wheels page (http://pythonwheels.com/):

- Faster installation for pure Python and native C extension packages
- Avoids arbitrary code execution for installation. (avoids `setup.py`)
- Installation of a C extension does not require a compiler on Windows, macOS, or Linux.
- Allows better caching for testing and continuous integration.

- Creates `.pyc` files as part of the installation to ensure they match the Python interpreter used
- More consistent installs across platforms and machines

According to PyPA's recommendation, wheels should be your default distribution format. For a very long time, the binary wheels for Linux were not supported, but that has changed fortunately. Binary wheels for Linux are called manylinux wheels. The process of building them is unfortunately not as straightforward as for Windows and macOS binary wheels. For these kind of wheels, PyPA maintains special Docker images that serve as a ready-to-use build environments. For sources of these images and more information, you can visit their official repository on GitHub: https://github.com/pypa/manylinux.

26.4 4. Standalone executables

Creating standalone executables is a commonly overlooked topic in materials that cover packaging of Python code. This is mainly because Python lacks proper tools in its standard library that could allow programmers to create simple executables that could be run by users without the need to install the Python interpreter.

Compiled languages have a big advantage over Python in that they allow you to create an executable application for the given system architecture that could be run by users in a way that does not require from them any knowledge of the underlying technology. Python code, when distributed as a package, requires the Python interpreter in order to be run. This creates a big inconvenience for users who do not have enough technical proficiency.

Developer-friendly operating systems, such as macOS or most Linux distributions, come with Python interpreter preinstalled. So, for their users, the Python-based application still could be distributed as a source package that relies on a specific **interpreter directive** in the main script file that is popularly called **shebang**. For most of Python applications, this takes the following form:

```
#!/usr/bin/env python
```

Such directive when used as a first line of script will mark it to be interpreted in the default Python version for the given environment. This can, of course, take a more detailed form that requires a specific Python version such as `python3.4`, `python3`, `python2` and so on. Note that this will work in most popular POSIX systems, but isn't portable at all. This solution relies on the existence of specific Python versions and also the availability of an `env` executable exactly at `/usr/bin/env`. Both of these assumptions may fail on some operating systems. Also, shebang will not work on Windows at all. Additionally, bootstrapping of the Python environment on Windows can be a challenge even for experienced developers, so you cannot expect that nontechnical users will be able to do that by themselves.

The other thing to consider is the simple user experience in the desktop environment. Users usually expect that applications can be run from the desktop by simply clicking on them. Not every desktop environment will support that with Python applications distributed as a source.

So it would be best if we are able to create a binary distribution that would work as any other compiled executable. Fortunately, it is possible to create an executable that has both the Python interpreter and our project embedded. This allows users to open our application without caring about Python or any other dependency.

26.4.1 4.1. When standalone executables useful?

Standalone executables are useful in situations where simplicity of the user experience is more important than the user's ability to interfere with the applications code. Note that the fact that you are distributing applications as executables only makes code reading or modification harder, not impossible. It is not a way to secure application code and should only be used as a way to make interacting with the application simpler.

Standalone executables should be a preferred way of distributing applications for nontechnical end users and also seems to be the only reasonable way of distributing any Python application for Windows.

Standalone executables are usually a good choice for the following:

- Applications that depend on specific Python versions that may not be easily available on the target operating systems

- Applications that rely on modified precompiled CPython sources
- Applications with graphical interfaces
- Projects that have many binary extensions written in different languages
- Games

26.4.2 4.2. Popular tools

Python does not have any built-in support for building standalone executables. Fortunately, there are some community projects solving that problem with varied amounts of success. The following four are the most notable:

- PyInstaller
- cx_Freeze
- py2exe
- py2app

Each one of them is slightly different in use and also each one of them has slightly different limitations. Before choosing your tool, you need to decide which platform you want to target, because every packaging tool can support only a specific set of operating systems.

It is best if you make such a decision at the very beginning of the project's life. None of these tools, of course, requires deep interaction in your code, but if you start building standalone packages early, you can automate the whole process and save future integration time and costs. If you leave this for later, you may find yourself in a situation where the project is built in such a sophisticated way that none of the available tools will work. Providing a standalone executable for such a project will be problematic and will take a lot of your time.

4.2.1. PyInstaller

PyInstaller (http://www.pyinstaller.org) is by far the most advanced program to freeze Python packages into standalone executables. It provides the most extensive multiplatform compatibility among every available solution at the moment, so it is the most highly recommended one. PyInstaller supports the following platforms:

- Windows (32-bit and 64-bit)
- Linux (32-bit and 64-bit)
- macOS (32-bit and 64-bit)
- FreeBSD, Solaris, and AIX

Supported versions of Python are Python 2.7 and Python 3.3, 3.4, and 3.5. It is available on PyPI, so it can be installed in your working environment using pip. If you have problems installing it this way, you can always download the installer from the project's page.

Unfortunately, cross-platform building (cross-compilation) is not supported, so if you want to build your standalone executable for a specific platform, then you need to perform building on that platform. This is not a big problem today with the advent of many virtualization tools. If you don't have a specific system installed on your computer, you can always use Vagrant, which will provide you with the desired operating system as a virtual machine.

Usage for simple applications is pretty straightforward. Let's assume our application is contained in the script named `myscript.py` . This is a simple hello world application. We want to create a standalone executable for Windows users and we have our sources located under `D://dev/app` in the filesystem. Our application can be bundled with the following short command:

```
$ pyinstaller myscript.py
2121 INFO: PyInstaller: 3.1
2121 INFO: Python: 2.7.10
2121 INFO: Platform: Windows-7-6.1.7601-SP1
```

(continues on next page)

(continued from previous page)

```
2121 INFO: wrote D:\dev\app\myscript.spec
2137 INFO: UPX is not available.
2138 INFO: Extending PYTHONPATH with paths ['D:\\dev\\app', 'D:\\dev\\app']
2138 INFO: checking Analysis
2138 INFO: Building Analysis because out00-Analysis.toc is non existent
2138 INFO: Initializing module dependency graph...
2154 INFO: Initializing module graph hooks...
2325 INFO: running Analysis out00-Analysis.toc
(...)
25884 INFO: Updating resource type 24 name 2 language 1033
```

PyInstaller's standard output is quite long, even for simple applications, so it was truncated in the preceding example for the sake of brevity. If run on Windows, the resulting structure of directories and files will be as follows:

```
$ tree /0066
|   ├── myscript.py
|   └── myscript.spec
|
├──build
|   └──myscript
|       ├── myscript.exe
|       ├── myscript.exe.manifest
|       ├── out00-Analysis.toc
|       ├── out00-COLLECT.toc
|       ├── out00-EXE.toc
|       ├── out00-PKG.pkg
|       ├── out00-PKG.toc
|       ├── out00-PYZ.pyz
|       ├── out00-PYZ.toc
|       └── warnmyscript.txt
|
└──dist
    └──myscript
        ├── bz2.pyd
        ├── Microsoft.VC90.CRT.manifest
        ├── msvcm90.dll
        ├── msvcp90.dll
        ├── msvcr90.dll
        ├── myscript.exe
        ├── myscript.exe.manifest
        ├── python27.dll
        ├── select.pyd
        ├── unicodedata.pyd
        └──_hashlib.pyd
```

The `dist/myscript` directory contains the built application that can now be distributed to the users. Note that whole directory must be distributed. It contains all the additional files that are required to run our application (DLLs, compiled extension libraries, and so on). A more compact distribution can be obtained with the `--onefile` switch of the `pyinstaller` command as follows:

```
$ pyinstaller --onefile myscript.py
(...)

$ tree /f
├──build
|   └──myscript
|       ├── myscript.exe.manifest
|       ├── out00-Analysis.toc
|       ├── out00-EXE.toc
|       ├── out00-PKG.pkg
```

(continues on next page)

(continued from previous page)

```
│            ├── out00-PKG.toc
│            ├── out00-PYZ.pyz
│            ├── out00-PYZ.toc
│            └── warnmyscript.txt
│
└──dist
   └──myscript
            └── myscript.exe
```

When built with the `--onefile` option, the only file you need to distribute to other users is the single executable found in the `dist` directory (here, `myscript.exe`). For small applications, this is probably the preferred option.

One of the side effects of running the `pyinstaller` command is the creation of the `*.spec` file. This is an auto generated Python module containing specification on how to create executables from your sources. This is the example specification file created automatically for `myscript.py` code:

```
# -*- mode: python -*-
block_cipher = None
a = Analysis(['myscript.py'],
             pathex=['D:\\dev\\app'],
             binaries=None,
             datas=None,
             hiddenimports=[],
             hookspath=[],
             runtime_hooks=[],
             excludes=[],
             win_no_prefer_redirects=False,
             win_private_assemblies=False,
             cipher=block_cipher)
pyz = PYZ(a.pure, a.zipped_data, cipher=block_cipher)
exe = EXE(pyz,
          a.scripts,
          a.binaries,
          a.zipfiles,
          a.datas,
          name='myscript',
          debug=False,
          strip=False,
          upx=True,
          console=True)
```

This `.spec` file contains all `pyinstaller` arguments specified earlier. This is very useful if you have performed a lot of customizations to your build. Once created, you can use it as an argument to the `pyinstaller` command instead of your Python script as follows:

```
$ pyinstaller.exe myscript.spec
```

Note that this is a real Python module, so you can extend it and perform more complex customizations to the building procedure. Customizing the `.spec` file is especially useful when you are targeting many different platforms. Also, not all of the `pyinstaller` options are available through the command-line interface and can be used only when modifying `.spec` file.

PyInstaller is an extensive tool, which by its usage is very simple for the great majority of programs. Anyway, thorough reading of its documentation is recommended if you are interested in using it as a tool to distribute your applications.

4.2.2. cx_Freeze

cx_Freeze (http://cx-freeze.sourceforge.net/) is another tool for creating standalone executables. It is a simpler solution than PyInstaller, but also supports the following three major platforms:

- Windows
- Linux
- macOS

Like PyInstaller, it does not allow you to perform cross-platform builds, so you need to create your executables on the same operating system you are distributing to. The major disadvantage of cx_Freeze is that it does not allow you to create real single-file executables. Applications built with it need to be distributed with related DLL files and libraries. Assuming that we have the same application as featured in the PyInstaller section, the example usage is very simple as well:

```
$ cxfreeze myscript.py
copying C:\Python27\lib\site-packages\cx_Freeze\bases\Console.exe ->
D:\dev\app\dist\myscript.exe
copying C:\Windows\system32\python27.dll ->
D:\dev\app\dist\python27.dll
writing zip file D:\dev\app\dist\myscript.exe
(...)
copying C:\Python27\DLLs\bz2.pyd -> D:\dev\app\dist\bz2.pyd
copying C:\Python27\DLLs\unicodedata.pyd -> D:\dev\app\dist\unicodedata.pyd
```

Resulting structure of files is as follows:

```
$ tree /f
|    └── myscript.py
|
└──dist
    ├── bz2.pyd
    ├── myscript.exe
    ├── python27.dll
    └── unicodedata.pyd
```

Instead of providing the own format for build specification (like PyInstaller does), cx_Freeze extends the distutils package. This means you can configure how your standalone executable is built with the familiar setup.py script. This makes cx_Freeze very convenient if you already distribute your package using setuptools or distutils because additional integration requires only small changes to your setup.py script. Here is an example of such a setup.py script using cx_Freeze.setup() for creating standalone executables on Windows:

```
import sys
from cx_Freeze import setup, Executable

# Dependencies are automatically detected, but it might need fine tuning.
build_exe_options = {"packages": ["os"], "excludes": ["tkinter"]}
setup(
    name="myscript",
    version="0.0.1",
    description="My Hello World application!",
    options={
        "build_exe": build_exe_options
    },
    executables=[Executable("myscript.py")]
)
```

With such a file, the new executable can be created using the new `build_exe` command added to the `setup.py` script as follows:

```
$ python setup.py build_exe
```

The usage of `cx_Freeze` seems a bit easier than PyInstaller's, and `distutils` integration is a very useful feature. Unfortunately this project may cause some trouble for inexperienced developers due to the following reasons:

- Installation using pip may be problematic under Windows.
- The official documentation is very brief and lacking in some places.

4.2.3. py2exe and py2app

py2exe (http://www.py2exe.org) and py2app (https:://py2app.readthedocs.io/en/latest/) are two complementary programs that integrate with Python packaging either via `distutils` or `setuptools` in order to create standalone executables. Here they are mentioned together because they are very similar in both usage and their limitations. The major drawback of py2exe and py2app is that they target only a single platform:

- py2exe allows building Windows executables.
- py2app allows building macOS apps.

Because the usage is very similar and requires only modification of the setup.py script, these packages complement each other. The documentation of the py2app project provides the following example of the setup.py script, which allows you to build standalone executables with the right tool (either py2exe or py2app) depending on the platform used:

```
import sys
from setuptools import setup


mainscript = 'MyApplication.py'

if sys.platform == 'darwin':
    extra_options = dict(
        setup_requires=['py2app'],
        app=[mainscript],
        # Cross-platform applications generally expect sys.argv to
        # be used for opening files.
        options=dict(py2app=dict(argv_emulation=True))
    )
elif sys.platform == 'win32':
    extra_options = dict(
        setup_requires=['py2exe'],
        app=[mainscript]
    )
else:
    extra_options = dict(
    # Normally unix-like platforms will use "setup.py install"
    # and install the main script as such
        scripts=[mainscript],
    )
    setup(
        name="MyApplication",
        **extra_options
    )
```

With such a script, you can build your Windows executable using the python `setup.py py2exe` command and macOS app using `python setup.py py2app`. Cross-compilation is, of course, not possible.

Despite py2app and py2exe having obvious limitations and offering less elasticity than PyInstaller or cx_Freeze, it is always good to be familiar with them. In some cases, PyInstaller or cx_Freeze might fail to build the executable for the project properly. In such situations, it is always worth checking whether other solutions can handle your code.

4.3. Security of Python code in executable packages

It is important to know that standalone executables do not make the application code secure by any means. It is not an easy task to decompile the embedded code from such executable files, but it is definitely doable. What is even more important is that the results of such decompilation (if done with proper tools) might look strikingly similar to original sources.

This fact makes standalone Python executables not a viable solution for closed source projects where leaking of the application code could harm the organization. So, if your whole business can be copied simply by copying the source code of you application, then you should think of other ways to distribute the application. Maybe providing software as a service will be a better choice for you.

4.3.1. Making decompilation harder

As already said, there is no reliable way to secure applications from decompilation with the tools available at the moment. Still, there are some ways to make this process harder. But harder does not mean less probable. For some of us, the most tempting challenges are the hardest ones. And we all know that the eventual price in this challenge is very high—the code that you tried to secure.

Usually the process of decompilation consists of the following steps:

1. Extracting the project's binary representation of bytecode from standalone executables
2. Mapping of a binary representation to bytecode of a specific Python version
3. Translation of bytecode to AST
4. Re-creation of sources directly from AST

Providing the exact solutions for deterring developers from such reverse engineering of standalone executables would be pointless for obvious reasons. So here are only some ideas for hampering the decompilation process or devaluing its results:

- Removing any code metadata available at runtime (docstrings) so the eventual results will be a bit less readable.
- Modifying the bytecode values used by the CPython interpreter; so conversion from binary to bytecode and later to AST requires more effort.
- Using a version of CPython sources modified in such a complex way that even if decompiled sources of the application are available, they are useless without decompiling the modified CPython binary.
- Using obfuscation scripts on sources before bundling them into an executable, which will make sources less valuable after the decompilation.

Such solutions make the development process a lot harder. Some of the preceding ideas require a very deep understanding of Python runtime, but each one of them is riddled with many pitfalls and disadvantages. Mostly, they only defer what is anyway inevitable. Once your trick is broken, it renders all your additional efforts a waste of time and resources.

The only reliable way to not allow your closed code to leak outside of your application is to not ship it directly to users in any form. And this is only possible if other aspects of your organization security stay airtight.

CHAPTER

TWENTYSEVEN

DEPLOYMENT

Even perfect code (if it exists) is useless if it is not able to run. So, in order to serve any purpose, our code needs to be installed on the target machine (computer) and executed. The process of making a specific version of your application or service available to end users is called deployment.

In the case of desktop applications, this seems to be simple as your job ends with providing a downloadable package with an optional installer, if necessary. It is the user's responsibility to download and install the package in their environment. Your responsibility is to make this process as easy and convenient as possible. Proper packaging is still not a simple task, but some tools were already explained in the previous chapter.

Surprisingly, things get more complicated when your code is not a standalone product. If your application only provides a service that is being sold to users, then it is your responsibility to run it on your own infrastructure. This scenario is typical for a web application or any X as a service product. In such a situation, the code is deployed to set off remote machines that are physically accessible to the developers. This is especially true if you are already a user of cloud computing services such as Amazon Web Services (AWS) or Heroku.

27.1 1. The Twelve-factor app

The main requirement for painless deployment is building your application in a way that ensures that this process will be simple and as streamlined as possible. This is mostly about removing obstacles and encouraging well-established practices. Following such common practices is especially important in organizations where only specific people are responsible for development (the developers team or Dev for short) and different people are responsible for deploying and maintaining the execution environments (the operations team or Ops for short).

All tasks related to server maintenance, monitoring, deployment, configuration, and so on are often put into one single bag called **operations**. Even in organizations that have no separate teams for operational tasks, it is common that only some of the developers are authorized to do deployment tasks and maintain the remote servers. The common name for such a position is DevOps. Also, it isn't such an unusual situation that every member of the development team is responsible for operations, so everyone in such a team can be called DevOps.

No matter how your organization is structured and what the responsibilities of each developer are, everyone should know how operations work and how code is deployed to the remote servers because, in the end, the execution environment and its configuration is a hidden part of the product you are building.

The following common practices and conventions are important mainly for the following reasons:

- At every company people quit and new ones are hired. By using best approaches, you are making it easier for fresh team members to jump into the project. You can never be sure that new employees are already familiar with common practices for system configuration and running applications in a reliable way, but you can at least make their fast adaptation more probable.
- In organizations where only some people are responsible for deployments, it simply reduces the friction between the operations and development teams.

A good source of such practices that encourage building easy deployable apps is a manifesto called the **Twelve-Factor App**. It is a general language-agnostic methodology for building software-as-a-service apps. One of its purposes is making applications easier to deploy, but it also highlights other topics such as maintainability or making applications easier to scale.

As its name says, the Twelve-Factor App consists of 12 rules:

- **Code base**: One code base tracked in revision control and many deploys
- **Dependencies**: Explicitly declare and isolate dependencies
- **Config**: Store configurations in the environment
- **Backing services**: Treat backing services as attached resources
- **Build, release, run**: Strictly separate build and run stages
- **Processes**: Execute the app as one or more stateless processes
- **Port binding**: Export services via port binding
- **Concurrency**: Scale out via the process model
- **Disposability**: Maximize robustness with fast startup and graceful shutdown
- **Dev/prod parity**: Keep development, staging, and production as similar as possible
- **Logs**: Treat logs as event streams
- **Admin processes**: Run administration/management tasks as one-off processes

Extending each of these rules here is a bit pointless because the official page of Twelve- Factor App methodology (http://12factor.net/) contains extensive rationale for each app factor with examples of tools for different frameworks and environments. This chapter tries to stay consistent with the preceding manifesto, so we will discuss some of them in detail when necessary. The techniques and examples that are presented may sometimes slightly diverge from these 12 factors, but remember that these rules are not carved in stone. They are great as long as they serve the purpose. In the end, what matters is the working application (product) and not being compatible with some arbitrary methodology.

27.2 2. Approaches to deployment automation

With the advent of application containerization (Docker and similar technologies), modern software provisioning tools (for example, Puppet, Chef, Ansible, and Salt), and infrastructure management systems (for example, Terraform and SaltStack) development and operations teams have a variety of ways in which they can organize and manage their code deployments and configuration of remote systems. Each solution has pros and cons, so advanced automation tools should be chosen very wisely with respect to the favored development processes and methodologies.

Fast paced teams that use microservice architecture and deploy code often (maybe even simultaneously in parallel versions) will definitely favor container orchestration systems such as Kubernetes or use dedicated services provided by their cloud vendor (for example, AWS). Teams that build old-style big monolithic applications and run them on their own bare-metal servers might want to use more low-level automation and software provisioning systems. Actually, there is no rule and you can find teams of every size using every possible approach to software provisioning, code deployments, and application orchestration. The limiting factors here are resources and knowledge.

That's why it's really hard to briefly provide a set of common tools and solutions that would fit the needs and capabilities of every developer and every team. Because of that, in this chapter, we will focus only on a pretty simple approach to automation using Fabric. We could say that this is outdated. And that's probably true. What seem to be the most modern are container orchestrations systems in the style of Kubernetes that allow you to leverage Docker containers for fast, maintainable, scalable, and reproducible environments. But these systems have quite a steep learning curve and it's impossible to introduce them in just a few sections of a single chapter. Fabric, on the other hand, is very simple and easy to grasp so it is a really great tool to introduce someone to the concept of automation.

27.2.1 2.1. Using Fabric for deployment automation

For very small projects, it may be possible to deploy your code by hand, that is, by manually typing the sequence of commands through the remote shells that are necessary to install a new version of code and execute it on a remote shell. Anyway, even for an average-sized project, this is error-prone and tedious and should be considered a waste of most of the precious resource you have, your own time.

The solution for that is automation. The simple thumb rule could be the following:

"If you needed to perform the same task manually at least twice, you should automate it so you won't need to do it for the third time."

There are various tools that allow you to automate different things, including the following:

- Remote execution tools such as Fabric are used for on-demand automated execution of code on multiple remote hosts.
- Configuration management tools such as Chef, Puppet, CFEngine, Salt, and Ansible are designed for automatized configuration of remote hosts (execution environments). They can be used to set up backing services (databases, caches, and so on), system permissions, users, and so on. Most of them can be used also as a tool for remote execution (such as Fabric) but, depending on their architecture, this may be more or less convenient.

Configuration management solutions is a complex topic. The truth is that the simplest remote execution frameworks have the lowest entry barrier and are the most popular choice, at least for small projects. In fact, every configuration management tool that provides a way to declaratively specify configuration of your machines has a remote execution layer implemented somewhere deep inside.

Also, some configuration management tools may not be best suited for actual automated code deployment. One such example is Puppet, which really discourages the explicit running of any shell commands. This is why many people choose to use both types of solution to complement each other: configuration management for setting up a system-level environment and on-demand remote execution for application deployment.

Fabric (http://www.fabfile.org/) is so far the most popular solution used by Python developers to automate remote execution. It is a Python library and command-line tool for streamlining the use of SSH for application deployment or systems administration tasks. We will focus on it because it is relatively easy to start with. Keep in mind that, depending on your needs, it may not be the best solution to your problems. Anyway, it is a great example of a utility that can add some automation to your operations, if you don't have any yet.

You could, of course, automate all of the work using only Bash scripts but this is very tedious and error-prone. Python has more convenient ways for string processing and encourages code modularization. Fabric is in fact only a tool for gluing the execution of commands via SSH. It means that you still need to know how to use the command-line interface and its utilities in your remote environment.

So, if you want to strictly follow the Twelve-Factor App methodology, you should not maintain its code in the source tree of the deployed application.

Complex projects are, in fact, very often built from various components maintained as separate code bases, so this is another reason why it is a good approach to have one separate repository for all of the project component configurations and Fabric scripts. This makes deployment of different services more consistent and encourages good code reuse.

To start working with Fabric, you need to install the `fabric` package (using `pip`) and create a script named `fabfile.py`. That script is usually located in the root of your project. Note that `fabfile.py` can be considered a part of your project configuration.

But before we create our `fabfile` let's define some initial utilities that will help us to set up the project remotely. Here's a module that we will call `fabutils`:

```
import os


# Let's assume we have private package repository created
# using 'devpi' project
```

(continues on next page)

(continued from previous page)

```
PYPI_URL = 'http://devpi.webxample.example.com'

# This is arbitrary location for storing installed releases.
# Each release is a separate virtual environment directory
# which is named after project version. There is also a
# symbolic link 'current' that points to recently deployed
# version. This symlink is an actual path that will be used
# for configuring the process supervision tool for example:
# .
# ├── 0.0.1
# ├── 0.0.2
# ├── 0.0.3
# ├── 0.1.0
# └── current -> 0.1.0/

REMOTE_PROJECT_LOCATION = "/var/projects/webxample"
def prepare_release(c):
    """ Prepare a new release by creating source distribution and
    uploading to out private package repository
    """
    c.local(f'python setup.py build sdist')
    c.local(f'twine upload --repository-url {PYPI_URL}')

def get_version(c):
    """ Get current project version from setuptools """
    return c.local('python setup.py --version').stdout.strip()

def switch_versions(c, version):
    """ Switch versions by replacing symlinks atomically """
    new_version_path = os.path.join(REMOTE_PROJECT_LOCATION, version)
    temporary = os.path.join(REMOTE_PROJECT_LOCATION, 'next')
    desired = os.path.join(REMOTE_PROJECT_LOCATION, 'current')

    # force symlink (-f) since probably there is a one already
    c.run(f"ln -fsT {new_version_path} {temporary}")
    # mv -T ensures atomicity of this operation
    c.run(f"mv -Tf {temporary} {desired}" )
```

An example of a final fabfile that defines a simple deployment procedure will look like this:

```
from fabric import task
from .fabutils import *


@task
def uptime(c):
    """
    Run uptime command on remote host - for testing connection.
    """
    c.run("uptime")

@task
def deploy(c):
    """ Deploy application with packaging in mind """
    version = get_version(c)

    pip_path = os.path.join(
        REMOTE_PROJECT_LOCATION, version, 'bin', 'pip'
    )

    if not c.run(f"test -d {REMOTE_PROJECT_LOCATION}", warn=True):
```

(continues on next page)

(continued from previous page)

```
        # it may not exist for initial deployment on fresh host
        c.run(f"mkdir -p {REMOTE_PROJECT_LOCATION}")

    with c.cd(REMOTE_PROJECT_LOCATION):
        # create new virtual environment using venv
        c.run(f'python3 -m venv {version}')
        c.run(f"{pip_path} install webxample=={version} --index-url {PYPI_URL}")

    switch_versions(c, version)
    # let's assume that Circus is our process supervision tool
    # of choice.
    c.run('circusctl restart webxample')
```

Every function decorated with `@task` is now treated as an available subcommand to the fab utility provided with the fabric package. You can list all of the available subcommands using the `-l` or `--list` switch. The code is shown in the following snippet:

```
$ fab --list
Available commands:
    deploy Deploy application with packaging in mind
    uptime Run uptime command on remote host - for testing connection.
```

Now, you can deploy the application to the given environment type with only the following single shell command:

```
$ fab -H myhost.example.com deploy
```

Note that the preceding `fabfile` serves only illustrative purposes. In your own code, you might want to provide extensive failure handling and try to reload the application without the need to restart the web worker process. Also, some of the techniques presented here may not be obvious right now but will be explained later in this chapter. These include the following:

- Deploying an application using the private package repository
- Using Circus for process supervision on the remote host

27.3 3. Index mirroring

These are three main reasons why you might want to run your own index of Python packages:

- The official Python Package Index does not have any availability guarantees. It is run by Python Software Foundation thanks to numerous donations. Because of that, it means that this site can be down at the most inconvenient time. You don't want to stop your deployment or packaging process in the middle due to PyPI outage.
- It is useful to have reusable components written in Python properly packaged, even for the closed source that will never be published publicly. It simplifies code base because packages that are used across the company for different projects do not need to be vendored. You can simply install them from the repository. This simplifies maintenance for such shared code and might reduce development costs for the whole company if it has many teams working on different projects.
- It is a very good practice to have your entire project packaged using `setuptools`. Then, deployment of the new application version is often as simple as running `pip install --update my-application`.

Tip: **Code vendoring** is a practice of including sources of the external package in the source code (repository) of other projects. It is usually done when the project's code depends on a specific version of some external package that may also be required by other packages (and in a completely different version).

For instance, the popular `requests` package uses some version of `urllib3` in its source tree because it is very tightly coupled to it and is very unlikely to work with any other version of urllib3 . An example of a module that is

particularly often used by others is `six`. It can be found in sources of numerous popular projects such as Django (`django.utils.six`), Boto (`boto.vendored.six`), or Matplotlib (`matplotlib.externals.six`).

Although vendoring is practiced even by some large and successful open source projects, it should be avoided if possible. This has justifiable usage only in certain circumstances and should not be treated as a substitute of package dependency management.

27.3.1 3.1. PyPI mirroring

The problem of PyPI outages can be somehow mitigated by allowing the installation tools to download packages from one of its mirrors. In fact, the official Python Package Index is already served through **Content Delivery Network (CDN)**, so it is intrinsically mirrored. This does not change the fact that it seems to have some bad days from time to time. Using unofficial mirrors is not a solution here because it might raise some security concerns.

The best solution is to have your own PyPI mirror that will have all of the packages you need. The only party that will use it is you, so it will be much easier to ensure proper availability. The other advantage is that whenever this service goes down, you don't need to rely on someone else to bring it up. The mirroring tool maintained and recommended by PyPA is **bandersnatch** (https://pypi.python.org/pypi/bandersnatch). It allows you to mirror the whole content of Python Package Index and it can be provided as the `index-url` option for the repository section in the `.pypirc` file (as explained in the previous chapter). This mirror does not accept uploads and does not have the web part of PyPI. Anyway, beware! A full mirror might require hundreds of gigabytes of storage and its size will continue to grow over time.

But why stop at a simple mirror while we have a much better alternative? There is a very low chance that you will require a mirror of the whole package index. Even with a project that has hundreds of dependencies, it will be only a minor fraction of all of the available packages. Also, not being able to upload your own private package is a huge limitation of such a simple mirror. It seems that the added value of using bandersnatch is very low for such a high price. And this is true in most situations. If the package mirror is to be maintained only for single or a few projects, a much better approach is to use **devpi** (http://doc.devpi.net/). It is a PyPI-compatible package index implementation that provides both of the following:

- A private index to upload nonpublic packages
- Index mirroring

The main advantage of devpi over bandersnatch is how it handles mirroring. It can, of course, do a full general mirror of other indexes like bandersnatch does, but it is not its default behavior. Instead of doing a rather expensive backup of the whole repository, it maintains mirrors for packages that were already requested by clients. So, whenever a package is requested by the installation tool (`pip`, `setuptools`, and `easy_install`), if it does not exist in the local mirror, the devpi server will attempt to download it from the mirrored index (usually PyPI) and serve. Once the package is downloaded, the devpi will periodically check for its updates to maintain a fresh state of its mirror.

The mirroring approach leaves a slight risk of failure when you request a new package that was not yet mirrored when the upstream package index has an outage. Anyway, this risk is reduced, thanks to the fact that in most deployments you will depend only on packages that were already mirrored in the index. The mirror state for packages that were already requested has eventual consistency guarantee and new versions will be downloaded automatically. This seems to be a very reasonable trade off.

Now let's see how to properly bundle and build additional non-Python resources in your Python application.

27.3.2 3.2. Bundling additional resources with your Python package

Modern web applications have a lot of dependencies and often require a lot of steps to properly install on the remote host. For instance, the typical bootstrapping process for a new version of the application on a remote host consists of the following steps:

1. Create a new virtual environment for isolation.
2. Move the project code to the execution environment.
3. Install the latest project requirements (usually from the `requirements.txt` file).
4. Synchronize or migrate the database schema.
5. Collect static files from project sources and external packages to the desired location.
6. Compile localization files for applications available in different languages.

For more complex sites, there might be lot of additional tasks mostly related to frontend code that is independent from previously defined tasks, as in the following example:

1. Generate CSS files using preprocessors such as SASS or LESS.
2. Perform minification, obfuscation, and/or concatenation of static files (JavaScript and CSS files).
3. Compile code written in JavaScript superset languages (CoffeeScript, TypeScript, and so on) to native JS.
4. Preprocess response template files (minification, style inlining, and so on).

Nowadays, for these kind of applications that require a lot of additional assets to be prepared, most developers would probably use Docker images. Dockerfiles allow you to easily define all of the steps that are necessary to bundle all assets with your application image. But if you don't use Docker, it means that all of these steps must be automated using other tools such as Make, Bash, Fabric, or Ansible. Still, it is not a good idea to do all of these steps directly on the remote hosts where the application is being installed. Here are the reasons:

- Some of the popular tools for processing static assets can be either CPU or memory intensive. Running them in production environments can destabilize your application execution.
- These tools very often will require additional system dependencies that may not be required for the normal operation of your projects. These are mostly additional runtime environments such as JVM, Node, or Ruby. This adds complexity to configuration management and increases the overall maintenance costs.
- If you are deploying your application to multiple servers (tens, hundreds, or thousands), you are simply repeating a lot of work that could be done once. If you have your own infrastructure, then you may not experience the huge increase of costs, especially if you perform deployments in periods of low traffic. But if you run cloud computing services in the pricing model that charges you extra for spikes in load or generally for execution time, then this additional cost may be substantial on a proper scale.
- Most of these steps just take a lot of time. You are installing your code on remote servers, so the last thing you want is to have your connection interrupted by some network issue. By keeping the deployment process quick, you are lowering the chance of deployment interruption.

Obviously, the results of these predeployment steps can't be included in your application code repository either. Simply, there are things that must be done with every release and you can't change that. It is obviously a place for proper automation but the clue is to do it in the right place and at the right time.

Most of the things, such as static collection and code/asset preprocessing, can be done locally or in a dedicated environment, so the actual code that is deployed to the remote server requires only a minimal amount of on-site processing. The following are the most notable of such deployment steps, either in the process of building distribution or installing a package:

1. Installation of Python dependencies and transferring of static assets (CSS files and JavaScript) to the desired location can be handled as a part of the `install` command of the `setup.py` script.
2. Preprocessing of code (processing JavaScript supersets, minification/obfuscation/concatenation of assets, and running SASS or LESS) and things such as localized text compilation (for example, `compilemessages` in Django) can be a part of the `sdist`/`bdist` command of the `setup.py` script.

Inclusion of preprocessed code other than Python can be easily handled with the proper `MANIFEST.in` file. Dependencies are, of course, best provided as an `install_requires` argument of the `setup()` function call from the setuptools package.

Packaging the whole application, of course, will require some additional work from you, such as providing your own custom `setuptools` commands or overriding the existing ones, but it gives you a lot of advantages and makes project deployment a lot faster and reliable.

Let's use a Django-based project (in Django 1.9 version) as an example. I have chosen this framework because it seems to be the most popular Python project of this type, so there is a high chance that you already know it a bit. A typical structure of files in such a project might look like the following:

```
$ tree . -I __pycache__ --dirsfirst
.
├── webxample
│   ├── conf
│   │   ├── __init__.py
│   │   ├── settings.py
│   │   ├── urls.py
│   │   └── wsgi.py
│   ├── locale
│   │   ├── de
│   │   │   └── LC_MESSAGES
│   │   │       └── django.po
│   │   ├── en
│   │   │   └── LC_MESSAGES
│   │   │       └── django.po
│   │   └── pl
│   │       └── LC_MESSAGES
│   │           └── django.po
│   ├── myapp
│   │   ├── migrations
│   │   │   └── __init__.py
│   │   ├── static
│   │   │   ├── js
│   │   │   │   └── myapp.js
│   │   │   └── sass
│   │   │       └── myapp.scss
│   │   ├── templates
│   │   │   ├── index.html
│   │   │   └── some_view.html
│   │   ├── __init__.py
│   │   ├── admin.py
│   │   ├── apps.py
│   │   ├── models.py
│   │   ├── tests.py
│   │   └── views.py
│   ├── __init__.py
│   └── manage.py
├── MANIFEST.in
├── README.md
└── setup.py
15 directories, 23 files
```

Note that this slightly differs from the usual Django project template. By default, the name of the package that contains the WSGI application, the settings module, and the URL configuration has the same name as the project. Because we decided to take the packaging approach, this would be named as `webxample`. This can cause some confusion, so it is better to rename it to `conf`. Without digging into the possible implementation details, let's just make the following few simple assumptions:

- Our example application has some external dependencies. Here, it will be two popular Django packages: `djangorestframework` and `django-allauth`, plus one non-Django package: `gunicorn`.
- `djangorestframework` and `django-allauth` are provided as `INSTALLED_APPS` in the

`webexample.webexample.settings` module.

- The application is localized in three languages (German, English, and Polish) but we don't want to store the compiled `gettext` messages in the repository.
- We are tired of vanilla CSS syntax, so we decided to use a more powerful SCSS language that we translate into CSS using SASS.

Knowing the structure of the project, we can write our `setup.py` script in a way that makes `setuptools` handle the following:

- Compilation of SCSS files under `webxample/myapp/static/scss`
- Compilation of `gettext` messages under `webexample/locale` from `.po` to `.mo` format
- Installation of the requirements
- A new script that provides an entry point to the package, so we will have the custom command instead of the `manage.py` script

We have a bit of luck here: Python binding for `libsass`, a C/C++ port of the SASS engine, provides some integration with `setuptools` and `distutils`. With only a little configuration, it provides a custom setup.py command for running the SASS compilation. This is shown in the following code:

```
from setuptools import setup
setup(
    name='webxample',
    setup_requires=['libsass == 0.6.0'],
    sass_manifests={
    'webxample.myapp': ('static/sass', 'static/css')
    }
)
```

So, instead of running the `sass` command manually or executing a subprocess in the `setup.py` script, we can type python `setup.py build_scss` and have our SCSS files compiled to CSS. This is still not enough. It makes our life a bit easier but we want the whole distribution fully automated so there is only one step for creating new releases. To achieve this goal, we are forced to override some of the existing `setuptools` distribution commands.

The example `setup.py` file that handles some of the project preparation steps through packaging might look like this:

```
import os
from setuptools import setup
from setuptools import find_packages
from distutils.cmd import Command
from distutils.command.build import build as _build

try:
    from django.core.management.commands.compilemessages \
    import Command as CompileCommand
except ImportError:
    # note: during installation django may not be available
    CompileCommand = None
    # this environment is requires


os.environ.setdefault(
    "DJANGO_SETTINGS_MODULE", "webxample.conf.settings"
)


class build_messages(Command):
    """ Custom command for building gettext messages in Django """
    description = """compile gettext messages"""
```

(continues on next page)

(continued from previous page)

```
    user_options = []

    def initialize_options(self):
        pass

    def finalize_options(self):
        pass

    def run(self):
        if CompileCommand:
            CompileCommand().handle(
                verbosity=2, locales=[], exclude=[]
            )
        else:
            raise RuntimeError("could not build translations")


class build(_build):
    """ Overridden build command that adds additional build steps """
    sub_commands = [
        ('build_messages', None),
        ('build_sass', None),
    ] + _build.sub_commands

    setup(
        name='webxample',
        setup_requires=[
            'libsass == 0.6.0',
            'django == 1.9.2'
        ],
        install_requires=[
            'django == 1.9.2',
            'gunicorn == 19.4.5',
            'djangorestframework == 3.3.2',
            'django-allauth == 0.24.1'
        ],
        packages=find_packages('.'),
        sass_manifests={
        '   webxample.myapp': ('static/sass', 'static/css')
        },
        cmdclass={
            'build_messages': build_messages,
            'build': build
        },
        entry_points={
            'console_scripts': {
                'webxample = webxample.manage:main'
            }
        }
    )
```

With such an implementation, we can build all assets and create the source distribution of a package for the `webxample` project using the following single Terminal command:

```
$ python setup.py build sdist
```

If you already have your own package index (created with devpi), you can add the `install` subcommand or use `twine` so this package will be available for installation with `pip` in your organization. If we look into a structure of source distribution created with our `setup.py` script, we can see that it contains the following compiled `gettext` messages and CSS style sheets generated from SCSS files:

```
$ tar -xvzf dist/webxample-0.0.0.tar.gz 2> /dev/null
$ tree webxample-0.0.0/ -I __pycache__ --dirsfirst
webxample-0.0.0/
├── webxample
│   ├── conf
│   │   ├── __init__.py
│   │   ├── settings.py
│   │   ├── urls.py
│   │   └── wsgi.py
│   ├── locale
│   │   ├── de
│   │   │   └── LC_MESSAGES
│   │   │       ├── django.mo
│   │   │       └── django.po
│   │   ├── en
│   │   │   └── LC_MESSAGES
│   │   │       ├── django.mo
│   │   │       └── django.po
│   │   └── pl
│   │       └── LC_MESSAGES
│   │           ├── django.mo
│   │           └── django.po
│   ├── myapp
│   │   ├── migrations
│   │   │   └── __init__.py
│   │   ├── static
│   │   │   ├── js
│   │   │   │   └── myapp.js
│   │   │   └── sass
│   │   │       └── myapp.scss.css
│   │   ├── templates
│   │   │   ├── index.html
│   │   │   └── some_view.html
│   │   ├── __init__.py
│   │   ├── admin.py
│   │   ├── apps.py
│   │   ├── models.py
│   │   ├── tests.py
│   │   └── views.py
│   ├── __init__.py
│   └── manage.py
├── webxample.egg-info
│   ├── PKG-INFO
│   ├── SOURCES.txt
│   ├── dependency_links.txt
│   ├── requires.txt
│   └── top_level.txt
├── MANIFEST.in
├── README.md
└── setup.py
16 directories, 33 files
```

The additional benefit of using this approach is that we were able to provide our own entry point for the project in place of Django's default `manage.py` script. Now, we can run any Django management command using this entry point, for instance:

```
$ webxample migrate
$ webxample collectstatic
$ webxample runserver
```

This required a little change in the `manage.py` script for compatibility with the `entry_points` argument in `setup()`, so the main part of its code is wrapped with the `main()` function call. This is shown in the following

code:

```
#!/usr/bin/env python3
import os
import sys


def main():
    os.environ.setdefault(
        "DJANGO_SETTINGS_MODULE", "webxample.conf.settings"
    )

    from django.core.management import execute_from_command_line
    execute_from_command_line(sys.argv)

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

Unfortunately, a lot of frameworks (including Django) are not designed with the idea of packaging your projects that way in mind. It means that, depending on the advancement of your application, converting it to a package may require a lot of changes. In Django, this often means rewriting many of the implicit imports and updating a lot of configuration variables in your settings file.

The other problem here is consistency of releases created using Python packaging. If different team members are authorized to create application distribution, it is crucial that this process takes place in the same replicable environment. Especially when you do a lot of asset preprocessing, it is possible that the package created in two different environments will not look the same, even if it is created from the same code base. This may be due to different versions of tools used during the build process. The best practice is to move the distribution responsibility to some continuous integration/delivery system such as Jenkins, Buildbot, Travis CI, or similar. The additional advantage is that you can assert that the package passes all of the required tests before going to distribution. You can even make the automated deployment as a part of such a continuous delivery system.

Mind that although distributing your code as Python packages using `setuptools` might seem elegant, it is actually not simple and effortless. It has potential to greatly simplify your deployments and so it is definitely worth trying but it comes with the cost of increased complexity. If your preprocessing pipeline for your application grows too complex, you should definitely consider building Docker images and deploying your application as containers.

Deployment with Docker requires some additional setup and orchestration but in the long term saves a lot of time and resources that are otherwise required to maintain repeatable build environments and complex preprocessing pipelines.

27.4 4. Common conventions and practices

There are a set of common conventions and practices for deployment that not every developer may know but are obvious for anyone who did some operations in their life. As explained in this chapter's introduction, it is crucial to know at least a few of them, even if you are not responsible for code deployment and operations, because it will allow you to make better design decisions during the development.

27.4.1 4.1. The filesystem hierarchy

The most obvious conventions that may come into your mind are probably about filesystem hierarchy and user naming. If you are looking for such suggestions here, then you will be disappointed. There is, of course, a **Filesystem Hierarchy Standard (FHS)** that defines the directory structure and directory contents in Unix and Unix-like operating systems, but it is really hard to find the actual OS distribution that is fully compliant with FHS. If system designers and programmers cannot obey such standards, it is very hard to expect the same from its administrators. During my experience, I've seen application code deployed almost everywhere it is possible, including nonstandard custom directories in the root filesystem level. Almost always the people behind such decisions had really strong arguments for doing so. The only suggestions in this matter that I can give you are as follows:

- Choose wisely and avoid surprises.
- Be consistent across all of the available infrastructure of your project.
- Try to be consistent across your organization (the company you work in).

What really helps is to document conventions for your project. Just remember to make sure that this documentation is accessible for every interested team member and that everyone knows that such a document exists.

27.4.2 4.2. Isolation

Reasons for isolation as well as recommended tools were already discussed. These are: better environment reproducibility and solving the inevitable problems of dependency conflicts. For the purpose of deployments, there is only one important thing to add. You should always isolate project dependencies for each release of your application. In practice, it means that, whenever you deploy a new version of the application, you should create a new isolated environment for this release (using `virtualenv` or `venv`). Old environments should be left for some time on your hosts, so that, in case of issues, you can easily perform a rollback to one of the older versions of your application.

Creating fresh environments for each release helps in managing their clean state and compliance with a list of provided dependencies. By fresh environment we mean creating a new directory tree in the filesystem instead of updating already existing files. Unfortunately, it may make it a bit harder to perform things such as the graceful reload of services, which is much easier to achieve if the environment is updated in place.

27.4.3 4.3. Using process supervision tools

Applications on remote servers are never usually expected to quit. If it is a web application, its HTTP server process will indefinitely wait for new connections and requests and will exit only if some unrecoverable error occurs.

It is, of course, not possible to run it manually in shell and have a never-ending SSH connection. Using `nohup`, `screen`, or `tmux` to semi-daemonize the process is not an option. Doing so is like designing your service to fail.

What you need is to have some process supervision tool that can start and manage your application process. Before choosing the right one, you need to make sure it does the following things:

- Restarts the service if it quits
- Reliably tracks its state
- Captures its stdout / stderr streams for logging purposes
- Runs a process with specific user/group permissions
- Configures system environment variables

Most of the Unix and Linux distributions have some built-in tools/subsystems for process supervision such as `initd` scripts, `upstart`, and `runit`. Unfortunately, in most cases, they are not well suited for running user-level application code and are really hard to maintain. In particular, writing reliable `init.d` scripts is a real challenge because it requires a lot of Bash scripting that is hard to do it right. Some Linux distributions such as Gentoo have a redesigned approach to `init.d` scripts, so writing them is a lot easier. Anyway, locking yourself to a specific OS distribution just for the purpose of a single process supervision tool is not a good idea.

Two popular tools in the Python community for managing application processes are Supervisor (http://supervisord.org) and Circus (https://circus.readthedocs.org/en/latest/). They are both very similar in configuration and usage. Circus is a bit younger than Supervisor because it was created to address some weaknesses of the latter. They both can be configured in simple INI-like configuration format. They are not limited to running Python processes and can be configured to manage any application. It is hard to say which one is better because they both provide very similar functionality. Anyway, Supervisor does not run on Python 3, so it does not get our approval. While it is not a problem to run Python 3 processes under Supervisor's control, I will take it as an excuse and feature only the example of the Circus configuration.

Let's assume that we want to run the `webxample` application using `gunicorn` webserver under Circus control. In production, we would probably run Circus under an applicable system-level process supervision tool (`initd`, `upstart`, and `runit`), especially if it was installed from the system packages repository. For the sake of simplicity, we will run this locally inside of the virtual environment. The minimal configuration file (here named `circus.ini`) that allows us to run our application in Circus looks like this:

```
[watcher:webxample]
cmd = /path/to/venv/dir/bin/gunicorn webxample.conf.wsgi:application
numprocesses = 1
```

Now, the circus process can be run with this configuration file as the execution argument:

```
$ circusd circus.ini
2016-02-15 08:34:34 circus[1776] [INFO] Starting master on pid 1776
2016-02-15 08:34:34 circus[1776] [INFO] Arbiter now waiting for commands
2016-02-15 08:34:34 circus[1776] [INFO] webxample started
[2016-02-15 08:34:34 +0100] [1778] [INFO] Starting gunicorn 19.4.5
[2016-02-15 08:34:34 +0100] [1778] [INFO] Listening at: http://127.0.0.1:8000␣
↪(1778)
[2016-02-15 08:34:34 +0100] [1778] [INFO] Using worker: sync
[2016-02-15 08:34:34 +0100] [1781] [INFO] Booting worker with pid: 1781
```

Now, you can use the `circusctl` command to run an interactive session and control all managed processes using simple commands. Here is an example of such a session:

```
$ circusctl
circusctl 0.13.0
webxample: active
(circusctl) stop webxample
ok
(circusctl) status
webxample: stopped
(circusctl) start webxample
ok
(circusctl) status
webxample: active
```

Of course, both of the mentioned tools have a lot more features available. All of them are explained in their documentation, so before making your choice, you should read them carefully.

27.4.4 4.4. Application code running in user space

Your application code should be always run in user space. This means it must not be executed under super-user privileges. If you design your application following the Twelve- Factor App, it is possible to run your application under a user that has almost no privileges. The conventional name for the user that owns no files and is in no privileged groups is `nobody`; anyway, the actual recommendation is to create a separate user for each application daemon. The reason for that is system security. It is to limit the damage that a malicious user can do if it gains control over your application process. In Linux, processes of the same user can interact with each other, so it is important to have different applications separated at the user level.

27.4.5 4.5. Using reverse HTTP proxies

Multiple Python WSGI-compliant web servers can easily serve HTTP traffic all by themselves without the need of any other web server on top of them. It is still very common to hide them behind a reverse proxy such as NGINX or Apache. A reverse proxy creates an additional HTTP server layer that proxies requests and responses between clients and your application and appears to your Python server as though it is the requesting client. Reverse proxies are useful for the following variety of reasons:

- TLS/SSL termination is usually better handled by top-level web servers such as NGINX and Apache. This allows the Python application to speak only simple HTTP protocol (instead of HTTPS), so complexity and configuration of secure communication channels are left for the reverse proxy.
- Unprivileged users cannot bind low ports (in the range of 0-1000), but the HTTP protocol should be served to the users on port 80, and HTTPS should be served on port 443. To do this, you must run the process with super-user privileges. Usually, it is safer to have your application serving on a high port or on a Unix domain socket and use that as an upstream for reverse proxy that is run under the more privileged user.
- Usually, NGINX can serve static assets (images, JS, CSS, and other media) more efficiently than Python code. If you configure it as a reverse proxy, then it is only a few more lines of configuration to serve static files through it.
- When a single host needs to serve multiple applications from different domains, Apache or NGINX are indispensable for creating virtual hosts for different domains served on the same port.
- Reverse proxies can improve performance by adding additional caching layers or can be configured as simple load balancers. Reverse proxies can also apply compression (for example, gzip) to responses in order to limit the amount of required network bandwidth.

Some of the web servers actually are recommended to be run behind a proxy such as NGINX. For example, `gunicorn` is a very robust WSGI-based server that can give exceptional performance results if its clients are fast as well. On the other hand, it does not handle slow clients well, so it is easily susceptible to the denial of service attacks based on a slow client connection. Using a proxy server that is able to buffer slow clients is the best way to solve this problem.

Mind that, with proper infrastructure, it is possible to almost completely get rid of reverse proxies in your architecture. Nowadays, things such as SSL termination and compression can be easily handled with load balancing services such as AWS Load Balancer. Static and media assets are also better served through Content Delivery Networks (CDNs) that can also be used to cache other responses of your service.

The mentioned requirement to serve HTTP/HTTPS traffic on low 80/433 ports (that cannot be bound by unprivileged users) is also no longer a problem if the only entry points that your clients communicate with are your load balancers and CDN. Still, even with that kind of architecture, it does not necessarily mean that your system does not facilitate reverse proxies at all. For instance, many load balancers support proxy protocol. It means that a load balancer may appear to your application as though it is the requesting client. In such scenarios, the load balancer acts as it were in fact a reverse proxy.

27.4.6 4.6. Reloading processes gracefully

The ninth rule of Twelve-Factor App methodology deals with process disposability and says that you should maximize robustness with fast start up times and graceful shutdowns. While fast start up time is quite self-explanatory, the graceful shutdowns require some additional discussion.

In the scope of web applications, if you terminate the server process in a non-graceful way, it will quit immediately without the time to finish processing requests and reply with proper responses to connected clients. In the best scenario case, if you use some kind of reverse proxy, then the proxy might reply to the connected clients with some generic error response (for example, 502 Bad Gateway), even though it is not the right way to notify users that you have restarted your application and have deployed a new release.

According to the Twelve-Factor App, the web serving process should be able to quit gracefully upon receiving the Unix `SIGTERM` signal. This means the server should stop accepting new connections, finish processing all of the pending requests, and then quit with some exit code when there is nothing more to do.

Obviously, when all of the serving processes quit or start their shutdown procedure, you are not able to process new requests any longer. This means your service will still experience an outage. So there is an additional step you need to perform—start new workers that will be able to accept new connections while the old ones are gracefully quitting. Various Python WSGI-compliant web server implementations allow you to reload the service gracefully without any downtime.

The most popular Python web servers are Gunicorn and uWSGI, which provide the following functionality:

- Gunicorn's master process upon receiving the `SIGHUP` signal (`kill -HUP <process-pid>`) will start new workers (with new code and configuration) and attempt a graceful shutdown on the old ones.
- uWSGI has at least three independent schemes for doing graceful reloads. Each of them is too complex to explain briefly, but its official documentation provides full information on all of the possible options.

Today, graceful reloads are a standard in deploying web applications. Gunicorn seems to have an approach that is the easiest to use but also leaves you with the least flexibility. Graceful reloads in uWSGI on the other hand allow much better control on reloads but require more effort to automate and set up. Also, how you handle graceful reloads in your automated deploys is also affected by what supervision tools you use and how they are configured. For instance, in Gunicorn, graceful reloads are as simple as the following:

```
kill -HUP <gunicorn-master-process-pid>
```

But, if you want to properly isolate project distributions by separating virtual environments for each release and configure process supervision using symbolic links (as presented in the `fabfile` example earlier), you will shortly notice that this feature of Gunicorn may not work as expected. For more complex deployments, there is still no system-level solution available that will work for you out-of-the-box. You will always have to do a bit of hacking and sometimes this will require a substantial level of knowledge about low-level system implementation details.

In such complex scenarios, it is usually better to solve the problem on a higher level of abstraction. If you finally decide to run your applications as containers and distribute new releases as new container images (it is strongly advised), then you can leave the responsibility of graceful reloads to your container orchestration system of choice (for example, Kubernetes) that can usually handle various reloading strategies out-of-the-box.

Even without advanced container orchestration systems, you can do graceful reloading on the infrastructure level. For instance, AWS Elastic Load Balancer is able to gracefully switch traffic from your old application instances (for example, EC2 hosts) to new ones. Once old application instances receive no new traffic and are done handling their requests, they can be simply terminated without any observable outage to your service. Other cloud providers, of course, usually provide analogous features in their service portfolio.

27.5 5. Code instrumentation and monitoring

Our work does not end on writing an application and deploying it to target the execution environment. It is possible to write an application, which after deployment will not require any further maintenance, although it is very unlikely. In reality, we need to ensure that it is properly observed for errors and performance.

To be sure that your product works as expected, you need to properly handle application logs and monitor the necessary application metrics. This often includes the following:

- Monitoring web application access logs for various HTTP status codes
- A collection of process logs that may contain information about runtime errors and various warnings
- Monitoring usage of system resources (CPU load, memory, network traffic, I/O performance, disk usage, and so on) on the remote hosts where the application is run
- Monitoring application-level performance and metrics that are business performance indicators (customer acquisition, revenue, conversion rates, and so on)
- Luckily, there are a lot of free tools available for instrumenting your code and monitoring its performance. Most of them are very easy to integrate.

27.5.1 5.1. Logging errors – Sentry/Raven

The truth is painful. No matter how precisely your application is tested, your code will eventually fail at some point. This can be anything: unexpected exception, resource exhaustion, crash of some backing service, network outage, or simply an issue in the external library. Some of the possible issues (such as resource exhaustion) can be predicted and prevented in advance with proper monitoring. Unfortunately, there will always be something that passes your defenses, no matter how much you try.

What you can do instead is to prepare for such scenarios and make sure that no error passes unnoticed. In most cases, any unexpected failure scenario results in an exception raised by the application and logged through the logging system. This can be `stdout`, `stderr`, log file, or whatever output you have configured for logging. Depending on your implementation, this may or may not result in the application quitting with some system exit code.

You could, of course, depend solely on the log files stored in the filesystem for finding and monitoring your application errors. Unfortunately, observing errors in plain textual form is quite painful and does not scale well beyond anything more complex than running code in development. You will eventually be forced to use some services designed for log collection and analysis. Proper log processing is very important for other reasons (that will be explained a bit later) but does not work well for tracking and debugging errors. The reason is simple. The most common form of error logs is just Python stack trace. If you stop only on that, you will shortly realize that it is not enough in finding the root cause of your issues. This is especially true when errors occur in unknown patterns or in certain load conditions.

What you really need is as much context information about the error occurrence as possible. It is also very useful to have a full history of the errors that occurred in the production environment that you can browse and search in some convenient way.

One of the most common tools that gives such capabilities is Sentry (https://getsentry.com). It is a battle-tested service for tracking exceptions and collecting crash reports. It is available as open source, written in Python, and originated as a tool for backend web developers. Now, it outgrew its initial ambitions and has support for many more languages, including PHP, Ruby, and JavaScript but still stays the most popular tool of choice for many Python web developers.

Tip: It is common that web applications do not exit on unhandled exceptions because HTTP servers are obliged to return an error response with a status code from the 5XX group if any server error occurs. Most Python web frameworks do such things by default. In such cases, the exception is, in fact, handled either on the internal web framework level or by the WSGI server middleware. Anyway, this will usually still result in the exception stack trace being printed (usually on standard output).

The Sentry is available as a paid software-as-a-service model, but it is open source, so it can be hosted for free on your own infrastructure. The library that provides integration with Sentry is `sentry-sdk` (available on PyPI). If you haven't worked with it yet and want to test it but have no access to your own Sentry server, then you can easily sign up for a free trial on Sentry's on-premise service site. Once you have access to a Sentry server and have created a new project, you will obtain a string called Data Source Name (DSN). This DSN string is the minimal configuration setting needed to integrate your application with sentry. It contains protocol, credentials, server location, and your organization/project identifier in the following form:

```
'{PROTOCOL}://{PUBLIC_KEY}:{SECRET_KEY}@{HOST}/{PATH}{PROJECT_ID}'
```

Once you have DSN, the integration is pretty straightforward, as shown in the following code:

```
import sentry_sdk

sentry_sdk.init(dsn='https://<key>:<secret>@app.getsentry.com/<project>')

try:
    1 / 0
except Exception as e:
    sentry_sdk.capture_exception(e)
```

Important: The old library for Sentry integration is Raven. It is still maintained and available on PyPI but is being phased out, so it is best to start your Sentry integration using the newer `python-sdk` package. It is possible though that some framework integrations or Raven extensions haven't been ported to new SDK, so in such situations, integration using Raven is still a feasible integration path.

Sentry SDK has numerous integrations with most popular Python frameworks such as Django, Flask, Celery, or Pyramid to make integration easier. These integrations will automatically provide additional context that is specific to the given framework. If your web framework of choice does not have a dedicated support, the `sentry-sdk` package provides generic WSGI middleware that makes it compatible with any WSGI-based web servers, as shown in the following code:

```
from sentry_sdk.integrations.wsgi import SentryWsgiMiddleware


sentry_sdk.init(dsn='https://<key>:<secret>@app.getsentry.com/<project>')

# ...
# note: application is some WSGI application object defined earlier
application = SentryWsgiMiddleware(application)
```

The other notable integration is the ability to track messages logged through Python's built- in `logging` module. Enabling such support requires only the following few additional lines of code:

```
import logging
import sentry_sdk
from sentry_sdk.integrations.logging import LoggingIntegration

sentry_logging = LoggingIntegration(
    level=logging.INFO,
    event_level=logging.ERROR,
)
sentry_sdk.init(
    dsn='https://<key>:<secret>@app.getsentry.com/<project>',
    integrations=[sentry_logging],
)
```

Capturing of logging messages may have caveats, so make sure to read the official documentation on that topic if you are interested in such a feature. This should save you from unpleasant surprises.

The last note is about running your own Sentry as a way to save some money. There ain't no such thing as a free lunch. You will eventually pay additional infrastructure costs and Sentry will be just another service to maintain. Maintenance = additional work = costs! As your application grows, the number of exceptions grow, so you will be forced to scale Sentry as you scale your product. Fortunately, this is a very robust project, but will not give you any value if overwhelmed with too much load. Also, keeping Sentry prepared for a catastrophic failure scenario where thousands of crash reports per second can be sent is a real challenge. So you must decide which option is really cheaper for you, and whether you have enough resources to do all of this by yourself. There is, of course, no such dilemma if security policies in your organization deny sending any data to third parties. If so, just host it on your own infrastructure. There are costs, of course, but ones that are definitely worth paying.

27.5.2 5.2. Monitoring system and application metrics

When it comes to monitoring performance, the amount of tools to choose from may be overwhelming. If you have high expectations, then it is possible that you will need to use a few of them at the same time.

Munin (http://munin-monitoring.org) is one of the popular choices used by many organizations regardless of the technology stack they use. It is a great tool for analyzing resource trends and provides a lot of useful information, even with a default installation without additional configuration. Its installation consists of the following two main components:

- The Munin master that collects metrics from other nodes and serves metrics graphs
- The Munin node that is installed on a monitored host, which gathers local metrics and sends it to the Munin master

The master node and most of the plugins are written in Perl. There are also node implementations in other languages: munin-node-c is written in C (https://github.com/munin-monitoring/munin-c) and munin-node-python is written in Python (https://github.com/agroszer/munin-node-python). Munin comes with a huge number of plugins available in its `contrib` repository. This means it provides out-of-the- box support for most of the popular databases and system services. There are even plugins for monitoring popular Python web servers, such as uWSGI or Gunicorn. The main drawback of Munin is the fact that it serves graphs as static images and actual plotting configuration is included in specific plugin configurations. This does not help in creating flexible monitoring dashboards and comparing metric values from different sources at the same graph. But this is the price we need to pay for simple installation and versatility. Writing your own plugins is quite simple. There is the `munin-python` package (http://python-munin.readthedocs.org/en/latest/) that helps to write Munin plugins in Python.

Unfortunately, the architecture of Munin that assumes that there is always a separate monitoring daemon process on every host that is responsible for collection of metrics may not be the best solution for monitoring custom application performance metrics. It is indeed very easy to write your own Munin plugins, but under the assumption that the monitoring process can already report its performance statistics in some way.

If you want to collect some custom application-level metrics, it might be necessary to aggregate and store them in some temporary storage until reporting to a custom Munin plugin. It makes creation of custom metrics more complicated, so you might want to consider other solutions for such purposes.

The other popular solution that makes it especially easy to collect custom metrics is StatsD (https://github.com/etsy/statsd). It's a network daemon written in Node.js that listens to various statistics such as counters, timers, and gauges. It is very easy to integrate, thanks to the simple protocol based on UDP. It is also easy to use the Python package named `statsd` for sending metrics to the StatsD daemon, as follows:

Because UDP is a connectionless protocol, it has a very low performance overhead on the application code, so it is very suitable for tracking and measuring custom events inside the application code.

Unfortunately, StatsD is the only metrics collection daemon, so it does not provide any reporting features. You need other processes that are able to process data from StatsD in order to see the actual metrics graphs. The most popular choice is Graphite (http://graphite.readthedocs.org). It does mainly the following two things:

- Stores numeric time-series data
- Renders graphs of this data on demand

Graphite provides you with the ability to save graph presets that are highly customizable. You can also group many graphs into thematic dashboards. Graphs are, similar to Munin, rendered as static images, but there is also the JSON API that allows other frontends to read graph data and render it by other means.

One of the great dashboard plugins integrated with Graphite is Grafana (http://grafana.org). It is really worth trying because it has way better usability than plain Graphite dashboards. Graphs provided in Grafana are fully interactive and easier to manage.

Graphite is unfortunately a bit of a complex project. It is not a monolithic service and consists of the following three separate components:

- **Carbon**: This is a daemon written using Twisted that listens for time-series data.
- **whisper**: This is a simple database library for storing time-series data.

- **graphite webapp**: This is a Django web application that renders graphs on- demand as static images (using Cairo library) or as JSON data.

When used with the StatsD project, the `statsd` daemon sends its data to the `carbon` daemon. This makes the full solution a rather complex stack of various applications, where each of them is written using completely different technology. Also, there are no preconfigured graphs, plugins, and dashboards available, so you will need to configure everything by yourself. This is a lot of work at the beginning and it is very easy to miss something important. This is the reason why it might be a good idea to use Munin as a monitoring backup, even if you decide to have Graphite as your core monitoring service.

Another good monitoring solution for arbitrary metric collection is Prometheus. It has a completely different architecture than Munin and StatsD. Instead of relying on monitored applications or daemons to push metrics in configured intervals, Prometheus actively pulls metrics directly from the source using the HTTP protocol. This requires monitored services to store (and sometimes preprocess) metrics internally and expose them on HTTP endpoints.

Fortunately, Prometheus comes with a handful of libraries for various languages and frameworks to make this kind of integration as easy as possible. There are also various exporters that act as bridges between Prometheus and other monitoring systems. So, if you already use other monitoring solutions, it is usually very easy to migrate gradually to a Prometheus architecture. Prometheus also wonderfully integrates with Grafana.

27.5.3 5.3. Dealing with application logs

While solutions such as Sentry are usually way more powerful than ordinary textual output stored in files, logs will never die. Writing some information to a standard output or file is one of the simplest things that an application can do and this should never be underestimated. There is a risk that messages sent to Sentry by Raven will not get delivered. The network can fail. Sentry's storage can get exhausted or may not be able to handle the incoming load. Your application might crash before any message is sent (with a segmentation fault, for example). These are only a few of the possible scenarios.

What is less likely is that your application won't be able to log messages that are going to be written to the filesystem. It is still possible, but let's be honest, if you face such a condition where logging fails, probably you have a lot more burning issues than some missing log messages.

Remember that logs are not only about errors. Many developers used to think about logs only as a source of data that is useful when debugging issues and/or that can be used to perform some kind of forensics.

Definitely, less of them try to use it as a source for generating application metrics or to do some statistical analysis. But logs may be a lot more useful than that. They can even be a core of the product implementation. A great example of building a product with logs is Amazon's article presenting example architecture for the real-time bidding service, where everything is centered around access log collection and processing. See https://aws.amazon.com/blogs/aws/real-time-ad-impression-bids-using-dynamodb/

5.3.1. Basic low-level log practices

The Twelve-Factor App manifesto says that logs should be treated as event streams. So, the log file is not a log by itself, but only an output format. The fact that they are streams means they represent time ordered events. In raw, they are typically in a plaintext format with one line per event, although in some cases they may span across multiple lines (this is typical for any back traces related to runtime errors).

According to the Twelve-Factor App methodology, the application should never be aware of the format in which logs are stored. This means that writing to the file, or log rotation and retention should never be maintained by the application code.

These are the responsibilities of the environment in which the applications is run. This may be confusing because a lot of frameworks provide functions and classes for managing log files as well as rotation, compression, and retention utilities. It is tempting to use them because everything can be contained in your application code base, but actually it is an anti-pattern that should be avoided.

The best practices for dealing with logs are as follows:

- The application should always write logs unbuffered to the standard output (`stdout`).

- The execution environment should be responsible for collection and routing of logs to the final destination.

The main part of the mentioned execution environment is usually some kind of process supervision tool. The popular Python solutions, such as Supervisor or Circus, are the first ones responsible for dealing with log collection and routing. If logs are to be stored in the local filesystem, then only they should write to actual log files.

Both Supervisor and Circus are also capable of handling log rotation and retention for managed processes but you should really consider whether this is a path that you want to take. Successful operations are mostly about simplicity and consistency. Logs of your own application are probably not the only ones that you want to process and archive. If you use Apache or NGINX as a reverse proxy, you might want to collect their access logs.

You might also want to store and process logs for caches and databases. If you are running some popular Linux distribution, then the chances are very high that each of these services have their own log files processed (rotated, compressed, and so on) by the popular utility named `logrotate`. My strong recommendation is to forget about Supervisor's and Circus' log rotation capabilities for the sake of consistency with other system services. `logrotate` is way more configurable and also supports compression.

Tip: There is an important thing to know when using `logrotate` with Supervisor or Circus. Rotation of logs will always happen while process Supervisor still has open descriptor to rotated logs. If you don't take proper countermeasures, then new events will be still written to the file descriptor that was already deleted by `logrotate`. As a result, nothing more will be stored in a filesystem. Solutions to this problem are quite simple. Configure `logrotate` for log files of processes managed by Supervisor or Circus with the `copytruncate` option. Instead of moving the log file after rotation, it will copy it and truncate the original file to zero size in place. This approach does not invalidate any of the existing file descriptors and processes that are already running can write to log files uninterrupted. Supervisor can also accept the `SIGUSR2` signal that will make it reopen all of the file descriptors. It may be included as the `postrotate ``script in the ``logrotate` configuration. This second approach is more economical in the terms of I/O operations, but is also less reliable and harder to maintain.

5.3.2. Tools for log processing

If you have no experience in working with big amounts of logs, you will eventually gain it when working with a product that has some substantial load. You will shortly notice that a simple approach based on storing them in files and backing them up in some persistent storage for later retrieval is not enough. Without proper tools, this will become crude and expensive. Simple utilities such as `logrotate` help you only to ensure that the hard disk is not overloaded by the ever-increasing amount of new events, although splitting and compressing log files only helps in the data archival process but does not make data retrieval or analysis simpler.

When working with distributed systems that span across multiple nodes, it is nice to have a single central point from which all logs can be retrieved and analyzed. This requires a log processing flow that goes way beyond simple compression and backing up. Fortunately, this is a well-known problem so there are many tools available that aim to solve it.

One of the popular choices among many developers is **Logstash**. This is the log collection daemon that can observe active log files, parse log entries, and send them to the backing service in a structured form. The choice of backing stays almost always the same: **Elasticsearch**. Elasticsearch is the search engine built on top of Lucene. Among text search capabilities, it has a unique data aggregation framework that fits extremely well into the purpose of log analysis. The other addition to this pair of tools is **Kibana**. It is a very versatile monitoring, analysis, and visualization platform for Elasticsearch. The way that these three tools complement each other is the reason why almost always they are used together as a single stack for log processing.

The integration of existing services with Logstash is very simple because it can listen on existing log file changes for the new events with only minimal changes in your logging configuration. It parses logs in textual form and has preconfigured support for some of the popular log formats, such as Apache/NGINX access logs. Logstash can be complemented with Beats. Beats are log shippers compatible with Logstash input protocols that can collect not only raw log data from files (Filebeat) but also various system metrics (Metricbeat) and even audit user activities on hosts (Auditbeat).

The other solution that seems to fill some of Logstash gaps is Fluentd. It is an alternative log collection daemon

that can be used interchangeably with Logstash in the mentioned log monitoring stack. It also has an option to listen and parse log events directly in log files, so integration requires only a little effort. In contrast to Logstash, it handles reloads very well and even does not need to be signaled if log files were rotated. Anyway, the most advantage comes from using one of its alternative log collection options that will require some substantial changes to logging configuration in your application.

Fluentd really treats logs as event streams (as recommended by the Twelve-Factor App). The file-based integration is still possible but it is only kind of backward compatible for legacy applications that treat logs mainly as files. Every log entry is an event and it should be structured. Fluentd can parse textual logs and has multiple plugin options to handle, including the following:

- Common formats (Apache, NGINX, and syslog)
- Arbitrary formats specified using regular expressions or handled with custom parsing plugins
- Generic formats for structured messages such as JSON

The best event format for Fluentd is JSON because it adds the least amount of overhead. Messages in JSON can also be passed almost without any change to the backing service such as Elasticsearch or the database.

The other very useful feature of Fluentd is the ability to pass event streams using transports other than a log file written to the disk. The following are the most notable built-in input plugins:

- `in_udp`: With this plugin, every log event is sent as UDP packets.
- `in_tcp`: With this plugin, events are sent through TCP connection.
- `in_unix`: With this plugin, events are sent through a Unix domain socket (named socket).
- `in_http`: With this plugin, events are sent as HTTP POST requests.
- `in_exec`: With this plugin, Fluentd process executes an external command periodically to pull events in the JSON or MessagePack format.
- `in_tail`: With this plugin, Fluentd process listens for an event in a textual file.

Alternative transports for log events may be especially useful in situations where you need to deal with poor I/O performance of machine storage. It is very often on cloud computing services that the default disk storage has a very low number of **Input Output Operations Per Second (IOPS)** and you need to pay a lot of money for better disk performance.

If your application outputs a large amount of log messages, you can easily saturate your I/O capabilities, even if the data size is not very high. With alternate transports, you can use your hardware more efficiently because you leave the responsibility of data buffering only to a single process-log collector. When configured to buffer messages in memory instead of disk, you can even completely get rid of disk writes for logs, although this may greatly reduce the consistency guarantees of collected logs.

Using different transports seems to be slightly against the 11 th rule of the Twelve-Factor App methodology. Treating logs as event streams when explained in detail suggests that the application should always log only through a single standard output stream (`stdout`). It is still possible to use alternate transports without breaking this rule. Writing to `stdout` does not necessarily mean that this stream must be written to file.

You can leave your application logging that way and wrap it with an external process that will capture this stream and pass it directly to Logstash or Fluentd without engaging the filesystem. This is an advanced pattern that may not be suitable for every project. It has the obvious disadvantage of higher complexity, so you need to consider for yourself whether it is really worth doing.

Part VIII

References

- https://www.packtpub.com/application-development/clean-code-python
- https://www.packtpub.com/application-development/expert-python-programming-third-edition
- https://www.packtpub.com/application-development/mastering-python-design-patterns-second-edition
- https://www.packtpub.com/application-development/hands-data-structures-and-algorithms-python-second-edition