ROBERT SMALLSHIRE & AUSTIN BINGHAM

THE PYTHON MASTER

SixtyNORTH

The Python Master

Robert Smallshire and Austin Bingham

This book is for sale at http://leanpub.com/python-master

This version was published on 2019-01-01

Leanpub

* * * * *

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

* * * * *

ISBN for EPUB version: 978-82-93483-08-3

ISBN for MOBI version: 978-82-93483-09-0

Table of Contents

Preface

In 2013, when we incorporated our Norway-based software consultancy and training business *Sixty North*, we were courted by *Pluralsight*, a publisher of online video training material, to produce Python training videos for the rapidly growing MOOC market. At the time, we had no experience of producing video training material, but we were sure we wanted to carefully structure our introductory Python content to respect certain constraints. For example, we wanted an absolute minimum of forward references since those would be very inconvenient for our viewers. We're both men of words who live by Turing Award winner Leslie Lamport's maxim *"If you're thinking without writing you only think you're thinking"*, so it was natural for us to attack video course production by first writing a script.

In the intervening years we worked on three courses with *Pluralsight*: Python Fundamentals, Python – Beyond the Basics, and Advanced Python. These three online courses have been transformed into three books The Python Apprentice, The Python Journeyman, and this one, The Python Master.

You can read *The Python Master* either as a standalone Python tutorial, or as the companion volume to the corresponding *Advanced Python* video course, depending on which style of learning suits you best. In either case we assume that you're up to speed with the material covered in the preceding books or courses.

Errata and Suggestions

All the material in this book has been thoroughly reviewed and tested; nevertheless, it's inevitable that some mistakes have crept in. If you do spot a mistake, we'd really appreciate it if you'd let us know via the *Leanpub* Python Master Discussion page so we can make amends and deploy a new version.

Conventions Used in This Book

Code examples in this book are shown in a fixed-width text which is colored with syntax highlighting:

```
>>> def square(x):
...     return x * x
...
```

Some of our examples show code saved in files, and others — such as the one above — are from interactive Python sessions. In such interactive cases, we include the prompts from the Python session such as the triple-arrow (>>>) and triple-dot (`...`) prompts. You don't need to type these arrows or dots. Similarly, for operating system shell-commands we will use a dollar prompt (`$`) for Linux, macOS and other Unixes, or where the particular operating system is unimportant for the task at hand:

```
$ python3 words.py
```

In this case, you don't need to type the `$` character.

For Windows-specific commands we will use a leading greater-than prompt:

```
> python words.py
```

Again, there's no need to type the > character.

For code blocks which need to be placed in a file, rather than entered interactively, we show code without any leading prompts:

```
def write_sequence(filename, num):
    """Write Recaman's sequence to a text file."""
    with open(filename, mode='wt', encoding='utf-8') as f:
        f.writelines("{0}\n".format(r)
                     for r in islice(sequence(), num + 1))
```

We've worked hard to make sure that our lines of code are short enough so that each single logical line of code corresponds to a single physical line of code in your book. However, the vagaries of publishing e-books to different devices and the very genuine need for occasional long lines of code mean we can't guarantee that lines don't wrap. What we can guarantee, however, is that where a line does wrap, the publisher has inserted a backslash character `\` in the final column. You need to use your judgement to determine whether this character is legitimate part of the code or has been added by the e-book platform.

```
>>> print("This is a single line of code which is very long. Too long, in fact, to fit
 a single physical line of code in the book.")
```

If you see a backslash at the end of the line within the above quoted string, it is *not* part of the code, and should not be entered.

Occasionally, we'll number lines of code so we can refer to them easily from the narrative next. These line numbers should not be entered as part of the code. Numbered code blocks look like this:

```
def write_grayscale(filename, pixels):
    height = len(pixels)
    width = len(pixels[0])

    with open(filename, 'wb') as bmp:
        # BMP Header
        bmp.write(b'BM')

        # The next four bytes hold the filesize as a 32-bit
        # little-endian integer. Zero placeholder for now.
        size_bookmark = bmp.tell()
        bmp.write(b'\x00\x00\x00\x00')
```

Sometimes we need to present code snippets which are incomplete. Usually this is for brevity where we are adding code to an existing block, and where we want to be clear about the block structure without repeating all existing contents of the block. In such cases we use a Python comment containing three dots `# ...` to indicate the elided code:

```
class Flight:

    # ...

    def make_boarding_cards(self, card_printer):
        for passenger, seat in sorted(self._passenger_seats()):
            card_printer(passenger, seat, self.number(), self.aircraft_model())
```

Here it is implied that some other code already exists within the `Flight` class block before the `make_boarding_cards()` function.

Finally, within the text of the book, when we are referring to an identifier which is also a function we will use the identifier with empty parentheses, just as we did with `make_boarding_cards()` in the preceding paragraph.

Welcome!

Welcome to *The Python Master*. This the third in Sixty North's trilogy of books which cover the core Python language, and it builds directly on the knowledge we impart in the first two books, *The Python Apprentice* and *The Python Journeyman*.

Our books follow a thoughtfully designed spiral curriculum. We visit the same, or closely related, topics several times in increasing depth, sometimes multiple times in the same book. For example, in *The Python Apprentice* we cover single class inheritance. Then in *The Python Journeyman* we cover multiple class inheritance. In this book we'll cover metaclasses to give you the ultimate power over class construction.

The Python Master covers some aspects of Python which you may use relatively infrequently. Mastery of the Python language calls for felicity with these features — at least to the extent that you can identify their use in existing code and appreciate their intent. We'll go all the way in this book to show you how to use the most powerful of Python's language features to greatest effect…with occasional reminders that sometimes there is a simpler way. Knowing how to use advanced features is what we will teach in this book. Knowing *when* to use advanced features — demonstrating a level of skilfulness that can only be achieved through time and experience — is the mark of the true master.

Specifically, we'll look at:

- **Advanced flow control**: loop-else clauses, try..else and switch emulation
- **Byte-oriented programming**: interpreting and manipulating data at the lowest level in Python
- **Object internals**: how objects are represented under the hood in Python
- **Descriptors**: gaining complete control over attribute access using a crucial mechanism in Python which is usually behind the scenes

- **Custom object allocation**: making the most efficient use of memory with tactics like interning of immutable objects
- **Metaclasses**: understanding the process by which class declarations are transformed into class objects and writing your own metaclasses to customise class construction
- **Class decorators**: these can be a simpler, although less powerful, alternative to metaclasses in many cases
- **Abstract base classes**: specifying and detecting class interface protocols, including how you can make built-in types become subclasses of your own types

That’s a lot to cover, but over the course of this book you’ll begin to see how many of these pieces fit together.

Prerequisites

A functioning Python 3.5+ runtime

First and foremost, you will need access to a working Python 3.5+[1] system. Any version from 3.5 onward will suffice, and we have tried to avoid any dependencies on minor versions. With that said, more recent Python 3 versions have lots of exciting new features and standard library functionality, so if you have a choice you should probably get the most recent stable version.

At a minimum, you need to be able to run a Python 3 REPL. You can, of course, use an IDE if you wish, but we won't require anything beyond what comes with the standard Python distribution.

Experience with concepts from the previous books

In this book we'll assume that you have knowledge of — and ideally first-hand experience using — the concepts and techniques covered in *The Python Apprentice* and *The Python Journeyman*. The material in this book will be accessible to anyone who's read the previous books, but it may not be very meaningful without associated practical application of that knowledge. In other words, this book is really intended for people who have worked on a few Python projects, developers seasoned enough to appreciate *how* and *why* some of these advanced topics might be applied.

If you don't have much Python experience yet, by all means read on. But be aware that some concepts (*e.g.* metaclasses) often don't make complete sense until you've personally encountered a situation where they could be helpful. Take what you can from this book for now, continue programming, and refer back to here when you encounter such a situation.

The Road Goes On Forever

> When Alexander saw the breadth of his domain, he wept for there were no more worlds to conquer.
>
> – *Plutarch (popular misquote)*

This book completes our trilogy on core Python, and after reading it you will know a great deal about the language. Unlike poor Alexander the Great, though, you can save your tears: there is still a lot to Python that we haven't covered, so you can keep learning for years to come!

The material in *The Python Master* can be tricky, and much of it, by its nature, requires experience to use well. So take your time, and try to correlate what you learn with your past and ongoing Python programming experience. And once you think you understand an idea, take the ultimate test: *teach it to others*. Give conference talks, write blog posts (or books!), speak at user groups, or just show what you've learned to a colleague.

If writing these books has taught us one thing, it's that explaining things to others is the best way to find out what you don't understand. This book can help you on the way to Python mastery, but going the full distance is up to you. Enjoy the journey!

Chapter 1 - Advanced Flow Control

In this chapter we'll be looking at some more advanced, and possibly even obscure, flow-control techniques used in Python programs with which you should be familiar at the advanced level.

Specifically, we'll cover:

- `else` clauses on loops
- `else` clauses on `try` blocks
- emulating switch statements
- dispatching on type

By understanding these unusual language features you'll be able to understand more code that you may encounter. You'll also be able to reduce the complexity of your own code by eliminating unnecessary conditionals.

`else` clauses on loops

You doubtless associate the `else` keyword with optional clauses complementary to the conditional clause introduced by the `if` statement. But did you know that `else` can also be used to associate optional code blocks with loops? That sounds pretty weird, and to be honest it *is* an unusual language feature. That's why it's only rarely seen in the wild and most definitely deserves some explanation.

`while` .. `else`

We'll start out by saying that Guido van Rossum, inventor and benevolent dictator for life of Python, has admitted that he would not include this feature if he developed Python again.[23] Back in our timeline though, the feature is there, and we need to understand it.

Apparently, the original motivation for using the `else` keyword this way in conjunction with `while` loops comes from Donald Knuth in early efforts to rid structured programming languages of goto. Although the choice of keyword is at first perplexing, it’s possible to rationalise the use of `else` in this way through comparison with the `if...else` construct:

```
if condition:
    execute_condition_is_true()
else:
    execute_condition_is_false()
```

In the `if...else` structure, the code in the `if` clause is executed if the condition evaluates to `True` when converted to `bool` — in other words, if the condition is ‘truthy’. On the other hand, if the condition is ‘falsey’, the code in the `else` clause is executed instead.

Now look at the `while..else` construct:

```
while condition:
    execute_condition_is_true()
else:
    execute_condition_is_false()
```

We can, perhaps, glimpse the logic behind choosing the `else` keyword. The `else` clause will be executed when, and only when, the condition evaluates to `False`. The condition may already be `False` when execution first reaches the `while` statement, so it may branch immediately into the `else` clause. Or there may be any number of cycles of the `while`-loop before the condition becomes `False` and execution transfers to the `else` block.

Fair enough, you say, but isn’t this equivalent to putting the `else` code after the loop, rather than in a special `else` block, like this? :

```
while condition:
    execute_condition_is_true()
execute_condition_is_false()
```

You would be right in this simple case. But if we place a `break` statement within the loop body it becomes possible to exit the loop without the loop conditional ever becoming false. In that case the `execute_condition_is_false()` call happens even though condition *is not* `False`:

```
while condition:
    flag = execute_condition_is_true()
```

```
    if flag:
        break
execute_condition_is_false()
```

To fix this, we could use a second test with an `if` statement to provide the desired behaviour:

```
while condition:
  flag = execute_condition_is_true()
  if flag:
      break

if not condition:
  execute_condition_is_false()
```

The drawback with this approach is that the test is duplicated, which violates the Don’t Repeat Yourself — or DRY — guideline, which hampers maintainability.

The `while..else` construct in Python allows us to avoid this second redundant test:

```
while condition:
    flag = execute_condition_is_true()
    if flag:
        break
else:
    execute_condition_is_false()
```

Now, the `else` block *only* executes when the main loop condition evaluates to `False`. If we jump out of the loop another way, such as with the `break` statement, execution jumps over the `else` clause. There’s no doubt that, however you rationalise it, the choice of keyword here is confusing, and it would have been better all round if a `nobreak` keyword had been used to introduce this block. In lieu of such a keyword, we heartily recommend that, if you *are* tempted to use this obscure and little-used language feature, you include such a `nobreak` comment like this:

```
while condition:
    flag = execute_condition_is_true()
    if flag:
        break
else:  # nobreak
    execute_condition_is_false()
```

So much for the theory; is this any use in practice?

We must admit that neither of the authors of this book have used `while..else` in practice. Almost every example we've seen could be implemented better by another, more easily understood, construct, which we'll look at later. That said, let's look at an example in `evaluator.py`:

```
def is_comment(item):
    return isinstance(item, str) and item.startswith('#')


def execute(program):
    """Execute a stack program.

    Args:
        program: Any stack-like collection where each item in the stack
            is a callable operator or non-callable operand. The top-most
            items on the stack may be strings beginning with '#' for
            the purposes of documentation.  Stack-like means support for:

              item = stack.pop()  # Remove and return the top item
              stack.append(item)  # Push an item to the top
              if stack:           # False in a boolean context when empty
    """
    # Find the start of the 'program' by skipping
    # any item which is a comment.
    while program:
        item = program.pop()
        if not is_comment(item):
            program.append(item)
            break
    else:  # nobreak
        print("Empty program!")
        return

    # Evaluate the program
    pending = []
    while program:
        item = program.pop()
        if callable(item):
            try:
                result = item(*pending)
            except Exception as e:
                print("Error: ", e)
                break
            program.append(result)
            pending.clear()
        else:
            pending.append(item)
    else:  # nobreak
        print("Program successful.")
        print("Result: ", pending)

    print("Finished")


if __name__ == '__main__':
    import operator

    program = list(reversed((
```

```
        "# A short stack program to add",
        "# and multiply some constants",
        9,
        13,
        operator.mul,
        2,
        operator.add)))

    execute(program)
```

This code evaluates simple ‘stack programs’. Such programs are specified as a stack of items where each item is either a callable function (for these we use any regular Python function) or an argument to that function. So to evaluate `5 + 2`, we would set up the stack like this:

```
5
2
+
```

When the plus operator is evaluated, its result is pushed onto the stack. This allows us to perform more complex operations such as `(5 + 2) * 3`:

```
5
2
+
3
*
```

As the stack contains the expression in reverse Polish notation, the parentheses we needed in the infix version aren’t required. In reality, the stack will be a Python list. The operators will be callables from the Python standard library `operators` module, which provides named-function equivalents of every Python infix operator. Finally, when we use Python lists as stacks the top of the stack is at the end of the list, so to get everything in the right order we’ll need to reverse our list:

```
program = list(reversed([5, 2, operator.add, 3, operator.mul]))
```

For added interest, our little stack language also supports comments as strings beginning with a hash symbol, just like Python. However, such comments are only allowed at the beginning of the program, which is at the top of the stack:

```
import operator

program = list(reversed((
          "# A short stack program to add",
          "# and multiply some constants",
```

```
        5,
        2,
        operator.add,
        3,
        operator.mul)))
```

We’d like to run our little stack program by passing it to a function `execute()`, like this:

```
execute(program)
```

Let’s see what such a function might look like and how it can use the `while..else` construct to good effect. The first thing our `execute()` function needs to do is pop all the comment strings from the top of the stack and discard them. To help with this, we’ll define a simple predicate which identifies stack items as comments:

```
def is_comment(item):
    return isinstance(item, str) and item.startswith('#')
```

> Notice that this function relies on an important Python feature called *boolean short-circuiting*. If `item` is not a string then the call to `startswith()` raises an `AttributeError`. However, when evaluating the boolean operators `and` and `or` Python will only evaluate the second operand if it is necessary for computing the result. When `item` is *not* as string (meaning the first operand evaluates to `False`) then the result of the boolean `and` must also be `False`; in this case there’s no need to evaluate the second operand.

Given this useful predicate, we’ll now use a while-loop to clear comment items from the top of the stack:

```
while program:
    item = program.pop()
    if not is_comment(item):
        program.append(item)
        break
else:  # nobreak
    print("Empty program!")
    return
```

The conditional expression for the `while` statement is the `program` stack object itself. Remember that using a collection in a boolean context like this evaluates to `True` if the collection is non-empty or `False` if it is empty. Or put another way, empty collections are ‘falsey’. So this statement reads as “While there are items remaining in the program.”

The while-loop has an associated `else` clause where execution will jump if the `while` condition should ever evaluate to `False`. This happens when there are no more items remaining in the program. In this clause we print a warning that the program was found to be logically empty, then returning early from the `execute()` function.

Within the `while` block, we `pop()` an item from the stack — recall that regular Python lists have this method which removes and returns the last item from a list. We use logical negation of our `is_comment()` predicate to determine if the just-popped item is *not* a comment. If the loop has reached a non-comment item, we push it back onto the stack using a call to `append()`, which leaves the stack with the first non-comment item on top, and then `break` from the loop. Remember that the while-loop `else` clause is best thought of as the “no break” clause, so when we break from the loop execution skips the `else` block and proceeds with the first statement after.

This loop executes the `else` block in the case of search failure — in this example if we fail to locate the first non-comment item because there isn’t one. Search failure handling is probably the most widespread use of loop `else` clauses.

Now we know that all remaining items on the stack comprise the actual program. We’ll use another while-loop to evaluate it:

```
pending = []
while program:
    item = program.pop()
    if callable(item):
        try:
            result = item(*pending)
        except Exception as e:
            print("Error: ", e)
            break
        program.append(result)
        pending.clear()
    else:
        pending.append(item)
else:  # nobreak
    print("Program successful.")
    print("Result: ", pending)
```

Before this loop we set up an empty list called `pending`. This will be used to accumulate arguments to functions in the stack, which we'll look at shortly.

As before, the condition on the while-loop is the `program` stack itself, so this loop will complete, and control will be transferred to the while-loop else-clause, when the program stack is empty. This will happen when program execution is complete.

Within the while-loop we pop the top item from the stack and inspect it with the built-in `callable()` predicate to decide if it is a function. For clarity, we'll look at the `else` clause first. That's the `else` clause associated with the `if`, not the `else` clause associated with the `while`!

If the popped item is *not* callable, we append it to the `pending` list, and go around the loop again if the `program` is not yet empty.

If the item *is* callable, we try to call it, passing any pending arguments to the function using the star-args extended call syntax. Should the function call fail, we catch the exception, print an error message, and `break` from the while-loop. Remember this will bypass the loop `else` clause. Should the function call succeed, we assign the return value to `result`, push this value back onto the program stack, and clear the list of pending arguments.

When the program stack is empty the `else` block associated with the while-loop is entered. This prints "Program successful" followed by any contents of the `pending` list. This way a program can "return" a result by leaving non-callable values at the bottom of the stack; these will be swept up into the `pending` list and displayed at the end.

for-else loops

Now we understand the `while..else` construct we can look at the analogous `for..else` construct. The `for..else` construct may seem even more odd than `while..else`, given the absence of an explicit condition in the `for` statement, but you need to remember that the `else` clause is really the no-break clause. In the case of the for-loop, that's exactly when it is called — when the loop is exited without breaking. This includes the case when the iterable series over which the loop is iterating is empty.[4]

```
for item in iterable:
    if match(item):
        result = item
        break
else:  # nobreak
    # No match found
    result = None

# Always come here
print(result)
```

The typical pattern of use is like this: We use a for-loop to examine each item of an iterable series, and test each item. If the item matches, we `break` from the loop. If we fail to find a match the code in the `else` block is executed, which handles the ‘no match found’ case.

For example, here is a code fragment which ensures that a list of integers contains at least one integer divisible by a specified value. If the supplied list does not contain a multiple of the divisor, the divisor itself is appended to the list to establish the invariant:

```
items = [2, 25, 9, 37, 28, 14]
divisor = 12

for item in items:
    if item % divisor == 0:
        found = item
        break
else:  # nobreak
    items.append(divisor)
    found = divisor

print("{items} contains {found} which is a multiple of {divisor}"
      .format(**locals()))
```

We set up a list of numeric items and a divisor, which will be 12 in this case. Our for-loop iterates through the items, testing each in turn for divisibility by the divisor. If a multiple of the divisor is located, the variable `found` is set to the current item, and we break from the loop — skipping over the loop-`else` clause — and print the list of items. Should the for-loop complete without encountering a multiple of 12, the loop-else clause will be entered, which appends the divisor itself to the list, thereby ensuring that the list contains an item divisible by the divisor.

For-else clauses are more common than while-else clauses, although we must emphasise that neither are common, and both are widely misunderstood. So although we want *you* to understand them, we can’t really recommend using them unless you're sure that everyone who needs to read your code is familiar with their use.

In a survey undertaken at PyCon 2011, a majority of those interviewed could not properly understand code which used loop-else clauses. Proceed with caution!

An alternative to loop `else` clauses

Having pointed out that loop `else` clauses are best avoided, it's only fair that we provide you with an alternative technique, which we think is better for several reasons, beyond avoiding an obscure Python construct.

Almost any time you see a loop `else` clause you can refactor it by extracting the loop into a named function, and instead of `break`-ing from the loop, prefer to return directly from the function. The search failure part of the code, which was in the `else` clause, can then be dedented a level and placed after the loop body. Doing so, our new `ensure_has_divisible()` function would look like this:

```
def ensure_has_divisible(items, divisor):
    for item in items:
        if item % divisor == 0:
            return item
    items.append(divisor)
    return divisor
```

which is simple enough to be understood by any Python programmer. We can use it, like this:

```
items = [2, 25, 9, 37, 24, 28, 14]
divisor = 12

dividend = ensure_has_divisible(items, divisor)

print("{items} contains {dividend} which is a multiple of {divisor}"
      .format(**locals()))
```

This is easier to understand, because it doesn't use any obscure and advanced Python flow-control techniques. It's easier to test because it is extracted into a standalone function. It's reusable because it's not mixed in with other code, and we can give it a useful and meaningful name, rather than having to put a comment in our code to explain the block.

The try..except..else construct

The third slightly oddball place we can use else blocks is as part of the `try..except` exception handling structure. In this case, the `else` clause is executed if **no** exception is raised:

```
try:
    # This code might raise an exception
    do_something()
except ValueError:
    # ValueError caught and handled
    handle_value_error()
else:
    # No exception was raised
    # We know that do_something() succeeded, so
    do_something_else()
```

Looking at this, you might wonder why we don’t call `do_something_else()` on the line after `do_something()`, like this:

```
try:
    # This code might raise an exception
    do_something()
    do_something_else()
except ValueError:
    # ValueError caught and handled
    handle_value_error()
```

The downside of this approach, is that we now have no way of telling in the `except` block whether it was `do_something()` or `do_something_else()` which raised the exception. The enlarged scope of the try block also obscures our intent with catching the exception; where are we expecting the exception to come from?

Although rarely seen in the wild, it is useful, particularly when you have a series of operations which may raise the same exception type, but where you only want to handle exceptions from the first such, operation, as commonly happens when working with files:

```
try:
    f = open(filename, 'r')
except OSError:  # OSError replaces IOError from Python 3.3 onwards
    print("File could not be opened for read")
else:
    # Now we're sure the file is open
    print("Number of lines", sum(1 for line in f))
    f.close()
```

In this example, both opening the file and iterating over the file can raise an `OSError`, but we're only interested in handling the exception from the call to `open()`.

It's possible to have both an `else` clause and a `finally` clause. The `else` block will only be executed if there was no exception, whereas the `finally` clause will *always* be executed.

Emulating switch

Most imperative programming languages include a switch or case statement which implements a multiway branch based on the value of an expression. Here's an example for the C programming language, where different functions are called depending on the value of a `menu_option` variable. There's also handing for the case of 'no such option':

```
switch (menu_option) {
    case 1:  single_player();    break;
    case 2:  multi_player();     break;
    case 3:  load_game();        break;
    case 4:  save_game();        break;
    case 5:  reset_high_score(); break;
    default:
        printf("No such option!");
        break;
}
```

Although switch can be emulated in Python by a chain of `if..elif..else` blocks, this can be tedious to write, and it's error prone because the condition must be repeated multiple times.

An alternative in Python is to use a mapping of callables. Depending on what you want to achieve, these callables may be lambdas or named functions.

We'll look at a simple adventure game you cannot win in `kafka.py`, which we'll refactor from using `if..elif..else` to using dictionaries of callables. Along the way, we'll also use `try..else`:

```
"""Kafka - the adventure game you cannot win."""


def play():

    position = (0, 0)
    alive = True
```

```
    while position:

        if position == (0, 0):
            print("You are in a maze of twisty passages, all alike.")
        elif position == (1, 0):
            print("You are on a road in a dark forest. To the north you can see a tower
        elif position == (1, 1):
            print("There is a tall tower here, with no obvious door. A path leads east.
        else:
            print("There is nothing here.")

        command = input()

        i, j = position
        if command == "N":
            position = (i, j + 1)
        elif command == "E":
            position = (i + 1, j)
        elif command == "S":
            position = (i, j - 1)
        elif command == "W":
            position = (i - 1, j)
        elif command == "L":
            pass
        elif command == "Q":
            position = None
        else:
            print("I don't understand")

    print("Game over")

if __name__ == '__main__':
    play()
```

The game-loop uses two `if..elif..else` chains. The first prints information dependent on the players current position. Then, after accepting a command from the user, the second `if..elif..else` chain takes action based upon the command.

Let’s refactor this code to avoid those long `if..elif..else` chains, both of which feature repeated comparisons of the same variable against different values.

The first chain describes our current location. Fortunately in Python 3, although not in Python 2, `print()` is a function, and can therefore be used in an expression. We’ll leverage this to build a mapping of position to callables called `locations`:

```
locations = {
    (0, 0): lambda: print("You are in a maze of twisty passages, all alike."),
    (1, 0): lambda: print("You are on a road in a dark forest. To the north you can see
```

```
 tower."),
    (1, 1): lambda: print("There is a tall tower here, with no obvious door. A path lea
 east.")
    }
```

We’ll look up a callable using our position in `locations` as a key, and call the resulting callable in a try block:

```
try:
    locations[position]()
except KeyError:
    print("There is nothing here.")
```

In fact, we don’t really intend to be catching `KeyError` from the *callable*, only from the dictionary lookup, so this also gives us opportunity to narrow the scope of the try block using the `try..else` construct we learned about earlier. Here’s the improved code:

```
try:
    location_action = locations[position]
except KeyError:
    print("There is nothing here.")
else:
    location_action()
```

We separate the lookup and the call into separate statements, and move the call into the `else` block.

Similarly, we can refactor the `if..elif..else` chain which handles user input into dictionary lookup for a callable. This time, though, we used named functions rather than lambdas to avoid the restriction that lambdas can only contain expressions and not statements. Here’s the branching construct:

```
actions = {
    'N': go_north,
    'E': go_east,
    'S': go_south,
    'W': go_west,
    'L': look,
    'Q': quit,
}

try:
    command_action = actions[command]
except KeyError:
    print("I don't understand")
else:
    position = command_action(position)
```

Again we split the lookup of the command action from the call to the command action.

Here are five callables referred to in the dictionary values:

```
def go_north(position):
    i, j = position
    new_position = (i, j + 1)
    return new_position


def go_east(position):
    i, j = position
    new_position = (i + 1, j)
    return new_position


def go_south(position):
    i, j = position
    new_position = (i, j - 1)
    return new_position


def go_west(position):
    i, j = position
    new_position = (i - 1, j)
    return new_position


def look(position):
    return position


def quit(position):
    return None
```

Notice that using this technique forces us into a more functional style of programming. Not only is our code broken down into many more functions, but the bodies of those functions can't modify the state of the `position` variable. Instead, we pass in this value explicitly and return the new value. In the new version mutation of this variable only happens in one place, not five.

Although the new version is larger overall, we'd claim it's much more maintainable. For example, if a new piece of game state, such as the players inventory, were to be added, all command actions would be required to accept and return this value. This makes it much harder to forget to update the state than it would be in chained `if..elif..else` blocks.

Let’s add a new “rabbit hole” location which, when the user unwittingly moves into it, leads back to the starting position of the game. To make such a change, we need to change *all* of our callables in the location mapping to accept and return a position and a liveness status. Although this may seem onerous, we think it’s a good thing. Anyone maintaining the code for a particular location can now see what state needs to be maintained. Here are the location functions:

```
def labyrinth(position, alive):
    print("You are in a maze of twisty passages, all alike.")
    return position, alive


def dark_forest_road(position, alive):
    print("You are on a road in a dark forest. To the north you can see a tower.")
    return position, alive


def tall_tower(position, alive):
    print("There is a tall tower here, with no obvious door. A path leads east.")
    return position, alive


def rabbit_hole(position, alive):
    print("You fall down a rabbit hole into a labyrinth.")
    return (0, 0), alive
```

The corresponding switch in the while-loop now looks like this:

```
locations = {
    (0, 0): labyrinth,
    (1, 0): dark_forest_road,
    (1, 1): tall_tower,
    (2, 1): rabbit_hole,
    }

try:
    location_action = locations[position]
except KeyError:
    print("There is nothing here.")
else:
    position, alive = location_action(position, alive)
```

We must also update the call to `location_action()` to pass the current state and receive the modified state.

Now let’s make the game a little more morbid by adding a deadly lava pit location which returns `False` for the `alive` status. Here’s the function for the lava pit location:

```python
def lava_pit(position, alive):
    print("You fall into a lava pit.")
    return position, False
```

And we must remember to add this to the location dictionary:

```python
locations = {
    (0, 0): labyrinth,
    (1, 0): dark_forest_road,
    (1, 1): tall_tower,
    (2, 1): rabbit_hole,
    (1, 2): lava_pit,
}
```

We'll also add an extra conditional block after we visit the location to deal with deadly situations:

```python
if not alive:
    print("You're dead!")
    break
```

Now when we die, we break out of the `while` loop which is the main game loop. This gives us an opportunity to use a `while..else` clause to handle non-lethal game loop exits, such as choosing to exit the game. Exits like this set the `position` variable to `None`, which is 'falsey':

```python
while position:
    # ...
else:  # nobreak
    print("You have chosen to leave the game.")
```

Now when we quit deliberately, setting `position` to `None` and causing the while-loop to terminate, we see the message from the `else` block associated with the while-loop:

```
You are in a maze of twisty passages, all alike.
E
You are on a road in a dark forest. To the north you can see a tower.
N
There is a tall tower here, with no obvious door. A path leads east.
Q
You have chosen to leave the game.
Game over
```

But when we die by falling into the lava `alive` gets set to `False`. This causes execution to break from the loop, but we don't see the "You have chosen to leave" message as the `else` block is skipped:

```
You are in a maze of twisty passages, all alike.
E
You are on a road in a dark forest. To the north you can see a tower.
N
There is a tall tower here, with no obvious door. A path leads east.
N
You fall into a lava pit.
You're dead!
Game over
```

Dispatching on Type

To "dispatch" on type means that the code which will be executed depends in some way on the type of an object or objects. Python dispatches on type whenever we call a method on an object; there may be several implementations of that method in different classes, and the one that is selected depends on the type of the `self` object.

Ordinarily, we can't use this sort of polymorphism with regular functions. One solution is to resort to switch-emulation to route calls to the appropriate implementation by using `type` objects as dictionary keys. This is ungainly, and it's tricky to make it respect inheritance relationships as well as exact type matches.

`singledispatch`

The `singledispatch` decorator, which we'll introduce in this section, provides a more elegant solution to this problem.

Consider the following code which implements a simple inheritance hierarchy of shapes, specifically a circle, a parallelogram and a triangle, all of which inherit from a base class called `Shape`:

```python
class Shape:

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


class Circle(Shape):

    def __init__(self, center, radius, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.center = center
        self.radius = radius

    def draw(self):
        print("\u25CF" if self.solid else "\u25A1")
```

```python
class Parallelogram(Shape):

    def __init__(self, pa, pb, pc, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.pa = pa
        self.pb = pb
        self.pc = pc

    def draw(self):
        print("\u25B0" if self.solid else "\u25B1")


class Triangle(Shape):

    def __init__(self, pa, pb, pc, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.pa = pa
        self.pb = pb
        self.pc = pc

    def draw(self):
        print("\u25B2" if self.solid else "\u25B3")


def main():
    shapes = [Circle(center=(0, 0), radius=5, solid=False),
              Parallelogram(pa=(0, 0), pb=(2, 0), pc=(1, 1), solid=False),
              Triangle(pa=(0, 0), pb=(1, 2), pc=(2, 0), solid=True)]

    for shape in shapes:
        shape.draw()


if __name__ == '__main__':
    main()
```

Each class has an initializer and a `draw()` method. The initializers store any geometric information peculiar to that type of shape. They pass any further arguments up to the `Shape` base class, which stores a flag indicating whether the shape is `solid`.

When we say:

```python
shape.draw()
```

in `main()`, the particular `draw()` that is invoked depends on whether `shape` is an instance of `Circle`, `Parallelogram`, or `Triangle`. On the receiving end, the object referred to by `shape` becomes referred to by the first formal parameter to the method, which as we know is conventionally called `self`. So we say the call is “dispatched” to the method, depending on the type of the first argument.

This is all very well and is the way much object-oriented software is constructed, but this can lead to poor class design because it violates the single responsibility principle. Drawing isn’t a behaviour inherent to shapes, still less drawing to a particular type of device. In other words, shape classes should be all about shape-ness, not about things you can do with shapes, such as drawing, serialising or clipping.

What we’d like to do is move the responsibilities which aren’t intrinsic to shapes out of the shape classes. In our case, our shapes don’t do anything else, so they become containers of data with no behaviour, like this:

```
class Circle(Shape):

    def __init__(self, center, radius, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.center = center
        self.radius = radius


class Parallelogram(Shape):

    def __init__(self, pa, pb, pc, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.pa = pa
        self.pb = pb
        self.pc = pc


class Triangle(Shape):

    def __init__(self, pa, pb, pc, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.pa = pa
        self.pb = pb
        self.pc = pc
```

With the drawing code removed there are several ways to implement the drawing responsibility outside of the classes. We could us a chain of `if..elif..else` tests using `isinstance()`:

```
def draw(shape):
    if isinstance(shape, Circle):
        draw_circle(shape)
    elif isinstance(shape, Parallelogram):
        draw_parallelogram(shape)
    elif isinstance(shape, Triangle):
        draw_triangle(shape)
```

```
    else:
        raise TypeError("Can't draw shape")
```

In this version of `draw()` we test `shape` using up to three calls to `isinstance()` against `Circle`, `Parallelogram` and `Triangle`. If the `shape` object doesn't match any of those classes, we raise a `TypeError`. This is awkward to maintain and is rightly considered to be very poor programming style.

Another approach is to emulate a switch using dictionary look-up where the dictionary keys are types and the dictionary values are the functions which do the drawing:

```
def draw(shape):
    drawers = {
        Circle: draw_circle,
        Parallelogram: draw_parallelogram,
        Triangle: draw_triangle,
    }

    try:
        drawer = drawers(type(shape))
    except KeyError as e:
        raise TypeError("Can't draw shape") from e
    else:
        drawer(shape)
```

Here we lookup a drawer function by obtaining the type of `shape` in a try block, translating the `KeyError` to a `TypeError` if the lookup fails. If we're on the happy-path of no exceptions, we invoke the drawer with the shape in the `else` clause.

This looks better, but is actually much more fragile because we're doing *exact* type comparisons when we do the key lookup. This means that a subclass of, say, `Circle` wouldn't result in a call to `draw_circle()`.

The solution to these problems arrived in Python 3.4 in the form of `singledispatch`, a decorator defined in the Python Standard Library `functools` module which performs dispatch on type. In earlier versions of Python, including Python 2, you can install the `singledispatch` package from the Python Package Index.

Functions which support multiple implementations dependent on the type of their arguments are called 'generic functions', and each version of the generic function is referred to as an ‘overload’ of the function. The act of providing another version of a generic function for different argument types is called *overloading* the function. These terms are common in statically typed languages such as C#, C++ or Java, but are rarely heard in the context of Python.

To use `singledispatch` we define a function decorated with the `singledispatch` decorator. Specifically, we define a particular version of the function which will be called if a more specific overload has not been provided. We’ll come to the type-specific overloads in a moment.

At the top of the file we need to import `singledispatch`:

```
from functools import singledispatch
```

Then lower down we implement the generic `draw()` function:

```
@singledispatch
def draw(shape):
    raise TypeError("Don't know how to draw {!r}".format(shape))
```

In this case, our generic function will raise a `TypeError`.

Recall that decorators wrap the function to which they are applied and bind the resulting wrapper to the name of the original function. So in this case, the wrapper returned by the decorator is bound to the name `draw`. The `draw` wrapper has an attribute called `register` which is *also* a decorator; `register()` can be used to provide extra versions of the original function which work on different types. This is function overloading.

Since our overloads will all be associated with the name of the original function, `draw`, it doesn’t matter what we call the overloads themselves. By convention we call them `_`, although this is by no means required. Here’s an overload for `Circle`, another for `Parallelogram`, and a third for `Triangle`:

```
@draw.register(Circle):
def _(shape):
    print("\u25CF" if shape.solid else "\u25A1")


@draw.register(Parallelogram)
def _(shape):
    print("\u25B0" if shape.solid else "\u25B1")
```

```
@draw.register(Triangle)
def _(shape):
    # Draw a triangle
    print("\u25B2" if shape.solid else "\u25B3")
```

By doing this we have cleanly separated concerns. Now drawing is dependent on shapes, but not shapes on drawing.

Our main function, which now looks like this, calls the global- scope generic draw function, and the `singledispatch` machinery will select the most specific overload if one exists, or fallback to the default implementation:

```
def main():
    shapes = [Circle(center=(0, 0), radius=5, solid=False),
              Parallelogram(pa=(0, 0), pb=(2, 0), pc=(1, 1), solid=False),
              Triangle(pa=(0, 0), pb=(1, 2), pc=(2, 0), solid=True)]

    for shape in shapes:
        draw(shape)
```

We could add other capabilities to `Shape` in a similar way, by defining other generic functions which behave polymorphically with respect to the shape types.

Using singledispatch with methods

You need to take care not to use the `singledispatch` decorator with methods. To see why, consider this attempt to implement a generic `intersects()` predicate method on the `Circle` class which can be used to determine whether a particular circle intersects instances of any of the three defined shapes:

```
class Circle(Shape):

    def __init__(self, center, radius, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.center = center
        self.radius = radius

    @singledispatch
    def intersects(self, shape):
        raise TypeError("Don't know how to compute intersection with {!r}".format(shape

    @intersects.register(Circle)
    def _(self, shape):
        return circle_intersects_circle(self, shape)
```

```
    @intersects.register(Parallelogram)
    def _(self, shape):
        return circle_intersects_parallelogram(self, shape)

    @intersects.register(Triangle)
    def _(self, shape):
        return circle_intersects_triangle(self, shape)
```

At first sight, this looks like a reasonable approach, but there are a couple of problems here.

The first problem problem is that we can't register the type of the class currently being defined with the `intersects` generic function, because we have not yet finished defining it.

The second problem is more fundamental: Recall that `singledispatch` dispatches based only on the type of the *first* argument:

```
do_intersect = my_circle.intersects(my_parallelogram)
```

When we're calling our new method like this it's easy to forget that `my_parallelogram` is actually the *second* argument to `Circle.intersects`. `my_circle` is the first argument, and it's what gets bound to the `self` parameter. Because `self` will *always* be a `Circle` in this case our `intersect()` call will always dispatch to the first overload, irrespective of the type of the second argument.

This behaviour prevents the use of `singledispatch` with methods. All is not lost however. The solution is to move the generic function out of the class, and invoke it from a regular method which swaps the arguments. Let's take a look:

```
class Circle(Shape):

    def __init__(self, center, radius, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.center = center
        self.radius = radius

    def intersects(self, shape):
        # Delegate to the generic function, swapping arguments
        return intersects_with_circle(shape, self)


@singledispatch
def intersects_with_circle(shape, circle):
    raise TypeError("Don't know how to compute intersection of {!r} with {!r}"
```

```
                    .format(circle, shape))

@intersects_with_circle.register(Circle)
def _(shape, circle):
    return circle_intersects_circle(circle, shape)

@intersects.register(Parallelogram)
def _(shape, circle):
    return circle_intersects_parallelogram(circle, shape)

@intersects.register(Triangle)
def _(shape, circle):
    return circle_intersects_triangle(circle, shape)
```

We move the generic function `intersects()` out to the global scope and rename it to `intersects_with_circle()`. The replacement `intersects()` method of `Circle`, which accepts the formal arguments `self` and `shape`, now delegates to `intersect_with_circle()` with the actual arguments swapped to `shape` and `self`.

To complete this example, we would need to implement two other generic functions, `intersects_with_parallelogram()` and `intersects_with_triangle()`, although will leave that as an exercise.

Combining class-based polymorphism and overloading-based polymorphism in this way would give us a complete implementation of not just single-dispatch but *double*-dispatch, allowing us to do:

```
shape.intersects(other_shape)
```

This way, the function is selected based on the types of both `shape` *and* `other_shape`, without the shape classes themselves having any knowledge of each other, keeping coupling in the system manageable.

Summary

That just about wraps up this chapter on advanced flow control in Python 3. Let’s summarise what we’ve covered:

- We looked at `else` clauses on while-loops, drawing an analogy with the much more well-known association between `if` and `else`. We showed how the `else` block is executed only when the while-loop condition evaluates to `False`. If the loop is exited by another means, such as via `break` or `return`, the `else` clause is *not* executed. As such, `else` clauses on while-loops are only ever useful if the loop contains a `break` statement somewhere within it.

- The loop-else clause is a somewhat obscure and little-used construct. We strongly advise commenting the `else` keyword with a “nobreak” remark so it is clear under what conditions the block is executed.
- The related `for..else` clause works in an identical way: The `else` clause is effectively the “nobreak” clause, and is only useful if the loop contains a break statement. They are most useful with for-loops in searching. When an item is found while iterating we `break` from the loop — skipping the `else` clause. If no items are found and the loop completes ‘naturally’ without breaking ,the `else` clause is executed and code handling the “not found” condition can be implemented.
- Many uses of loop `else` clauses — particularly for searching — may be better handled by extracting the loop into its own function. From here execution is returned directly when an item is found and code after the loop can handle the “not found” case. This is less obscure, more modular, more reusable, and more testable than using a loop `else` clause within a longer function.
- Next we looked at the `try..except..else` construct. In this case, the `else` clause is executed only if the try block completed successfully without any exception being raised. This allows the extent of the `try` block to be narrowed, making it clearer from where we are expecting exceptions to be raised.
- Python lacks a “switch” or “case” construct to implement multi-branch control flow. We showed the alternatives, including chained `if..elif..else` blocks, and dictionaries of callables. The latter approach also forces you to be more explicit and consistent about what is required and produced by each branch, since you must pass arguments and return values rather than mutating local state in each branch.
- We showed how to implement generic functions which dispatch on type using the `singledispatch` decorator available from Python 3.4. This decorator can be applied only to module scope functions, not methods, but by implementing forwarding methods and argument swapping, we can delegate to generic functions from methods. This gives us a way of implementing double-dispatch calls.
- In passing, we saw that the Python logical operators use short-circuit evaluation. This means that the operators only evaluate as many

operands as are required to find the result. This can be used to ‘protect’ expressions from run time situations in which they would not make sense.

Chapter 2 - Byte Oriented Programming

In this chapter we’ll be going low-level and look at byte-oriented programming in Python. At the bottom, everything is bits and bytes, and sometimes it’s necessary to work at this level, particularly when dealing with binary data from other sources.

Remember that in Python 3 — although not in Python 2 — there is a very clear separation between text data, which are stored in the Unicode capable `str` type, and raw bytes, which are stored in the aptly named `bytes` type.

Specifically, we’re going to:

- review the bitwise operators
- look at the binary representation of integers
- fill in some additional details about the `bytes` type
- introduce the `bytearray` type
- packing and unpacking binary data with the `struct` module
- look at the `memoryview` object
- memory-mapped files

Bitwise operators

Let’s start right at the bottom with the bitwise operators. These will seem straightforward enough, but our exploration of them will lead us into some murky corners of Python’s integer implementation.

We covered the bitwise-and, bitwise-or, and shift operators in Chapter 9 of *The Python Apprentice* when reading binary BMP image files. Now we’ll complete the set by introducing the bitwise exclusive-or operator and the bitwise complement operator. Along the way, we’ll also use bitwise shifts:

Operator	Description
&	Bitwise AND

<table>
<tr><td>|</td><td>Bitwise OR</td></tr>
<tr><td>^</td><td>Bitwise XOR</td></tr>
<tr><td>~</td><td>Bitwise NOT</td></tr>
<tr><td><<</td><td>Left shift</td></tr>
<tr><td>>></td><td>Right shift</td></tr>
</table>

We'll demonstrate how each of these work, but this will necessitate an interesting detour into Python's integer representation.

> Recall that we can specify binary literals using the `0b` prefix and we can display integers in binary using the built-in `bin()` function:
>
> ```
> >>> 0b11110000
> 240
> >>> bin(240)
> '0b11110000'
> ```

Exclusive-or

The exclusive-OR operator behaves exactly as you would expect, setting bits in the output value if exactly one of the corresponding operand bits is set:

```
>>> bin(0b11100100 ^ 0b00100111)
'0b11000011'
```

Bitwise complement (bitwise not)

The bitwise complement — or NOT — operator is more unexpectedly difficult to demonstrate, although very easy to use:

```
>>> bin(~0b11110000)
'-0b11110001'
```

You were probably expecting `0b00001111` as the result, although of course Python doesn't usually print leading zeros, so `0b1111` is perhaps more reasonable. Actually, leading zeros are part of the reason we get the surprising `-0b1110001` result.

Two's-complement

Computer number systems typically represent negative integer numbers using a system called two’s-complement. Here’s a refresher on how two’s complement works.

8-bit two’s-complement can represent numbers from -128 to 127. Let’s see how to represent the number -58 in 8-bit two’s complement:

1. We start with the signed decimal -58, the value we want to represent.
2. We take the absolute value, which is 58, and represent this in 8-bit binary, which is 00111010. The same binary pattern can be interpreted as the *unsigned* decimal 58.
3. Because -58 is negative, we must apply additional steps 4 and 5, below. Positive numbers have the same representation in signed and unsigned representations, so don’t require these additional steps.
4. Now we flip all the bits — a bitwise-not operation — to give 11000101. This is where the “two” in “two’s complement” comes from: the complement operation is done in base two (or binary). This gives a bit-pattern that would be interpreted as 197 as an unsigned integer, although that’s not too important here.
5. Finally, we add one, which gives the bit pattern 11000110 which, although it could be interpreted as the unsigned integer 198, is outside the range -128 to +127 allowed for *signed* 8-bit integers. This is the two’s complement 8-bit representation for -58.

Two’s complement has particular advantages over other representations, such as a sign-bit and magnitude scheme. For example, two’s-complement works naturally with arithmetic operations such as addition and subtraction involving negative numbers.

Two’s-complement and unlimited precision integers

Recall that Python 3 uses arbitrary precision integers. That is, Python integers are not restricted to one, two, four or eight bytes, but can use as many bytes as are necessary to store integers of any magnitude.

However, two’s-complement present a particular problem when used with unlimited precision number. When we take the complement by flipping the bits, how many leading zeros should we flip? In a fixed-precision integer representation the answer is obvious. But what about variable bit-width integers. What if we flipped an unlimited (*i.e.* infinite) number of leading zeros to give an infinite number of leading ones? How would you interpret the result?

Of course, Python doesn't really represent negative integers using an infinite number of leading ones, but conceptually this is what is going on.

This is why, when asked to represent negative numbers in binary, Python actually uses a leading unary minus as part of magnitude representation, rather than a giving the unsigned binary two's-complement representation:

```
>>> bin(4)
'0b100'
>>> bin(-4)
'-0b100'
```

This means we don't easily get to see the internal bit representation of negative numbers using `bin()` — in fact, we can't even determine what internal representation scheme is used! This can make it tricky to inspect the behaviour of code which uses the bitwise operators.

When our use of the bitwise operators — such as the bitwise-not operator — results in a bit pattern that would normally represent a negative integer in two's complement format, Python displays that value in sign- magnitude format, obscuring the result we wanted. To get at the *actual* bit pattern, we need to do a little extra work. Let's return to our earlier example:

	Binary two's complement (9-bit)	Decimal	Binary sign-magnitude
v	0b011110000	240	+0b11110000
~v	0b100001111	-241	-0b11110001

Let's make `v` equal to the value `0b11110000`:

```
>>> v = 0b11110000
```

This value is equal to 240 in signed decimal:

```
>>> v
240
```

Now we'll use the bitwise-not operator:

```
>>> ~v
-0b11110001
```

Here’s the explanation of what’s just happened.Working with the binary representation of 240 we need an 8-bit representation, although to accommodate the two’s-complement representation for positive integers we need to accommodate at least one zero, so really we need at least a 9-bit representation.

It is the bits of this 9-bit representation to which the bitwise-not is applied, giving 100001111. This is the two-complement representation of -241.

Displayed back in the sign-magnitude representation Python uses when displaying binary numbers, -241 is displayed as -0b11110001.

A more intuitive two’s-complement

So much for the explanation, but how do we see the flipped bits arising from our application of bitwise-not in a more intuitive way? One approach is to manually take the two’s-complement of the *magnitude*, or absolute value of the number:

```
     -0b11110001
flip    00001110
add 1   00001111
```

Unfortunately, this also uses the bitwise-not operator, and we end up chasing our tail trying to escape the cleverness of Python’s arbitrary precision integers.

Another approach is to rely on the characteristics of two’s complement arithmetic:

1. Take the signed interpretation of the two’s complement binary value, in this case -241.
2. Add to it 2 raised to the power of the number of bits used in the representation, excluding the leading ones. In this case that’s 2^8 or 256, and -241 + 256 is 15
3. 15 has the 00001111 bit pattern we’re looking for — the binary value we expected to get when we applied bitwise-not to 11110000.

Remember that the two's complement of a positive number is itself, so our function needs to take account of this:

```
>>> def twos_complement(x, num_bits):
...     if x < 0:
...         return x + (1 << num_bits)
...     return x
```

If you have difficulty understanding how this works — don't worry — it *is* tricky. Fifteen minutes with pencil, paper, and a Python interpreter will be well rewarded.

An intuitive two's-complement with bitwise-and

An less usable, but perhaps more obvious approach, is to use bitwise-and with a mask of 1s to discard all the leading 1s in negative results. This effectively specifies to how many bits precision we want the result to be presented. So for an 8 bit result we do this:

```
>>> bin(~0b11110000 & 0b11111111)
'0b1111'
```

giving what we wanted. See how asking for a 9 bit result reveals the leading 1 of the two's complement representation:

```
>>> bin(~0b11110000 & 0b111111111)
'0b100001111'
```

In fact, since Python 3.2, we can ask Python how many bits are required to represent the integer value using the `bit_length()` method of the integer type, although notice that this excludes the sign:

```
>>> int(32).bit_length()
6
>>> int(240).bit_length()
8
>>> int(-241).bit_length()
8
>>> int(256).bit_length()
9
```

Examining bits with `to_bytes()`

We can also retrieve a byte-oriented representation directly as a `bytes` object using the `to_bytes()` method[5]:

```
>>> int(0xcafebabe).to_bytes(length=4, byteorder='big')
b'\xca\xfe\xba\xbe'
>>> int(0xcafebabe).to_bytes(length=4, byteorder='little')
b'\xbe\xba\xfe\xca'
```

If you want to use the native bytes order, you can retrieve the `sys.byteorder` value:

```
>>> import sys
>>> sys.byteorder
'little'
>>> little_cafebabe = int(0xcafebabe).to_bytes(length=4, byteorder=sys.byteorder)
>>> little_cafebabe
b'\xbe\xba\xfe\xca'
```

Of course, given some bytes we can also turn them back into an integer, using the complementary class method, `from_bytes()`:

```
>>> int.from_bytes(little_cafebabe, byteorder=sys.byteorder)
3405691582
>>> hex(_)
'0xcafebabe'
```

Asking for the byte-oriented representation of an integer using `to_bytes()` will fail for negative integers with an `OverflowError`:

```
>>> int(-241).to_bytes(2, byteorder='big')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: can't convert negative int to unsigned
```

However, if we set the optional `signed` argument to `True`, rather than its default value of `False` we can get a two's-complement representation back:

```
>>> int(-241).to_bytes(2, byteorder='little', signed=True)
b'\x0f\xff'
```

This indicates another way to answer the question that started this quest, by indexing into the result bytes object to retrieve the least significant byte, and converting that to a binary representation:

```
>>> bin((~0b11110000).to_bytes(2, byteorder='little', signed=True)[0])
'0b1111'
```

While this may be useful, it's certainly not concise.

The `bytes` type in depth

We first introduced the `bytes` type way back in Chapter 2 of *The Python Apprentice*. We did this to highlight the essential differences between `str`, which is the immutable sequence of Unicode codepoints and `bytes` which is the immutable sequence of bytes. This is particularly important if you're coming to Python 3 from Python 2, where `str` behaved differently.

Literals `bytes``

At this point you should be comfortable with the `bytes` literal, which uses the `b` prefix. The default Python source code encoding is UTF-8, so the the characters used in a *literal* byte string are restricted to printable 7-bit ASCII characters — that is those with codes from 0 to 127 inclusive which aren't control codes:

```
>>> b"This is OK because it's 7-bit ASCII"
b"This is OK because it's 7-bit ASCII"
```

Seven-bit control codes or characters which can't be encoded in 7-bit ASCII result in a `SyntaxError`:

```
>>> b"Norwegian characters like Å and Ø are not 7-bit ASCII"
  File "<stdin>", line 1
SyntaxError: bytes can only contain ASCII literal characters.
```

To represent other bytes with values equivalent to ASCII control codes and byte values from 128 to 255 inclusive we must use escape sequences:

```
>>> b"Norwegian characters like \xc5 and \xd8 are not 7-bit ASCII"
b'Norwegian characters like \xc5 and \xd8 are not 7-bit ASCII'
```

Notice that Python echoes these back to us as escape sequences too. This is just a sequence of bytes, not a sequence of characters. If we want a sequence of Unicode code points we must decode the bytes into a text sequence of the `str` type, and for this we need to know the encoding. In this case I used Latin 1:

```
>>> norsk = b"Norwegian characters like \xc5 and \xd8 are not 7 bit ASCII"
>>> norsk.decode('latin1')
'Norwegian characters like Å and ø are not 7 bit ASCII'
```

Indexing and slicing `bytes`

Notice that when we retrieve an item from the `bytes` object by indexing, we get an `int` object, not a one byte sequence:

```
>>> norsk[0]
78
>>> type(norsk[0])
<class 'int'>
```

This is another fundamental difference between `bytes` and `str`.

Slicing of bytes objects however *does* return a new bytes object:

```
>>> norsk[21:25]
b'like'
```

The `bytes` constructors

There are a few other forms of the `bytes` constructor it's good to be aware of. You can create a zero length bytes sequence simply by calling the constructor with no arguments:

```
>>> bytes()
b''
```

You can create a zero-filled sequence of bytes by passing a single integer to the `bytes` constructor:

```
>>> bytes(5)
b'\x00\x00\x00\x00\x00'
```

You can also pass an iterable series of integers:

```
>>> bytes(range(65, 65+26))
b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

It's up to you to ensure that the values are non-negative and less than 256 to prevent a `ValueError` being raised:

```
>>> bytes([63, 127, 255, 511])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: bytes must be in range(0, 256)
```

One option if you need to construct a `bytes` object by encoding a Unicode `str` object is to use the two-argument form of the `bytes` constructor, which accepts a `str` in the first argument and and encoding for the second:

```
>>> bytes('Norwegian characters Å and Ø', 'utf16')
b'\xff\xfeN\x00o\x00r\x00w\x00e\x00g\x00i\x00a\x00n\x00 \x00c\x00h\x00a\x00r\x00a\x00c\x00t\x00e\x00r\x00s\x00 \x00\xc5\x00 \x00a\x00n\x00d\x00 \x00\xd8\x00'
```

Finally there is a class method `fromhex()` which is a factory method for creating a `bytes` object from a string consisting of concatenated two digit hexadecimal numbers:

```
>>> bytes.fromhex('54686520717569636b2062726f776e20666f78')
b'The quick brown fox'
```

There isn't a method to go in the other direction, so we have to use a generator expression to convert each byte to its hex representation, stripping the leading `0x` from each resulting string using slicing:

```
>>> ''.join(hex(c)[2:] for c in b'The quick brown fox')
'54686520717569636b2062726f776e20666f78'
```

The mutable `bytearray` sequence

The `bytes` type we just looked at is only one of several so-called *binary sequence types* in Python. Whereas `bytes` is immutable, the `bytearray` type is mutable. This means it supports many of the same operations as the other mutable sequence type with which you are already deeply familiar: `list`.

The `bytearray` constructors

The `bytearray` type supports the same constructors as `bytes`, so we won't go over them in detail here:

```
>>> bytearray()
bytearray(b'')
>>> bytearray(5)
bytearray(b'\x00\x00\x00\x00\x00')
>>> bytearray(b'Construct from a sequence of bytes')
bytearray(b'Construct from a sequence of bytes')
>>> bytearray('Norwegian characters Å and Ø', 'utf16')
bytearray(b'\xff\xfeN\x00o\x00r\x00w\x00e\x00g\x00i\x00a\x00n\x00 \x00c\x00h\x00a\x00r\x00a\x00c\x00t\x00e\x00r\x00s\x00 \x00\xc5\x00 \x00a\x00n\x00d\x00 \x00\xf8\x00')
>>> bytearray.fromhex('54686520717569636b2062726f776e20666f78')
bytearray(b'The quick brown fox')
```

Being mutable we can use any of the mutable sequence operations to modify the `bytearray`, in place:

```
>>> pangram = bytearray(b'The quick brown fox')
>>> pangram.extend(b' jumps over the lazy dog')
>>> pangram
bytearray(b'The quick brown fox jumps over the lazy dog')
>>> pangram[40:43] = b'god'
>>> pangram
bytearray(b'The quick brown fox jumps over the lazy god')
```

String-like operations on `bytearray`

The `bytearray` type supports the same operations as `list` together with many of the string-like operations supported by `bytes`, such as `upper()`, `split()` and `join()`:

```
>>> pangram.upper()
bytearray(b'THE QUICK BROWN FOX JUMPS OVER THE LAZY GOD')
>>>
>>> words = pangram.split()
>>> words
[bytearray(b'The'), bytearray(b'quick'), bytearray(b'brown'), bytearray(b'fox'), bytear
y(b'jumps'), bytearray(b'over'), bytearray(b'the'), bytearray(b'lazy'), bytearray(b'god
]
>>>
>>> bytearray(b' ').join(words)
bytearray(b'The quick brown fox jumps over the lazy god')
```

When using string-like operations such as `upper()` or `capitalize()` on the `bytes` and `bytearray` types you must take particular care that only 7-bit ASCII byte strings are in play.

Interpreting byte streams with the `struct` module

The Python Standard Library `struct` module is so called because it is capable of decoding the byte pattern of `struct` objects from the C and C++ languages. Structs — short for structures — are composite data types which, in binary form, consist of the byte patterns for C language primitives concatenated together. It may not be immediately obviously why this is necessary — or even useful — in a language like Python, but being very close to the machine level, C data types are something of a low-level *lingua franca* for binary data exchange between programs in many languages. Furthermore, as most operating systems are written in C or C++, the ability to interpret C structures into Python values and back again takes on even greater significance.

In this demo, we're going to:

- Write a binary file from a C program
- Read the file from a Python program
- Create Python classes which mirror C structures
- Demonstrate diagnostic techniques for dealing with binary data

Writing structs from C

This isn't a course on C, but we're sure you'll understand the following C structures for `Vector`, `Color` and `Vertex` in this program written in the C99 variant of the C language:

```
/**
 *  colorpoints.c
 *
 *  A C99 program to write a colored vertex
 *  structures to a binary file.
 *
 */

#include <stdio.h>

struct Vector {
    float x;
    float y;
    float z;
};

struct Color {
    unsigned short int red;
    unsigned short int green;
    unsigned short int blue;
};

struct Vertex {
    struct Vector position;
    struct Color color;
};

int main(int argc, char** argv) {
    struct Vertex vertices[] = {
        { .position = { 3323.176, 6562.231, 9351.231 },
          .color = { 3040, 34423, 54321 } },

        { .position = { 7623.982, 2542.231, 9823.121 },
          .color = { 32736, 5342, 2321 } },

        { .position = { 6729.862, 2347.212, 3421.322 },
```

```
          .color = { 45263, 36291, 36701 } },

        { .position = { 6352.121, 3432.111, 9763.232 },
          .color = { 56222, 36612, 11214 } } };

    FILE* file = fopen("colors.bin", "wb");

    if (file == NULL) {
        return -1;
    }

    fwrite(vertices, sizeof(struct Vertex), 4, file);
    fclose(file);

    return 0;
}
```

As you can see, we declare a `Vector` to be comprised of three `float` values. It's important to realise that a C float is actually a single-precision floating point value represented with 32 bits, which is different from a Python `float`, which is a double-precision floating point value represented with 64 bits.

We then declare a `Color` structure, which comprises three unsigned short integers. Again, this integer type is quite different from what we have available in Python, where we have arbitrary precision *signed* integers. In fact, having just 16 bits of precision, C's `unsigned short int` can only represent values from 0 through to 65535.

The third structure we declare is a `Vertex`, which combines a `Vector` and a `Color` together into one larger structure.

In the `main()` function, the program creates an array of four `Vertex` structures, and writes them to a file called `colors.bin` before exiting.

We'll now compile this C program into an executable. The details of how you do this are heavily system dependent, and require that you at least have access to a C99 compiler. On our macOS system with the XCode development tools installed, we can simply use `make` from the command line:

```
$ make colorpoints
cc     colorpoints.c   -o colorpoints
```

This produces an executable called `colorpoints`:

```
$ ls
colorpoints   colorpoints.c
```

When we run the executable, a `colors.bin` file is produced, as expected:

```
$ ./colorpoints
$ ls
colorpoints   colorpoints.c colors.bin
```

Interpreting structs in Python

Now that we have a binary data file, let's try to make sense of it in Python. Our starting point will be a simple program called `reader.py` to read only the first vertex of data from the file:

```
import struct


def main():
    with open('colors.bin', 'rb') as f:
        buffer = f.read()

    items = struct.unpack_from('@fffHHH', buffer)

    print(repr(items))


if __name__ == '__main__':
    main()
```

In the main function, we open the file for read, being careful to remember to do so in binary mode. We then use the `read()` method of file objects to read the entire contents of the file in to a `bytes` object.

We use the `struct.unpack_from()` function to convert the raw byte sequence into more friendly types. This function accepts a special format string containing codes which specify how to interpret the bytes.

The leading `@` character specifies that native byte order and alignment are to be used. Other characters can be used to force particular byte orderings, such as < for little-endian, and > for big-endian. It's also possible to choose between native and no *alignment*, a topic we'll be revisiting shortly. If no byte-order character is specified, then `@` is assumed.

There are also code letters corresponding to all of the common C data types, which are mostly variations on different precisions of signed and unsigned integers, together with 32- and 64-bit floating point numbers, byte arrays, pointers and fixed-length strings.

In our example, each of the three ‘f’ characters tells `struct` to expect a single-precision C `float`, and each of the ‘H’ characters tells `struct` to expect an `unsigned short int`, which is a 16-bit type.

Let’s run our program:

```
$ python3 reader.py
(3323.176025390625, 6562.23095703125, 9351.2314453125, 3040, 34423, 54321)
```

We can see that `struct.unpack_from` returns a `tuple`. The reason our values don’t look *exactly* the same as they did in the source code to our C program, is because we specified values in *decimal* in the source, and the values we chose are not representable exactly in binary. There has also been a conversion from the single-precision C `float` to the double-precision Python `float`, which is why the values we get back have so many more digits. Of course, the 16 bit `unsigned short int` values from C can be represented exactly as Python `int` objects.

TODO: We need to show the program output here!

Tuple unpacking

One obvious improvement to our program, given that `unpack_from()` returns a `tuple`, is to use tuple unpacking to place the values into named variables:

```
x, y, z, red, green, blue = struct.unpack_from('@fffHHH', buffer)
```

We can also shorten our format string slightly by using repeat counts. For example `3f` means the same as `fff`:

```
x, y, z, red, green, blue = struct.unpack_from('@3f3H', buffer)
```

That’s not a big win in this case, but it can be very useful for larger data series.

Reading all of the vertices

Finally, of course, we’d like to read all four `Vertex` structures from our file. We’ll also make the example a bit more realistic by reading the data into

Python classes which are equivalents of the `Vector`, `Color`, and `Vertex` `structs` we had in C. Here they are:

```
class Vector:

    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def __repr__(self):
        return 'Vector({}, {}, {})'.format(self.x, self.y, self.z)


class Color:

    def __init__(self, red, green, blue):
        self.red = red
        self.green = green
        self.blue = blue

    def __repr__(self):
        return 'Color({}, {}, {})'.format(self.red, self.green, self.blue)


class Vertex:

    def __init__(self, vector, color):
        self.vector = vector
        self.color = color

    def __repr__(self):
        return 'Vertex({!r}, {!r})'.format(self.vector, self.color)
```

We'll also make a factory function to construct a type Vertex, which aggregates a `Vector` and a `Color`, being careful to use an argument order that is compatible with what we get back from the unpack function:

```
def make_colored_vertex(x, y, z, red, green, blue):
    return Vertex(Vector(x, y, z),
                  Color(red, green, blue))
```

We'll add an import for pretty-printing at the top of the module:

```
from pprint import pprint as pp
```

Finally we'll re-work the `main()` function to use `struct.iter_unpack()`:

```
def main():
    with open('colors.bin', 'rb') as f:
        buffer = f.read()

    vertices = []
```

```
    for x, y, z, red, green, blue in struct.iter_unpack('@3f3H', buffer):
        vertex = make_colored_vertex(x, y, z, red, green, blue)
        vertices.append(vertex)

    pp(vertices)
```

In fact, we can unwind one our earlier refactorings, and simply unpack the tuple directly into the arguments of `make_colored_vertex()` using extended call syntax:

```
def main():
    with open('colors.bin', 'rb') as f:
        buffer = f.read()

    vertices = []
    for fields in struct.iter_unpack('@3f3H', buffer):
        vertex = make_colored_vertex(*fields)
        vertices.append(vertex)

    pp(vertices)
```

In this code, `fields` will will be the tuple of three `float` and three `int` values returned for each structure. Rather than unpacking into named variables, we use extended call syntax to unpack the `fields` tuple directly into the arguments of `make_colored_vertex()`. When we've accumulated all vertices into a list, we pretty-print the resulting data structure. Let's try it!:

```
$ python3 reader.py
Traceback (most recent call last):
  File "examples/reader.py", line 59, in <module>
    main()
  File "examples/reader.py", line 52, in main
    for fields in struct.iter_unpack('@3f3H', buffer):
struct.error: iterative unpacking requires a bytes length multiple of 18
```

Oh dear! What happened?

Accounting for padding

The `struct.iter_unpack()` function is complaining because it expects the `buffer` byte sequence to be a multiple of 18 bytes long. Each of our structures consists of three 4-byte floats and three 2-byte unsigned short integers. We know that $3 \times 4 + 3 \times 2 = 18$. So how long is our buffer? Let's add some temporary diagnostic code to our program. After we read the file, we'll print the buffer length, and the buffer contents:

```
print("buffer: {} bytes".format(len(buffer)))
print(buffer)
```

When we run now, we get this output before the stack trace:

```
$ python3 reader.py
buffer: 80 bytes
b"\xd1\xb2OE\xd9\x11\xcdE\xed\x1c\x12F\xe0\x0bw\x861\xd4\x00\x00\xdb?\xeeE\xb2\xe3\x1eE
\x19F\xe0\x7f\xde\x14\x11\t\x00\x00\xe5N\xd2Ed\xb3\x12E'\xd5UE\xcf\xb0\xc3\x8d]\x8f\x00
00\xf8\x80\xc6E\xc7\x81VE\xee\x8c\x18F\x9e\xdb\x04\x8f\xce+\x00\x00"
Traceback (most recent call last):
  File "example/reader.py", line 56, in <module>
    main()
  File "example/reader.py", line 48, in main
    for fields in struct.iter_unpack('@3f3H', buffer):
struct.error: iterative unpacking requires a bytes length multiple of 18
```

Curiously, the buffer is reported as having 80 bytes, and checking at the command line, we can see that that is consistent with the file length:

```
$ ls -l
total 40
-rwxr-xr-x  1 rjs  staff  8732 11 Dec 12:32 colorpoints
-rw-r--r--  1 rjs  staff  1029 11 Dec 11:34 colorpoints.c
-rw-r--r--  1 rjs  staff    80 11 Dec 12:33 colors.bin
-rw-r--r--  1 rjs  staff  1068 11 Dec 17:39 reader.py
```

We're expecting $4 \times 18 = 72$ bytes, so where are the extra eight bytes coming from?

Our diagnostic print statements above are a good start, but it's really awkward to read the standard `bytes` representation, especially when it contains a mix of ASCII characters and escape sequences. We can't directly convert a binary sequence into a readable hex string, but Python 3 has some tools in the standard library to help, in the form of the `binascii` module, which contains the oddly named `hexlify()` function. Let's import it:

```
from binascii import hexlify
```

and modify our diagnostic `print` statement to:

```
print(hexlify(buffer))
```

This gives us a long string, which is perhaps not even an improvement!

```
b'd1b24f45d911cd45ed1c1246e00b778631d40000db3fee45b2e31e457c7c1946e07fde1411090000e54ed
564b3124527d55545cfb0c38d5d8f0000f880c645c7815645ee8c18469edb048fce2b0000'
```

What we really want to do is to split pairs of digits with spaces. We can do this by slicing out consecutive pairs:

```
hex_buffer = hexlify(buffer).decode('ascii')
hex_pairs = ' '.join(hex_buffer[i:i+2] for i in range(0, len(hex_buffer), 2))
print(hex_pairs)
```

In this code, we `hexlify` then encode to an ASCII string. We then `join()` the successive two-digit slices with spaces, using a `range()` expression with a step of 2. We then print the hex pairs:

```
d1 b2 4f 45 d9 11 cd 45 ed 1c 12 46 e0 0b 77 86 31 d4 00 00 db 3f ee 45 b2 e3 1e 45 7c
 19 46 e0 7f de 14 11 09 00 00 e5 4e d2 45 64 b3 12 45 27 d5 55 45 cf b0 c3 8d 5d 8f 00
0 f8 80 c6 45 c7 81 56 45 ee 8c 18 46 9e db 04 8f ce 2b 00 00
```

This is a big improvement, but still leaves us counting bytes on the display. Let’s precede our line of data with an integer count:

```
indexes = ' '.join(str(n).zfill(2) for n in range(len(buffer)))
print(indexes)
```

We generate integers using a range, convert then to strings and pad each number with leading zeros to a width of two; good enough for the first hundred bytes of our data.

Finally, we have something we can work with[6]:

```
buffer: 80 bytes
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19
d1 b2 4f 45 d9 11 cd 45 ed 1c 12 46 e0 0b 77 86 31 d4 00 00

20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
db 3f ee 45 b2 e3 1e 45 7c 7c 19 46 e0 7f de 14 11 09 00 00

40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59
e5 4e d2 45 64 b3 12 45 27 d5 55 45 cf b0 c3 8d 5d 8f 00 00

60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
f8 80 c6 45 c7 81 56 45 ee 8c 18 46 9e db 04 8f ce 2b 00 00
```

Now we’ve got a useful way of viewing our bytes object, let’s get back to diagnosing our problem of why we have 80 bytes rather than 72. Looking carefully, we can see that the first bytes at indices 0 to 17 inclusive contain legitimate data — and we know this to be the case because we decoded it earlier. Looking at bytes 18 and 19 though, we see two zero bytes. From bytes 20 to 37, we have another run of what looks like legitimate data, again followed by another two zero bytes at indices 38 and 39.

This pattern continues to the end of the file. What we're seeing is 'padding' added by the C compiler to align structures on four-byte boundaries. Our 18 byte structure needs to be padded with two bytes to take it to 20 bytes which is divisible by four. In order to skip this padding we can use 'x' which is the format code for pad bytes. In this case there are two pad bytes per structure, so we can add 'xx' to our format string:

```
vertices = []
for fields in struct.iter_unpack('@3f2Hxx', buffer):
    vertex = make_colored_vertex(*fields)
    vertices.append(vertex)
```

With this change in place, we can successfully read our structures from C into Python!:

```
buffer: 80 bytes
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19
d1 b2 4f 45 d9 11 cd 45 ed 1c 12 46 e0 0b 77 86 31 d4 00 00

20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
db 3f ee 45 b2 e3 1e 45 7c 7c 19 46 e0 7f de 14 11 09 00 00

40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59
e5 4e d2 45 64 b3 12 45 27 d5 55 45 cf b0 c3 8d 5d 8f 00 00

60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
f8 80 c6 45 c7 81 56 45 ee 8c 18 46 9e db 04 8f ce 2b 00 00
[Vertex(Vector(3323.176025390625, 6562.23095703125, 9351.2314453125),
        Color(3040, 34423, 54321)),
 Vertex(Vector(7623.98193359375, 2542.23095703125, 9823.12109375),
        Color(32736, 5342, 2321)),
 Vertex(Vector(6729.86181640625, 2347.2119140625, 3421.322021484375),
        Color(45263, 36291, 36701)),
 Vertex(Vector(6352.12109375, 3432.111083984375, 9763.232421875),
        Color(56222, 36612, 11214))]
```

Here's the complete code so far:

```
from binascii import hexlify
from pprint import pprint as pp
import struct

class Vector:

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

```python
    def __repr__(self):
        return 'Vector({}, {}, {})'.format(self.x, self.y, self.z)


class Color:

    def __init__(self, red, green, blue):
        self.red = red
        self.green = green
        self.blue = blue

    def __repr__(self):
        return 'Color({}, {}, {})'.format(self.red, self.green, self.blue)


class Vertex:

    def __init__(self, vector, color):
        self.vector = vector
        self.color = color

    def __repr__(self):
        return 'Vertex({!r}, {!r})'.format(self.vector, self.color)


def make_colored_vertex(x, y, z, red, green, blue):
    return Vertex(Vector(x, y, z),
                  Color(red, green, blue))


def main():
    with open('colors.bin', 'rb') as f:
        buffer = f.read()

    print("buffer: {} bytes".format(len(buffer)))
    indexes = ' '.join(str(n).zfill(2) for n in range(len(buffer)))
    print(indexes)
    hex_buffer = hexlify(buffer).decode('ascii')
    hex_pairs = ' '.join(hex_buffer[i:i+2] for i in range(0, len(hex_buffer), 2))
    print(hex_pairs)

    vertices = []
    for fields in struct.iter_unpack('@3f3Hxx', buffer):
        vertex = make_colored_vertex(*fields)
        vertices.append(vertex)

    pp(vertices)


if __name__ == '__main__':
    main()
```

> From this point onwards, we'll no longer need the diagnostic print statements which helped us get this far, so consider them removed from this point onwards.

Understanding “native” in context

You might be wondering why, given that we used the `@` character at the beginning of our format string to specify native byte-order, native size, and native alignment, this didn’t work out of the box.

So did we!

Eventually, we traced this mismatch to the fact that our Python interpreter — which is itself implemented in C — and our little C program for writing vertices to a file, were compiled using different C compilers with different structure padding conventions. This just goes to show that when dealing with binary data you need to be *very* careful if you want your programs to be portable between systems and compilers.

Memory Views

Python has a built-in type called `memoryview` which wraps any existing, underlying collection of bytes and which supports something called the *buffer protocol*. The buffer protocol is implemented at the C level inside the Python interpreter, and isn’t a protocol in the same sense that we use the word when talking about the Python-level *sequence* and *mapping* protocols. In fact, the `memoryview` type *implements* the Python-level *sequence* protocol, allowing us to view the underlying byte buffer as a sequence of Python objects.

Our previous example required that we read the data from the file into a byte array, and translate it with `struct.unpack()` into a tuple of numeric objects, effectively duplicating data. We’re going to change that example now to use `memoryview`s, avoiding the duplication.

We can construct `memoryview` instances by passing any object that supports the buffer protocol C API to the constructor. The only built-in types which support the buffer protocol are `bytes` and `bytearray`. We’ll construct a memory view from the buffer just after our diagnostic print statements, with this line of code:

```
mem = memoryview(buffer)
```

Exploring code at runtime with `code.interact`

To explore the capabilities of `memoryview` we could use a debugger such as PDB to stop the program just after this line, but we'll introduce you to another technique from the Python Standard Library `code` module: `code.interact()`. This function will suspend the program and drop us to the REPL. By passing a reference to the local namespace we can get access to the 'live' variables in our program, including our `memoryview`. With these two changes, our `main()` function now looks like this:

```
def main():
    with open('colors.bin', 'rb') as f:
        buffer = f.read()

    mem = memoryview(buffer)
    code.interact(local=locals())  # Temporary

    vertices = []
    for fields in struct.iter_unpack('@3f3Hxx', buffer):
        vertex = make_colored_vertex(*fields)
        vertices.append(vertex)

    pp(vertices)
```

We use a call to the `locals()` built-in function to get a reference to the current namespace. Now, when we run our program, we get a REPL prompt at which we can access our new `mem` object:

```
$ python3 reader.py
>>> mem
<memory at 0x10214e5c0>
```

The `memoryview` object supports indexing, so retrieving the byte at index 21 and converted to hexadecimal gives us 3f, as we might expect:

```
>>> hex(mem[21])
'0x3f'
```

Being bona-fide sequences, `memoryview` objects support slicing:

```
>>> mem[12:18]
<memory at 0x10214e750>
```

Crucially, `memoryview` slices are *also* `memoryviews`. There's no copying going on here! The zeroth byte of the slice has value e0:

```
>>> hex(mem[12:18][0])
'0xe0'
```

Casting `memoryview` elements

Much in the same way we used the `struct` module to interpret binary data as native types, we can use the `memoryview.cast()` method to do something equivalent[7]. This method accepts uses the same format codes as used by the `struct` module, except only a single element code is permitted. In other words the interpreted items must all be of the same type.

We know that the bytes in the `[12:18]` slice represent three `unsigned short int` values, so by passing 'H' to the `cast()` method we can interpret the values that way:

```
>>> mem[12:18].cast('H')
<memory at 0x10214e688>
```

Notice that this *also* returns a `memoryview`, but this time one that knows the type of its elements:

```
>>> mem[12:18].cast('H')[0]
3040
>>> mem[12:18].cast('H')[1]
34423
>>> mem[12:18].cast('H')[2]
54321
```

Alternatively, we can use the `memoryview.tolist()` method to convert the series of elements to a `list`:

```
>>> mem[12:18].cast('H').tolist()
[3040, 34423, 54321]
```

When you've finished with the interactive session, you can send an end-of-file character just as you normally would to terminate a REPL session with Ctrl-D on Unix or Ctrl-Z on Windows; your program will then continue executing from the first statement after `code.interact()`.

There are other interesting features of `memoryviews` such as the ability to interpret multidimensional C arrays. If you need those facilities we recommend you consult the documentation.

> Before moving on, remove the `code.interact()` line, so our program runs uninterrupted again.

Python classes built on `memoryviews`

Let’s use the ability to slice and cast `memoryviews` to modify our `Vector` and `Color` types to use the `bytes` buffer as the underlying storage.

Our modified `Vector` class now looks like this:

```
class Vector:

    def __init__(self, mem_float32):
        if mem_float32.format not in "fd":
            raise TypeError("Vector: memoryview values must be floating-"
                            "point numbers")
        if len(mem_float32) < 3:
            raise TypeError("Vector: memoryview must contain at least 3 floats")
        self._mem = mem_float32

    @property
    def x(self):
        return self._mem[0]

    @property
    def y(self):
        return self._mem[1]

    @property
    def z(self):
        return self._mem[2]

    def __repr__(self):
        return 'Vector({}, {}, {})'.format(self.x, self.y, self.z)
```

The initializer accepts a `memoryview` which we expect to expose `floats`. We validate this by checking the format code returned by the `memoryview.format` attribute against a string containing ‘f’ and ‘d’, those codes for single- and double-precision floats respectively. We also check that the memory view exposes at least three items; note than when used with a `memoryview` `len()` returns the number of *items* not the number of *bytes*. The only instance attribute now holds a reference to the `memoryview`.

Our old instance attributes are replaced by properies which perform the appropriate lookups in the `memoryview`. Since we can use properties to replace attributes, our `__repr__()` implementation can remain unmodified.

The modified `Color` class works exactly the same way, except now we check that we're wrapping unsigned integer types:

```
class Color:

    def __init__(self, mem_uint16):
        if mem_uint16.format not in "HILQ":
            raise TypeError("Color: memoryview values must be unsigned integers")
        if len(mem_uint16) < 3:
            raise TypeError("Color: memoryview must contain at least 3 integers")
        self._mem = mem_uint16

    @property
    def red(self):
        return self._mem[0]

    @property
    def green(self):
        return self._mem[1]

    @property
    def blue(self):
        return self._mem[2]

    def __repr__(self):
        return 'Color({}, {}, {})'.format(self.red, self.green, self.blue)
```

Our `Vertex` class, which simply combines a `Vector` and `Color`, can remain as before, although our `make_colored_vertex()` factory function needs to be changed to accept a `memoryview` — specifically one that is aligned with the beginning of a `Vertex` structure:

```
def make_colored_vertex(mem_vertex):
    mem_vector = mem_vertex[0:12].cast('f')
    mem_color = mem_vertex[12:18].cast('H')
    return Vertex(Vector(mem_vector),
                  Color(mem_color))
```

The function now slices the vertex `memoryview` into two parts, for the vector and color respectively, and casts each to a typed `memoryview`. These are used to construct the `Vector` and `Color` objects, which are then passed on to the `Vertex` constructor.

Back in our main function after our creation of the `mem` instance, we'll need to rework our main loop. We'll start by declaring a couple of constants describing the size of a `Vertex` structure, and the stride between successive `Vertex` structures, this allows us to take account of the two padding bytes between structures:

```
VERTEX_SIZE = 18
VERTEX_STRIDE = VERTEX_SIZE + 2
```

Next we'll set up a generator expression which yields successive `memoryviews` into whole `Vertex` structures:

```
vertex_mems = (mem[i:i + VERTEX_SIZE] for i in range(0, len(mem), VERTEX_STRIDE))
```

Remember that each slice into `mem` is itself a `memoryview`.

This time, rather than an explicit for-loop to build the list of vertices, we'll use a list comprehension to pass each vertex memory view in turn to `make_colored_vertex()`:

```
vertices = [make_colored_vertex(vertex_mem) for vertex_mem in vertex_mems]
```

Running this program, we can see we get exactly the same results as before, except that now our `Vector` and `Color` objects are backed by the binary data we loaded from the file, with much reduced copying.

Here's the complete program as it now stands:

```
from binascii import hexlify
from pprint import pprint as pp
import struct

class Vector:

    def __init__(self, mem_float32):
        if mem_float32.format not in "fd":
            raise TypeError("Vector: memoryview values must be floating-"
                            "point numbers")
        if len(mem_float32) < 3:
            raise TypeError("Vector: memoryview must contain at least 3 floats")
        self._mem = mem_float32

    @property
    def x(self):
        return self._mem[0]

    @property
    def y(self):
        return self._mem[1]

    @property
    def z(self):
        return self._mem[2]

    def __repr__(self):
        return 'Vector({}, {}, {})'.format(self.x, self.y, self.z)
```

```
class Color:

    def __init__(self, mem_uint16):
        if mem_uint16.format not in "HILQ":
            raise TypeError("Color: memoryview values must be unsigned integers")
        if len(mem_uint16) < 3:
            raise TypeError("Color: memoryview must contain at least 3 integers")
        self._mem = mem_uint16

    @property
    def red(self):
        return self._mem[0]

    @property
    def green(self):
        return self._mem[1]

    @property
    def blue(self):
        return self._mem[2]

    def __repr__(self):
        return 'Color({}, {}, {})'.format(self.red, self.green, self.blue)


class Vertex:

    def __init__(self, vector, color):
        self.vector = vector
        self.color = color

    def __repr__(self):
        return 'Vertex({!r}, {!r})'.format(self.vector, self.color)


def make_colored_vertex(mem_vertex):
    mem_vector = mem_vertex[0:12].cast('f')
    mem_color = mem_vertex[12:18].cast('H')
    return Vertex(Vector(mem_vector),
                  Color(mem_color))


VERTEX_SIZE = 18
VERTEX_STRIDE = VERTEX_SIZE + 2

def main():
    with open('colors.bin', 'rb') as f:
        buffer = f.read()

    mem = memoryview(buffer)

    VERTEX_SIZE = 18
    VERTEX_STRIDE = VERTEX_SIZE + 2

    vertex_mems = (mem[i:i + VERTEX_SIZE]
                   for i in range(0, len(mem), VERTEX_STRIDE))

    vertices = [make_colored_vertex(vertex_mem)
                for vertex_mem in vertex_mems]
```

```
    pp(vertices)


if __name__ == '__main__':
    main()
```

Memory-mapped files

There's still one copy happening though: the transfer of bytes from the file into our `buffer` bytes object. This is not a problem for our trifling 80 bytes, but for very large files this could be prohibitive.

By using an operating system feature called *memory-mapped files* we can use the virtual memory system to make large files *appear* as if they are in memory. Behind the scenes the operating system will load, discard and sync pages of data from the file. The details are operating system dependent, but the pages are typically only 4 kilobytes in size, so this can be memory efficient if you need access to relatively small parts of large files.

Memory-mapped file support in Python is provided by the standard library `mmap` module. This module contains a single class, also called `mmap`, which behaves like a mix between a `bytearray` and a file-like object. The `mmap` instance is created around a file-handle on Windows or a file-descriptor on Unix, either of which can be obtained by calling the `fileno()` method of a regular file object.

In fact, `mmap` instances support the C API *buffer protocol* and so can be used as the argument we pass to our `memoryview` constructor.

Using `mmap` in our implementation

Let's modify our example to do so. All this requires is that we import the `mmap` module at the top of our file:

```
import mmap
```

We then need to modify our `main()` function to retrieve the file handle or descriptor, passing it to the `mmap` class constructor. Like other file-like objects, `mmap`s must be closed when we're done with them. We can either call the `close()` method explicity or, more conveniently, use the `mmap` object as a context manager. We’ll go with the latter. Here’s the resulting `main()` method:

```
def main():
    with open('colors.bin', 'rb') as f:
        with mmap.mmap(f.fileno(), 0, access=mmap.ACCESS_READ) as buffer:

            mem = memoryview(buffer)

            VERTEX_SIZE = 18
            VERTEX_STRIDE = VERTEX_SIZE + 2

            vertex_mems = (mem[i:i + VERTEX_SIZE]
                           for i in range(0, len(mem), VERTEX_STRIDE))

            vertices = [make_colored_vertex(vertex_mem)
                        for vertex_mem in vertex_mems]

            pp(vertices)
```

The only difference here is that `buffer` is now a memory-mapped file rather than `bytes` sequence as it was previously. We’ve avoided reading the file into memory twice — once into the operating system file cache and once more into our own collection — by directly working on the operating system’s view of the file.

Dangling memory references

This “works” after a fashion when we run it, insofar as our `Vertex` objects are created with the memory-mapped file backing store. However, we get a nasty failure when the `mmap` object is closed by the context manager, which tells us that “exported pointers exist”:

```
Traceback (most recent call last):
  File "examples/reader.py", line 108, in <module>
    main()
  File "examples/reader.py", line 100, in main
    pp(vertices)
  BufferError: cannot close exported pointers exist
```

The cause is that at the point the `mmap` object is closed we still have a chain of extant `memoryview` objects which ultimately depend on the `mmap`. A reference counting mechanism in the buffer-protocol has tracked this, and knows that the `mmap` still has `memoryview` instances pointing to it.

There are a couple of approaches we could take here. We could arrange for the `memoryview.release()` method to be called on the `memoryview` objects inside our `Vector` and `Color` instances. This method deregisters the `memoryview` from any underlying buffers and invalidates the `memoryview` so any further operations with it raise a `ValueError`. This would just move the problem though, now we'd have zombie `Vector` and `Color` instances containing invalid `memoryview`s. Better, we think, to respect the constraint in our design that the lifetime of our memory-mapped file-backed objects must be shorter than the lifetime of our memory mapping.

Thinking about our live object-graph, there are two local variables which ultimately hold references to the memory-mapped file: `mem` which is our lowest level `memoryview`, and `vertices` which is the list of `Vertex` objects.

By explicitly removing these name bindings, using two invocations of the `del` statement, we can clean up the `memoryview`s, so the memory map can be torn down safely:

```
del mem
del vertices
```

With this change in place, the main function looks like this:

```
def main():
    with open('colors.bin', 'rb') as f:
        with mmap.mmap(f.fileno(), 0, access=mmap.ACCESS_READ) as buffer:

            mem = memoryview(buffer)

            VERTEX_SIZE = 18
            VERTEX_STRIDE = VERTEX_SIZE + 2

            vertex_mems = (mem[i:i + VERTEX_SIZE]
                           for i in range(0, len(mem), VERTEX_STRIDE))

            vertices = [make_colored_vertex(vertex_mem)
                        for vertex_mem in vertex_mems]

            del vertices
            del mem
```

And our program runs flawlessly, with minimal memory usage, even for huge sets of vertex objects:

```
[Vertex(Vector(3323.176025390625, 6562.23095703125, 9351.2314453125),
        Color(3040, 34423, 54321)),
 Vertex(Vector(7623.98193359375, 2542.23095703125, 9823.12109375),
        Color(32736, 5342, 2321)),
 Vertex(Vector(6729.86181640625, 2347.2119140625, 3421.322021484375),
        Color(45263, 36291, 36701)),
```

```
 Vertex(Vector(6352.12109375, 3432.111083984375, 9763.232421875),
        Color(56222, 36612, 11214))]
```

Summary

Let’s summarize what we’ve covered in this chapter:

- Our review of Python’s bitwise operators including exlusive-or and complement led us down the path of understanding how Python internally represents and externally *presents* arbitrary precision integer values at the bit level.
- To print bit patterns for values which equate to negative numbers you can sign-extend the result by using bitwise-and with a binary value consisting of consecutive 1 bits.
- We reviewed the constructors and some key methods of the *immutable* binary sequence type `bytes` and the *mutable* binary sequence type `bytearray`.
- We showed how to interpret a series of bytes as native or C language types such as `short unsigned integer`, translating them into compatible Python types, using the `struct` module.
- We demonstrated how to display byte streams in an easily readable hexadecimal format using the `hexlify` module in conjuction with some simple formatting expressions.
- We explained that a detailed understanding of how C compilers align data structures on certain byte or word boundaries can be crucial to correctly reading data file produced by C or C++ programs.
- We introduced `memoryview` for obtaining copy-free views, slices, and type casts from underlying byte sequences.
- We touched on the use of the `interact()` function from the standard library `code` module, which allows us to drop to the REPL at any point in a program, and resume execution later.
- We showed that memory-mapped files implemented with the `mmap` module implement the buffer-protocol, and so can be used as the basis for `memoryview` objects, further reducing the amount of data copying when reading files.

There’s a lot more to be learned about low-level byte-oriented programming in Python, including writeable `memoryviews`, shared memory with `mmap`, and interfacing to native C and C++ code. We can’t possibly cover all of that in *this* book, but we’ve shown you the starting points for your own explorations.

Chapter 3 - Object Internals and Custom Attributes

In this chapter we'll show how objects are represented internally in Python, as a dictionary called `__dict__`. We'll also demonstrate how to directly query and manipulate objects through the internal `__dict__`. We'll then use this knowledge to implement custom attribute access, by overriding the special `__getattr__()`, `__getattrribute__()`, `__setattr__()`, and `__delattr__()` methods. We'll round off by seeing how to make objects more memory efficient by defining `__slots__`.

How are Python objects stored?

We'll start with a simple class to represent two-dimensional vectors, called simply `Vector`:

```
class Vector:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return "{}({}, {})".format(
            type(self).__name__, self.x, self.y)
```

Let's instantiate an object of that class in the REPL:

```
>>> v = Vector(5, 3)
>>> v
Vector(5, 3)
>>> dir(v)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'x', 'y']
```

In the list returned by `dir()` we see the two named attributes `x` and `y` along with many of Python's special attributes, quite a few of which we've explained previously in *The Python Apprentice* and *The Python Journeyman*.

One attribute in particular is of interest to us today, and that is `__dict__`. Let's see what it is:

```
>>> v.__dict__
{'x': 5, 'y': 3}
```

As its name indicates, `__dict__` is indeed a dictionary, one which contains the names of our object's attributes as keys, and the values of our object's attributes as, well, values. Here's further proof, if any were needed, that `__dict__` is a Python dictionary:

```
>>> type(v.__dict__)
<class 'dict'>
```

We can retrieve attributes values directly from `__dict__`:

```
>>> v.__dict__['x']
5
```

Modify values:

```
>>> v.__dict__['x'] = 17
>>> v.x
17
```

and even remove them!:

```
>>> del v.__dict__['x']
>>> v.x
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'Vector' object has no attribute 'x'
```

As you might by now suspect, we can test for their existence:

```
>>> 'x' in v.__dict__
False
>>> 'y' in v.__dict__
True
```

and insert new attributes into the dictionary:

```
>>> v.__dict__['z'] = 13
>>> v.z
13
```

Although all of these direct queries and manipulations of `__dict__` are possible, for the most part you should prefer to use the built-in functions `getattr()`, `hasattr()`, `delattr()` and `setattr()`:

```
>>> getattr(v, 'y')
3
>>> hasattr(v, 'x')
False
>>> delattr(v, 'z')
>>> setattr(v, 'x', 9)
```

Direct access to `__dict__` does have legitimate uses though, so it's essential to be aware of its existence, and how and when to use it for advanced Python programming.

Our `Vector` class, like most vector classes, has hardwired attributes called `x` and `y` to store the two components of the vector.

Many problems though, require us to deal with vectors in different coordinate systems within the same code. Or perhaps it's just convenient to use a different labelling scheme, such a `u` and `v` instead of `x` and `y` in a particular context. Let's see what we can come up with:

```
class Vector:

    def __init__(self, **coords):
        self.__dict__.update(coords)

    def __repr__(self):
        return "{}({})".format(
            type(self).__name__,
            ', '.join("{k}={v}".format(
                k=k,
                v=self.__dict__[k])
                for k in sorted(self.__dict__.keys())))
```

In this code, we accept a keyword argument, which is received into the `coords` dictionary, the contents of which we use to update the entries in `__dict__`. Remember that dictionaries are unordered[8], so there's no way to ensure that the coordinates are stored in the order they are specified. Our `__repr__()` implementation must iterate over the dictionary, sorting by key, for convenience.

This allows us to provide arbitrary coordinate names:

```
>>> v = Vector(p=3, q=7)
>>> v
Vector(p=3, q=7)
>>> dir(v)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
 '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', '__weakref__', 'p', 'q']
```

This is all very well, but our coordinates are now essentially “public” attributes of our vector objects. What if we want our `Vector` class to be an immutable value type, so values provided to the constructor can’t be subsequently changed? Ordinarily, we would do this by prefixing our attributes with an underscore to signal that they are implementation details, and then provide a property with only a getter, to prevent modification. In this case though, we don’t know the attribute names in advance, so we can’t declare a property getter which must be named at *class* definition time, not at *object* instantiation time. We’ll show how to work around this using the special `__getattr__` method, but first, let’s change our `__init__()` method to store data in “private” attributes, and our `__repr__()` to report them correctly:

```
class Vector:

    def __init__(self, **coords):
        private_coords = {'_' + k: v for k, v in coords.items()}
        self.__dict__.update(private_coords)

    def __repr__(self):
        return "{}({})".format(
            type(self).__name__,
            ', '.join("{k}={v}".format(
                k=k[1:],
                v=self.__dict__[k])
                for k in sorted(self.__dict__.keys())))
```

We can construct and represent instances as before:

```
>>> v = Vector(p=9, q=3)
>>> v
Vector(p=9, q=3)
```

But now the attributes are stored in “private” attributes called _p and _q:

```
>>> dir(v)
['__class__', '__delattr__', '__dict__', '__dir__',
 '__doc__',   '__eq__', '__format__', '__ge__', '__getattribute__',
 '__gt__',   '__hash__', '__init__', '__le__', '__lt__', '__module__',
```

```
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', '_p', '_q']
```

So we can no longer access `p` directly, because it doesn't exist:

```
>>> v.p
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'Vector' object has no attribute 'p'
```

What we'd like to do is to fake the existence of `p` for read access, and to do that, we need to intercept attribute access before the `AttributeError` is raised. To do that, we can override `__getattr__()`. Notice that there are two very similarly named special methods: `__getattr__()` and `__getattribute__()`. The former is only called when regular attribute lookup fails. The latter is called for *all* attribute access irrespective of whether an attribute of the requested name exists or not. For our case, we'll be using `__getattr__()` to intercept lookup failures.

Here's our `Vector` definition with `__getattr__()` added. We've just added a simple stub that prints the attribute name:

```
class Vector:

    def __init__(self, **coords):
        private_coords = {'_' + k: v for k, v in coords.items()}
        self.__dict__.update(private_coords)

    def __getattr__(self, name):
        print("name =", name)

    def __repr__(self):
        return "{}({})".format(
            type(self).__name__,
            ', '.join("{k}={v}".format(k=k[1:], v=self.__dict__[k])
                for k in sorted(self.__dict__.keys())))
```

When we request non-existent attributes, the name of the attribute is now printed:

```
>>> v = Vector(p=3, q=9)
>>> v.p
name = p
>>> v.q
name = q
```

But when we request an attribute that exists, we simply get the attribute value, indicating that `__getattr__` isn't being called:

```
>>> v._q
9
```

Now we can modify `__getattr__()` to prepend the underscore to name, and retrieve the attribute for us:

```
class Vector:

    def __init__(self, **coords):
        private_coords = {'_' + k: v for k, v in coords.items()}
        self.__dict__.update(private_coords)

    def __getattr__(self, name):
        private_name = '_' + name
        return getattr(self, private_name)

    def __repr__(self):
        return "{}({})".format(
        type(self).__name__,
        ', '.join("{k}={v}".format(k=k[1:], v=self.__dict__[k])
            for k in sorted(self.__dict__.keys())))
```

At first sight, this appears to work just fine:

```
>>> from vector_05 import *
>>> v = Vector(p=5, q=10)
>>> v.p
5
>>> v.q
10
```

but there are some serious problems lurking here. The first, is that we can still *assign* to `p` and `q`:

```
>>> v.p = 13
```

Remember, there wasn't really an attribute called `p`, but Python has no qualms about creating it for us on demand. Worse, because we have unwittingly brought `p` into existence, `__getattr__()` is not longer invoked for requests of `p`, even though our hidden attribute `_p` is still there behind the scenes, with a different value:

```
>>> v.p
13
>>> v._p
5
```

We can prevent this, by overriding `__setattr__()` too, to intercept attempts to write to our attributes:

```
class Vector:

    def __init__(self, **coords):
        private_coords = {'_' + k: v for k, v in coords.items()}
        self.__dict__.update(private_coords)

    def __getattr__(self, name):
        private_name = '_' + name
        return getattr(self, private_name)

    def __setattr__(self, name, value):
        raise AttributeError("Can't set attribute {!r}".format(name))

    def __repr__(self):
        return "{}({})".format(
            type(self).__name__,
            ', '.join("{k}={v}".format(k=k[1:], v=self.__dict__[k])
                for k in sorted(self.__dict__.keys())))
```

This successfully prevents writing to any attribute:

```
>>> v = Vector(p=4, q=8)
>>> v.p = 7
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "examples/vector/vector.py", line 16, in __setattr__
    raise AttributeError("Can't set attribute {!r}".format(name))
AttributeError: Can't set attribute 'p'
```

We'll return to `__setattr__` shortly, but first, we need to demonstrate another problem with our `__getattr__()`. Although `__getattr__()` works fine with attributes we've carefully faked:

```
>>> v.p
4
```

Look what happens when we try to access an attribute for which there is no faked support:

```
>>> v.x
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "examples/vector/vector.py", line 13, in __getattr__
    return getattr(self, private_name)
  File "examples/vector/vector.py", line 13, in __getattr__
    return getattr(self, private_name)
  File "examples/vector/vector.py", line 13, in __getattr__
    return getattr(self, private_name)
...
  File "examples/vector/vector.py", line 13, in __getattr__
```

```
    return getattr(self, private_name)
RuntimeError: maximum recursion depth exceeded while calling a Python object
```

This happens because our request for attribute `x` causes `__getattr__()` to look for an attribute _x, which doesn't exist, which invokes `__getattr__()` again, to lookup attribute __x, which doesn't exist. And so on recursively, until the Python interpreter exceeds its maximum recursion depth and raises a `RuntimeError`.

To prevent this happening, you might be tempted to check for existence of the private attribute using `hasattr()`, like this:

```
class Vector:

    def __init__(self, **coords):
        private_coords = {'_' + k: v for k, v in coords.items()}
        self.__dict__.update(private_coords)

    def __getattr__(self, name):
        private_name = '_' + name
        if not hasattr(self, private_name):
            raise AttributeError('{!r} object has no attribute {!r}'.format(
                self.__class__, name))
        return getattr(self, private_name)

    def __setattr__(self, name, value):
        raise AttributeError("Can't set attribute {!r}".format(name))

    def __repr__(self):
        return "{}({})".format(
            type(self).__name__,
            ', '.join("{k}={v}".format(k=k[1:], v=self.__dict__[k])
                for k in sorted(self.__dict__.keys())))
```

Unfortunately, this doesn't work either, since it turns out that `hasattr()` also ultimately calls `__getattr__()` in search of the attribute! What we need to do, is directly check for the presence of our attribute in `__dict__`:

```
class Vector:

    def __init__(self, **coords):
        private_coords = {'_' + k: v for k, v in coords.items()}
        self.__dict__.update(private_coords)

    def __getattr__(self, name):
        private_name = '_' + name
        if private_name not in self.__dict__:
            raise AttributeError('{!r} object has no attribute {!r}'.format(
                type(self).__name__, name))
        return getattr(self, private_name)

    def __setattr__(self, name, value):
```

```
        raise AttributeError("Can't set attribute {!r}".format(name))

    def __repr__(self):
        return "{}({})".format(
            type(self).__name__,
            ', '.join("{k}={v}".format(k=k[1:], v=self.__dict__[k])
                for k in sorted(self.__dict__.keys())))
```

This now works as we would wish:

```
>>> v = Vector(p=9, q=14)
>>> v
Vector(p=9, q=14)
>>> v.p
9
>>> v.x
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "examples/vector/vector.py", line 14, in __getattr__
    raise AttributeError('{!r} object has no attribute {!r}'.format(self.__class__, na
AttributeError: <class 'Vector'> object has no attribute 'x'
```

In fact, attribute lookup in Python follows a pretty complex procedure, so instead of invoking that procedure again by calling `getattr()`, we can just directly return the attribute value from `__dict__`. This also enables us to switch to easier-to-ask-for-forgiveness-than permission (EAFP) rather than look-before-you-leap (LBYL) style:

```
class Vector:

    def __init__(self, **coords):
        private_coords = {'_' + k: v for k, v in coords.items()}
        self.__dict__.update(private_coords)

    def __getattr__(self, name):
        private_name = '_' + name
        try:
            return self.__dict__[private_name]
        except KeyError:
            raise AttributeError('{!r} object has no attribute {!r}'.format(
                type(self).__name__, name))

    def __setattr__(self, name, value):
        raise AttributeError("Can't set attribute {!r}".format(name))

    def __repr__(self):
        return "{}({})".format(
            type(self).__name__,
            ', '.join("{k}={v}".format(k=k[1:], v=self.__dict__[k])
                for k in sorted(self.__dict__.keys())))
```

Giving:

```
>>> v = Vector(p=1, q=2)
>>> v
Vector(p=1, q=2)
>>> v.p
1
>>> v.x
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "examples/vector/vector.py", line 16, in __getattr__
    raise AttributeError('{!r} object has no attribute {!r}'.format(type(self).__name_
name))
AttributeError: 'Vector' object has no attribute 'x'
```

Of course, the EAFP version has the same behaviour as the LBYL version.

Customizing attribute deletion

Deleting attributes is something that is very rarely seen in Python, although it is possible. Our `Vector` class also allows us to remove attributes by calling by calling `delattr()`:

```
>>> v
Vector(p=1, q=2)
>>>
>>> delattr(v, '_p')
```

or even:

```
>>> del v._q
>>>
>>> v
Vector()
```

Although a client is unlikely to attempt this inadvertently, we can prevent it by overriding `__delattr__()`:

```
class Vector:

    def __init__(self, **coords):
        private_coords = {'_' + k: v for k, v in coords.items()}
        self.__dict__.update(private_coords)

    def __getattr__(self, name):
        private_name = '_' + name
        try:
            return self.__dict__[private_name]
        except KeyError:
            raise AttributeError('{!r} object has no attribute {!r}'.format(
                type(self).__name__, name))

    def __setattr__(self, name, value):
        raise AttributeError("Can't set attribute {!r}".format(name))
```

```
    def __delattr__(self, name):
        raise AttributeError("Can't delete attribute {!r}".format(name))

    def __repr__(self):
        return "{}({})".format(
            type(self).__name__,
            ', '.join("{k}={v}".format(k=k[1:], v=self.__dict__[k])
                for k in sorted(self.__dict__.keys())))
```

Here’s how that code behaves when trying do delete either public or private attributes:

```
>>> v = Vector(p=9, q=12)
>>>
>>> v
Vector(p=9, q=12)
>>>
>>> del v.q
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example/vector/vector.py", line 19, in __delattr__
    raise AttributeError("Can't delete attribute {!r}".format(name))
AttributeError: Can't delete attribute 'q'
>>> del v._q
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example/vector/vector.py", line 19, in __delattr__
    raise AttributeError("Can't delete attribute {!r}".format(name))
AttributeError: Can't delete attribute '_q'
```

Customising attribute storage

There’s no requirement for us to store attributes directly in `__dict__`. Here’s an example of a subclass of `Vector` that stores a *mutable* red-green-blue color along with the immutable `Vector` components:

```
class ColoredVector(Vector):

    COLOR_INDEXES = ('red', 'green', 'blue')

    def __init__(self, red, green, blue, **coords):
        super().__init__(**coords)
        self.__dict__['color'] = [red, green, blue]

    def __getattr__(self, name):
        try:
            channel = ColoredVector.COLOR_INDEXES.index(name)
        except ValueError:
            return super().__getattr__(name)
        else:
            return self.__dict__['color'][channel]

    def __setattr__(self, name, value):
        try:
```

```
            channel = ColoredVector.COLOR_INDEXES.index(name)
        except ValueError:
            super().__setattr__(name, value)
        else:
            self.__dict__['color'][channel] = value
```

Internally, we store the red-green-blue channels in a `list`. We override both `__getattr__()` and `__setattr__()` to provide read and write access to the color channels, being careful to forward requests to the superclass when necessary. So long as we are careful to only ever access our `color` attribute by accessing `__dict__` directly, everything seems to work well:

```
>>> cv = ColoredVector(red=23, green=44, blue=238, p=9, q=14)
>>> cv.red
23
>>> cv.green
44
>>> cv.blue
238
>>> cv.p
9
>>> cv.q
14
>>> dir(cv)
['COLOR_INDEXES', '__class__', '__delattr__', '__dict__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattr__',
 '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
 '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '__weakref__', '_p', '_q', 'color']
>>> cv.red = 50
>>> cv.red
50
>>> cv.p = 12
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "examples/vector/vector.py", line 53, in __setattr__
    super().__setattr__(name, value)
  File "examples/vector/vector.py", line 20, in __setattr__
    raise AttributeError("Can't set attribute {!r}".format(name))
AttributeError: Can't set attribute 'p'
```

There’s a gremlin lurking here though: Our `__repr__()` implementation in the base class makes an assumption which is no longer valid. It assumes all attributes are prefixed with an underscore internally, and it doesn’t know about `color`:

```
>>> cv
ColoredVector(p=9, q=14, olor=[50, 44, 238])
```

This is surprisingly hard to fix elegantly, without the derived `ColoredVector` class knowing too much about implementation details of the `Vector` base class; we believe it should be possible to derive from a class without knowing how it works. The base class `__repr__()` makes assumptions about the contents of its `__dict__` which it cannot reasonably expect to be respected by subclasses. As an exercise, we recommend changing `Vector` so it stores its components in a dedicated dictionary separate from `__dict__`, although of course, this dictionary itself will need to be stored in `__dict__`. Here is a fix which "works" by duplicating and modifying some of the logic in the base class:

```
def __repr__(self):
    keys = set(self.__dict__.keys())
    keys.discard('color')
    coords = ', '.join(
        "{k}={v}".format(k=k[1:], v=self.__dict__[k])
        for k in sorted(keys))

    return "{cls}(red={red}, green={green}, blue={blue}, {coords})".format(
        cls=type(self).__name__,
        red=self.red,
        green=self.green,
        blue=self.blue,
        coords=coords)
```

This method override works by removing the attribute name 'color' from the list of keys before using the same logic as the superclass to produce the string of sorted co-ordinates. The color channel values are accessed in the normal way, which will invoke `__getattr__()`.

It's worth bearing in mind that this example demonstrates the awkwardness of inheriting from classes which were not deliberately designed as base classes. Our code serves it's purpose in demonstrating customised attribute access, but we couldn't recommend such use of inheritance in production code.

Using `vars()` to access `__dict__`

Do you remember the `vars()` built-in function? Without an argument it returns a dictionary containing the current namespace, as so acts just like `locals()`. However, if we supply an argument it returns the `__dict__` attribute of the argument, so instead of writing:

```
obj.__dict__
```

we can write:

```
vars(obj)
```

Arguably, this is more *Pythonic* than accessing `__dict__` directly, for much the same reason that calling:

```
len(collection)
```

is definitely more idiomatic than calling:

```
collection.__len__()
```

That said, the length returned by `len()` is always immutable, whereas `__dict__` is a mutable dictionary. In our opinion it us much clearer that the internal state of an object is being modified when we directly modify the `__dict__` attribute, like this:

```
self.__dict__['color'] = [red, green, blue]
```

than we we go via `vars()`:

```
vars(self)['color'] = [red, green, blue]
```

Whichever you use – and we don't feel strongly either way – you should be aware of this use of `vars()` with an argument.

Overriding `__getattribute__()`

Recall that `__getattr__()` is called only in cases when "normal attribute lookup fails"; it is our last chance for us to intervene before the Python runtime raises an `AttributeError`. But what if we want to intercept *all* attribute access? In that case, we can override `__getattribute__()`. I use the term "override" advisedly, because it is the implementation of `__getattribute__()` in the ultimate base class `object` that is responsible for the normal lookup behaviour, including calling `__getattr__()`. This level of control is seldom required, and you should always consider whether `__getattr__()` is sufficient for your needs. That said, `__getattribute__()` does have its uses.

Consider this class, which implements a `LoggingProxy` which logs every attribute retrieval made against the target object, supplied to the constructor:

```
class LoggingProxy:

    def __init__(self, target):
        super().__setattr__('target', target)

    def __getattribute__(self, name):
        target = super().__getattribute__('target')

        try:
            value = getattr(target, name)
        except AttributeError as e:
            raise AttributeError("{} could not forward request {} to {}"
                .format(
                    super().__getattribute__('__class__').__name__,
                    name,
                    target)
                ) from e
        print("Retrieved attribute {!r} = {!r} from {!r}"
              .format(name, value, target))
        return value
```

Since `__getattribute__()` intercepts all attribute attribute access through the ‘dot’ operator, we must be very careful to never access attributes of the `LoggingProxy` through the dot. In the initialiser we use the `__setattr__()` implementation inherited from the `object` superclass to do our work. Inside `__getattribute__()` itself, we retrieve the attribute called “target” using a call to the superclass implementation of `__getattribute__()`, which implements the default lookup behaviour. Once we have a reference to the target class, we delegate to the `getattr()` built-in function. If attribute lookup on the target fails, we then raise an `AttributeError` with an informative message. Note how careful we must be even to return our object’s `__class__`!

If attribute retrieval was successful, we report as much before returning the attribute value.

Let’s see it in action. If you’re following along at the keyboard, we recommend using the regular Python REPL in a terminal, rather than using the more advanced REPLs such as `ipython` or the PyCharm REPL, because these advanced REPLs performs a great many invocations of `__getattribute__()` themselves, to support automatic code-completion:

```
>>> cv = ColoredVector(red=23, green=44, blue=238, p=9, q=14)
>>> cw = LoggingProxy(cv)
>>> cw.p
Retrieved attribute 'p' = 9 from ColoredVector(p=9, q=14, olor=[23, 44, 238])
9
>>> cw.red
Retrieved attribute 'red' = 23 from ColoredVector(p=9, q=14, olor=[23, 44, 238])
23
```

So far, so good. But what happens when we *write* to an attribute through the proxy? In this example both writes *appear* to be accepted without error, although only one of them should be:

```
>>> cw.p = 19
>>> cw.red = 5
```

Although in fact, neither of the writes succeeded:

```
>>> cw.p
Retrieved attribute 'p' = 9 from ColoredVector(p=9, q=14, olor=[23, 44, 238])
9
>>> cw.red
Retrieved attribute 'red' = 23 from ColoredVector(p=9, q=14, olor=[23, 44, 238])
23
```

What's happening here is that our attribute *writes* to the `cw` proxy are invoking `__setattr__()` on the `object` base class, which is actually creating new attributes in the `LoggingProxy` instance `__dict__`. However, *reads* through the proxy correctly bypass this `__dict__` and are redirected to the target. In effect, the proxy `__dict__` has become write-only!

The solution to this is to override `__setattr__()` on the `LoggingProxy` too:

```
class LoggingProxy:

    def __init__(self, target):
        super().__setattr__('target', target)

    def __getattribute__(self, name):
        target = super().__getattribute__('target')

        try:
            value = getattr(target, name)
        except AttributeError as e:
            raise AttributeError("{} could not forward request {} to {}"
                .format(
                    super().__getattribute__('__class__').__name__,
                    name,
                    target)
                ) from e
        print("Retrieved attribute {!r} = {!r} from {!r}"
```

```
              .format(name, value, target))
        return value

    def __setattr__(self, name, value):
        target = super().__getattribute__('target')

        try:
            setattr(target, name, value)
        except AttributeError as e:
            raise AttributeError("{} could not forward request {} to {}"
                .format(
                    super().__getattribute__('__class__').__name__,
                    name,
                    target)
                )
        print("Set attribute {!r} = {!r} on {!r}"
              .format(name, value, target))
```

With `__setattr__()` in place we can successfully write through the logging proxy to modify mutable attributes – here we update `red` to 55, and attempted writes to immutable attributes, such as `p` are rejected as planned:

```
>>> from vector import *
>>> from loggingproxy import *
>>>
>>> cw = ColoredVector(red=23, green=44, blue=238, p=9, q=14)
>>> cw
ColoredVector(red=23, green=44, blue=238, p=9, q=14)
>>>
>>> cw.red = 55
>>>
>>> cw
ColoredVector(red=55, green=44, blue=238, p=9, q=14)
>>>
>>> cw.p = 19
Traceback (most recent call last):
  File "examples/vector/vector.py", line 48, in __setattr__
    channel = ColoredVector.COLOR_INDEXES.index(name)
ValueError: tuple.index(x): x not in tuple

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "examples/vector/vector.py", line 50, in __setattr__
    super().__setattr__(name, value)
  File "examples/vector/vector.py", line 16, in __setattr__
    raise AttributeError("Can't set attribute {!r}".format(name))
AttributeError: Can't set attribute 'p'
>>>
```

Special methods which bypass `__getattribute__()`

It’s important to realise that `__getattribute__()` only intercepts attribute lookup through the dot operator. Let’s create a `ColoredVector` called `cv` and a `LoggingProxy` for it called `cw`:

```
>>> from vector import *
>>> from loggingproxy import *
>>>
>>> cv = ColoredVector(red=39, green=22, blue=89, s=45, t=12)
>>> cv
ColoredVector(red=39, green=22, blue=89, s=45, t=12)
>>>
>>> cw = LoggingProxy(cv)
>>> cw
<loggingproxy.LoggingProxy object at 0x101976ba8>
```

If we call the `__repr__()` method directly, the call is routed via the proxy and is dispatched successfully:

```
>>> cw.__repr__()
Retrieved attribute '__repr__' = <bound method ColoredVector.__repr__ of ColoredVector(
d=39, green=22, blue=89, s=45, t=12)> from ColoredVector(red=39, green=22, blue=89, s=4
 t=12)
'ColoredVector(red=39, green=22, blue=89, s=45, t=12)'
```

However, if we request the `repr()` of `cw` in the conventional manner, using the built-in function, the `__getattribute__()` function of our `LoggingProxy` is not invoked, the call is not forwarded to the `ColouredVector`, and instead we get the default `repr` for `LoggingProxy`:

```
>>> repr(cw)
'<loggingproxy.LoggingProxy object at 0x101976ba8>'
```

This shows that `__getattribute__()` can only be used to intercept special method calls when the special method is retrieved *directly* – which is something we don't normally do. Normal access to facilities provided by special methods is through built-in functions such as `len()`, `iter()`, `repr()` and so on. These all bypass the `__getattribute()__` override for performance reasons.

What this means in practice, is that if you want to write a proxy object such as `LoggingProxy` which transparently proxies an object including its `repr` or other special methods, it's up to you to provide an implementation of `__repr__()` that forwards the call appropriately:

```
def __repr__(self):
    target = super().__getattribute__('target')
    repr_callable = getattr(target, '__repr__')
    return repr_callable()
```

This now works when called via the built-in `repr()` function:

```
>>> from vector import *
>>> from loggingproxy import *
>>>
>>> cv = ColoredVector(red=39, green=22, blue=89, s=45, t=12)
>>> cv
ColoredVector(red=39, green=22, blue=89, s=45, t=12)
>>>
>>> cw = LoggingProxy(cv)
>>>
>>> repr(cw)
'ColoredVector(red=39, green=22, blue=89, s=45, t=12)'
```

The same goes for the other special methods.

Where are the methods?

One question you may have is “Where are the methods?”. Why is it, that when we introspect the `__dict__` of an object, we see only attributes and not methods, but that as we have just seen with `__repr__` we can retrieve methods using `getattr()`. So, where *are* the methods?

The answer is that methods are attributes of another object, the class object associated with our instance. As we already know, we can get to the class object via the `__class__` attribute, and sure enough it too has a `__dict__` attribute which contains references to the callable objects which are the manifestations of the methods of our class.

Returning briefly to our `Vector` example:

```
>>> v = Vector(x=3, y=7)
>>> v.__dict__
{'_y': 7, '_x': 3}
>>> v.__class__
<class 'vector_09.Vector'>
>>> v.__class__.__dict__
mappingproxy({
'__delattr__': <function Vector.__delattr__ at 0x103013400>,
'__module__': 'vector_09',
'__setattr__': <function Vector.__setattr__ at 0x1030130d0>,
'__repr__': <function Vector.__repr__ at 0x103013488>,
'__init__': <function Vector.__init__ at 0x1030131e0>,
'__dict__': <attribute '__dict__' of 'Vector' objects>,
'__getattr__': <function Vector.__getattr__ at 0x103013158>,
'__weakref__': <attribute '__weakref__' of 'Vector' objects>,
'__doc__': None})
```

Of course we can retrieve the callable and pass our instance to it:

```
>>> v.__class__.__dict__['__repr__'](v)
'Vector(x=3, y=7)'
```

It's well worth spending some experimental time on your own poking around with these special attributes to get a good sense of how the Python object model hangs together.

It's worth noting that the `__dict__` attribute of a *class* object is not a regular `dict`, but is instead of type `mappingproxy`, a special mapping type used internally in Python, which does not support item assignment:

```
>>> v.__class__.__dict__['a_vector_class_attribute'] = 5
TypeError: 'mappingproxy' object does not support item assignment
```

To add an attribute to a *class* you must use the `setattr()` function:

```
>>> setattr(v.__class__, 'a_vector_class_attribute', 5)
>>> Vector.a_vector_class_attribute
5
```

The machinery of `setattr()` knows how to insert attributes into the class dictionary:

```
>>> v.__class__.__dict__
mappingproxy({'__weakref__': <attribute '__weakref__' of 'Vector' objects>,
'__setattr__': <function Vector.__setattr__ at 0x101a5fa60>,
'a_vector_class_attribute': 5, '__doc__': None, '__repr__':
<function Vector.__repr__ at 0x101a5fb70>, '__init__':
<function Vector.__init__ at 0x101a5f620>, '__getattr__':
<function Vector.__getattr__ at 0x101a5f6a8>, '__module__':
'vector', '__dict__': <attribute '__dict__' of 'Vector' objects>,
'__delattr__': <function Vector.__delattr__ at 0x101a5fae8>})
```

Slots

We'll finish off this part of the course with a brief look at a mechanism in Python for reducing memory use: slots. As we've seen, each and every object stores its attributes in a dictionary. Even an empty Python dictionary is quite a hefty object, weighing in at 288 bytes:

```
>>> d = {}
>>> import sys
>>> sys.getsizeof(d)
288
```

If you have thousands or millions of objects this quickly adds up, causing your programs to need megabytes or gigabytes of memory. Given contemporary computer architectures this tends to lead to reduced performance as CPU caches can hold relatively few objects.

Techniques to solve the high memory usage of Python programs can get pretty involved, such as implementing Python objects in lower-level languages such as C or C++, but fortunately Python provides the slots mechanism which can provide some big wins for low effort, with tradeoffs that will be acceptable in most cases.

Let's take a look! Consider the following class to describe the type of electronic component called a resistor:

```
class Resistor:

    def __init__(self, resistance_ohms, tolerance_percent, power_watts):
        self.resistance_ohms = resistance_ohms
        self.tolerance_percent = tolerance_percent
        self.power_watts = power_watts
```

It's difficult to determine the size in memory of Python objects, but with care, we can use the `getsizeof()` function in the `sys` module. To get the size of an instance of `Resistor`, we need to account for the size of the `Resistor` object itself, and the size of its `__dict__`:

```
>>> from resistor import *
>>> r10 = Resistor(10, 5, 0.25)
>>>
>>> import sys
>>> sys.getsizeof(r10) + sys.getsizeof(r10.__dict__)
152
```

Python objects being the highly-dynamic dictionary-based objects they are we can add attributes to them a runtime, and see this reflected as an increased size:

```
>>> r10.cost_dollars = 0.02
>>> sys.getsizeof(r10) + sys.getsizeof(r10.__dict__)
248
```

This a quite a big object – especially when you consider that the equivalent `struct` in the C programming language would weight in at no more than 64 bytes with very generous precision on the number types.

Let's see if we can improve on this using slots. To use slots we must declare a class attribute called `__slots__` to which we assign a sequence of strings containing the fixed names of the attributes we want all instances of the class to contain:

```
class Resistor:

    __slots__ = ['resistance_ohms', 'tolerance_percent', 'power_watts']

    def __init__(self, resistance_ohms, tolerance_percent, power_watts):
        self.resistance_ohms = resistance_ohms
        self.tolerance_percent = tolerance_percent
        self.power_watts = power_watts
```

Now let's look at the space performance of this new class. We can instantiate `Resistor` just as before:

```
>>> from resistor import *
>>> r10 = Resistor(10, 5, 0.25)
```

And retrieve it's attributes in exactly the same way:

```
>>> r10.tolerance_percent
5
>>> r10.power_watts
0.25
```

However, it's size is much reduced, from 152 bytes down to 64 bytes, less than half the size:

```
>>> import sys
>>> sys.getsizeof(r10)
64
```

There's always a tradeoff though, as we can no longer dynamically add attributes to instances of `Resistor`:

```
>>> r10.cost_dollars = 0.02
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'Resistor' object has no attribute 'cost_dollars'
```

This is because the internal structure of `Resistor` no longer contains a `dict`:

```
>>> r10.__dict__
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'Resistor' object has no attribute '__dict__'
```

For most applications, slots won't be required, and you shouldn't use them unless measurements indicate that they may help, as slots can interact with other Python features and diagnostic tools in surprising ways. In an ideal world, slots wouldn't be necessary and in our view they're quite an ugly language feature, but at the same time we've worked on applications where the simple addition of a `__slots__` attribute has made the difference between the pleasure of programming in Python, and the pain of programming in a lower-level, but more efficient language. Use wisely!

Summary

Let's summarise what we've covered in this chapter:

- We discovered that Python objects store their attributes internally within a dictionary called `__dict__` which maps attribute names to attribute values.
- We showed that instance attributes can be created, retrieved, updated and deleted by direct manipulation of `__dict__`.
- We showed how any failure to retrieve an attribute by normal means causes the `__getattr__()` special method to be invoked. The implementation of `__getattr__()` can use arbitrary logic to 'fake' the existence of attributes programatically.
- Similarly, assignment to, and deletion of, attributes can be customised by overriding `__setattr__()` and `__delattr__()`.
- Calls to the `hasattr()` built-in function may also invoke `__getattr__()` so `__getattr__()` implementations need to be particularly careful to avoid non-terminating recursion.
- Occasionally, it's necessarily to customise *all* attribute lookup – even for regular attributes. In these cases the default lookup machinery in the special `__getattribute__()` of the `object` base class may be overridden, taking care to delegate to the base class implementation as necessary, with a call via `super()`.
- Method callables are stored in the `__class__.__dict__` dictionary.
- Slots are a quick way to make Python objects more memory efficient at the cost of being less dynamic.

Chapter 4 - Descriptors

In this chapter we'll investigate a feature of Python you've been using — perhaps unknowingly — called *descriptors*. Descriptors are the mechanism used to implement properties in Python. We covered properties thoroughly in *The Python Journeyman*, but we'll start here with a short review.

In Chapter 4 of *The Python Journeyman* we showed how to create properties using the property decorator. In this chapter we'll dig deeper and show how to create properties in the raw using the `property()` constructor. Then we'll show how to create a specialised property by a defining a custom descriptor which implements the descriptor protocol. We'll round off by demonstrating that there are two categories of descriptor — data descriptors and non-data descriptors — and we'll show how these interact with Python's somewhat complicated attribute lookup rules.

A review of properties

As promised, we'll start with a very brief review of properties, our entry point into the world of descriptors. To explain descriptors, we'll be building a simple class to model planets, focusing on particular physical attributes such as size, mass, and temperature.

Let's start with this basic class definition for a planet in `planet.py`, consisting of little more than an initializer. There are no properties here yet, we'll add them in a moment:

```
# planet.py

class Planet:

    def __init__(self,
                 name,
                 radius_metres,
                 mass_kilograms,
                 orbital_period_seconds,
                 surface_temperature_kelvin):
        self.name = name
        self.radius_metres = radius_metres
        self.mass_kilograms = mass_kilograms
```

```
        self.orbital_period_seconds = orbital_period_seconds
        self.surface_temperature_kelvin = surface_temperature_kelvin
```

This is simple enough to use[9]:

```
>>> pluto = Planet(name='Pluto', radius_metres=1184e3,
...                mass_kilograms=1.305e22, orbital_period_seconds=7816012992,
...                surface_temperature_kelvin=55)
>>> pluto.radius_metres
1184000.0
```

Unfortunately, our code also allows us to represent nonsensical situations, such as setting a negative radius, either by directly mutating an attribute, like this:

```
>>> pluto.radius_metres = -10000
```

Or by simply passing nonsense such as zero mass, negative orbital periods, or temperatures below absolute zero to the constructor:

```
>>> planet_x = Planet(name='X', radius_metres=10e3, mass_kilograms=0,
... orbital_period_seconds=-7293234, surface_temperature_kelvin=-5)
>>> planet_x.surface_temperature_kelvin
-5
```

We already know how to improve this: by wrapping our instance attribute in property getters and setters which perform validation by checking that the physical quantities are positive. We then assign through those properties in `__init__` to get validation on construction for free:

```
# planet.py

class Planet:

    def __init__(self,
                 name,
                 radius_metres,
                 mass_kilograms,
                 orbital_period_seconds,
                 surface_temperature_kelvin):
        self.name = name
        self.radius_metres = radius_metres
        self.mass_kilograms = mass_kilograms
        self.orbital_period_seconds = orbital_period_seconds
        self.surface_temperature_kelvin = surface_temperature_kelvin

    @property
    def name(self):
        return self._name

    @name.setter
```

```
    def name(self, value):
        if not value:
            raise ValueError("Cannot set empty Planet.name")
        self._name = value

    @property
    def radius_metres(self):
        return self._radius_metres

    @radius_metres.setter
    def radius_metres(self, value):
        if value <= 0:
            raise ValueError("radius_metres value {} is not "
                             "positive.".format(value))
        self._radius_metres = value

    @property
    def mass_kilograms(self):
        return self._mass_kilograms

    @mass_kilograms.setter
    def mass_kilograms(self, value):
        if value <= 0:
            raise ValueError("mass_kilograms value {} is not "
                             "positive.".format(value))
        self._mass_kilograms = value

    @property
    def orbital_period_seconds(self):
        return self._orbital_period_seconds

    @orbital_period_seconds.setter
    def orbital_period_seconds(self, value):
        if value <= 0:
            raise ValueError("orbital_period_seconds value {} is not "
                             "positive.".format(value))
        self._orbital_period_seconds = value

    @property
    def surface_temperature_kelvin(self):
        return self._surface_temperature_kelvin

    @surface_temperature_kelvin.setter
    def surface_temperature_kelvin(self, value):
        if value <= 0:
            raise ValueError("surface_temperature_kelvin value {} is not "
                             "positive.".format(value))
        self._surface_temperature_kelvin = value
```

From a robustness standpoint, this code is much better. For example, we can no longer construct massless planets:

```
>>> planet_x = Planet(name='X', radius_metres=10e3, mass_kilograms=0,
...                   orbital_period_seconds=-7293234, surface_temperature_kelvin=-5)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "examples/descriptors/properties.py", line 12, in __init__
    self.mass_kilograms = mass_kilograms
  File "examples/descriptors/properties.py", line 43, in mass_kilograms
```

```
    raise ValueError("mass_kilograms value {} is not positive.".format(value))
ValueError: mass_kilograms value 0 is not positive.
```

The trade-off though, is that the amount of code has exploded, and worse, there is a lot of duplicated code checking that all those numeric attribute values are non-negative.

Descriptors will ultimately provide a way out of this, but first we need to do a little more unravelling of properties to aid our understanding.

Unravelling the property function

In *The Python Journeyman* we introduced `property` as a function decorator for property getters. To briefly recap, a getter method, which encapsulates direct attribute access, is decorated by the property decorator which creates a property object. The getter function is bound to an attribute of the property object called `fget`, and the original name of the getter is conceptually re-bound to the property object.

The property object also has an attribute called `setter` which is in fact another decorator. When a setter method is decorator by the setter, the original property object is modifed to bind an attribute called `fset` to the setter method.

The property object effectively aggregates the getter and setter into a single property object which behaves like an attribute, and it behaves like an attribute because it is a *descriptor*. Shortly, we'll learn how it is able to appear so attribute-like, but first let's unravel properties a bit more.

Remember that function decorators are just regular functions which process an existing function and return a new object — usually a new function which wraps the decorated function.

Instead of using decorator syntax to apply the decorator to a function, we can define a regular undecorated function and then pass the function to the decorator, rebinding it to the same, or indeed any other, name.

Given that decorators are functions, let's rework our code to apply `property` explicitly using regular function-call syntax, avoiding the special decorator application syntax using the `@` symbol. When doing this, it's important to note that the `property()` function supports several arguments for simultaneously supplying the getter, setter and deleter, functions along with a docstring value. In fact, `help(property)` makes this quite clear:

```
>>> help(property)

Help on class property in module builtins:

  class property(object)
   |  property(fget=None, fset=None, fdel=None, doc=None) -> property attribute
   |
   |  fget is a function to be used for getting an attribute value, and likewise
   |  fset is a function for setting, and fdel a function for del'ing, an
   |  attribute.  Typical use is to define a managed attribute x:
   |
   |  class C(object):
   |      def getx(self): return self._x
   |      def setx(self, value): self._x = value
   |      def delx(self): del self._x
   |      x = property(getx, setx, delx, "I'm the 'x' property.")
   |
   |  Decorators make defining new properties or modifying existing ones easy:
   |
   |  class C(object):
   |      @property
   |      def x(self):
   |          "I am the 'x' property."
   |          return self._x
   |      @x.setter
   |      def x(self, value):
   |          self._x = value
   |      @x.deleter
   |      def x(self):
   |          del self._x
```

As you can see, we can separately define our getter and setter functions, then call the `property()` constructor within the *class* definition to produce a class attribute. Let's use this form in our `Planet` class for the numerical attributes. We'll leave the name attribute using the decorator form so you can compare side-by-side.

Implementing properties without decorators

First we'll remove the property decorator from the getter functions, before removing the setter decorator from the setter functions. Then we'll prefix the names of the getter functions with `_get_` and the names of the setter functions with `_set_`. We use single-underscore prefixes here because these are not special methods. Finally, we create a property object using the property function — you can think of it as a constructor call in this context — passing the pair of getter and setter functions. Continuing, we need to do the same for the `mass_kilograms` property, the `orbital_period_seconds` property, and the `surface_temperature_kelvin` property:

```
class Planet:

    def __init__(self,
                 name,
                 radius_metres,
                 mass_kilograms,
                 orbital_period_seconds,
                 surface_temperature_kelvin):
        self.name = name
        self.radius_metres = radius_metres
        self.mass_kilograms = mass_kilograms
        self.orbital_period_seconds = orbital_period_seconds
        self.surface_temperature_kelvin = surface_temperature_kelvin

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        if not value:
            raise ValueError("Cannot set empty Planet.name")
        self._name = value

    def _get_radius_metres(self):
        return self._radius_metres

    def _set_radius_metres(self, value):
        if value <= 0:
            raise ValueError("radius_metres value {} is not "
                             "positive.".format(value))
        self._radius_metres = value

    radius_metres = property(fget=_get_radius_metres,
                             fset=_set_radius_metres)

    def _get_mass_kilograms(self):
        return self._mass_kilograms

    def _set_mass_kilograms(self, value):
        if value <= 0:
            raise ValueError("mass_kilograms value {} is not "
                             "positive.".format(value))
        self._mass_kilograms = value

    mass_kilograms = property(fget=_get_mass_kilograms,
                              fset=_set_mass_kilograms)

    def _get_orbital_period_seconds(self):
        return self._orbital_period_seconds

    def _set_orbital_period_seconds(self, value):
        if value <= 0:
```

```
            raise ValueError("orbital_period_seconds value {} is not "
                             "positive.".format(value))
        self._orbital_period_seconds = value

    orbital_period_seconds = property(fget=_get_orbital_period_seconds,
                                      fset=_set_orbital_period_seconds)

    def _get_surface_temperature_kelvin(self):
        return self._surface_temperature_kelvin

    def _set_surface_temperature_kelvin(self, value):
        if value <= 0:
            raise ValueError("surface_temperature_kelvin value {} is not "
                             "positive.".format(value))
        self._surface_temperature_kelvin = value

    surface_temperature_kelvin = property(fget=_get_surface_temperature_kelvin,
                                          fset=_set_surface_temperature_kelvin)
```

If you think this form of property set up is a retrograde step compared to the decorator form, we'd agree with you; there's a good reason we introduce properties as decorators first. Nevertheless, seeing this alternative style reinforces the notion that `property` is simply a function, which returns an object called a descriptor, which is in turn bound to a class attribute.

The runtime behaviour of this code hasn't changed at all. We can still create objects, retrieve attribute values through properties, and attempt to set attributes values through properties with rejection of nonsensical values:

```
>>> pluto = Planet(name='Pluto', radius_metres=1184e3,
...                mass_kilograms=1.305e22, orbital_period_seconds=7816012992,
...                surface_temperature_kelvin=55)
>>> pluto.radius_metres
1184000.0
>>> pluto.radius_metres = -13
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "examples/descriptors/properties.py", line 31, in _set_radius_metres
    raise ValueError("radius_metres value {} is not positive.".format(value))
ValueError: radius_metres value -13 is not positive.
```

We already know what sort of operations we can perform with a descriptor. We can *get* a value, *set* a value and *delete* a value. In the case of `property`, these getting, setting and deleting operations call functions we supply, which query and manipulate instance attributes. In general though, the descriptor operations can be implemented to do almost anything.

Implementing a descriptor

We've seen that `property` is a descriptor which wraps three functions. Let's create a more specialised descriptor useful for modelling the strictly positive numeric values in our `Planet` class.

Here is a simple descriptor, called `Positive`:

```
from weakref import WeakKeyDictionary


class Positive:

    def __init__(self):
        self._instance_data = WeakKeyDictionary()

    def __get__(self, instance, owner):
        return self._instance_data[instance]

    def __set__(self, instance, value):
        if value <= 0:
            raise ValueError("Value {} is not positive".format(value))
        self._instance_data[instance] = value

    def __delete__(self, instance):
        raise AttributeError("Cannot delete attribute")
```

The descriptor class implements the three functions which comprise the descriptor protocol: `__get__()`, `__set__()`, and `__delete__()`. These are called when we get a value from a descriptor, set a value through a descriptor, or delete a value through a descriptor, respectively. In addition, the `Positive` class implements `__init__()` to configure new instances of the descriptor. Before we look in more detail at each of these methods, let's make use of our new descriptor to refactor our `Planet` class.

We remove the setters and getters for `radius_meters` and replace the call to the `property` constructor with a call to the `Positive` constructor. We do the same for the `mass_kilograms`, `orbital_period_seconds` and `surface_temperature_kelvin` quantities:

```
class Planet:

    def __init__(self,
                 name,
                 radius_metres,
                 mass_kilograms,
                 orbital_period_seconds,
                 surface_temperature_kelvin):
        self.name = name
        self.radius_metres = radius_metres
        self.mass_kilograms = mass_kilograms
```

```
        self.orbital_period_seconds = orbital_period_seconds
        self.surface_temperature_kelvin = surface_temperature_kelvin

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        if not value:
            raise ValueError("Cannot set empty Planet.name")
        self._name = value

    radius_metres = Positive()
    mass_kilograms = Positive()
    orbital_period_seconds = Positive()
    surface_temperature_kelvin = Positive()
```

With the `Positive` descriptor on hand, the `Planet` class shrinks by a huge amount!

At first sight, this may appear confusing. It looks like we're assigning to `radius_metres` twice — once in the initializer and once in the body of the class. In fact, the call in the body of the class is binding a instance of a `Positive` descriptor object to a *class* attribute of `Planet`. The call in `__init__` is then apparently assigning to an *instance* attribute — although as we'll see in a moment, this assignment is actually invoking a method on the descriptor object.

Allow us to explain the machinery we've created: As we said earlier `Postive` implements *descriptor protocol* consisting in full of the three methods `__get__`, `__set__` and `__delete__`.

Let's start with an instance of `Planet`:

```
>>> pluto = Planet(name='Pluto',
...                radius_metres=1184e3,
...                mass_kilograms=1.305e22,
...                orbital_period_seconds=7816012992,
...                surface_temperature_kelvin=55)
```

When we retrieve an attribute like this:

```
>>> pluto.mass_kilograms
1.305e+22
```

The `Positive.__get__()` function is called. The `instance` argument is set to `pluto` and the `owner` argument is set to the object which *owns* the descriptor, in this case the *class* `Planet`. So our attribute retrieval above, is equivalent to:

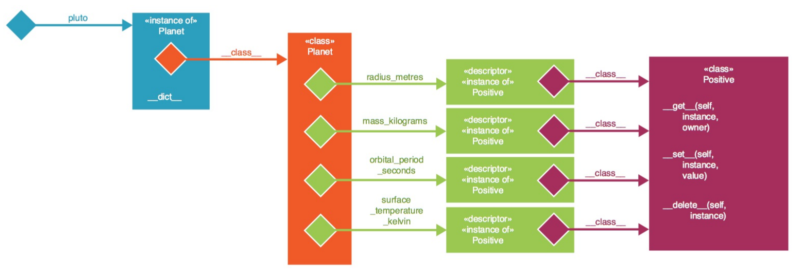
```
Positive.__get__(pluto, Planet)
```

Similarly, our assignments to the descriptors in the `Planet.__init__()` function resolve to calls equivalent to:

```
Positive.__set__(self, radius_metres)
```

At this point the new `Planet` instance isn't yet bound to the name `pluto`, only to `self`.

Here's what we have so far in a graphical format:



Positive descriptors on Pluto

We have a reference named `pluto` bound to an instance of `Planet`. The `__class__` reference of the `pluto` instance points to the `Planet` class object. The `Planet` class object contains four attributes, each of which references a distinct instance of the `Positive` descriptor object. The `__class__` reference of each descriptor object refers to the `Positive` class. When you put it all together, the call `m = pluto.mass_kilograms` boils down to `m = Positive.__get__(pluto, Planet)`.

Storing instance data

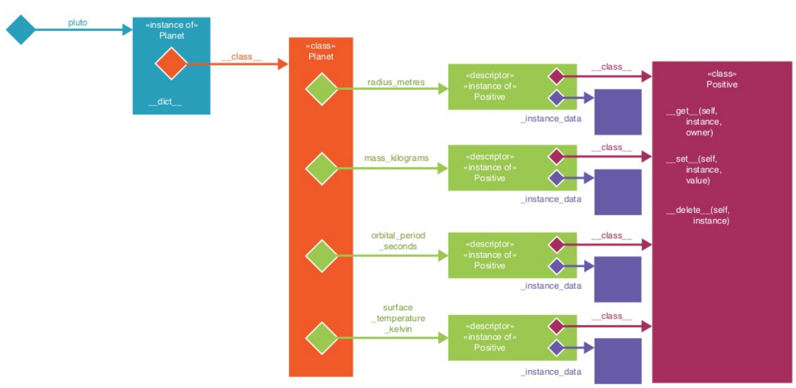
Notice that because the descriptor is owned by the `Planet` *class*, rather than by the `pluto` *instance*, we can't just store the value in the descriptor object. If we did that, the value would be shared between all `Planet` instances. Instead, we need to somehow associate the attribute value with the instance.

At first, this seems easy. The `__get__()` call is handed a reference to the instance - `pluto` in this case - so why not store the value in `pluto`'s `__dict__`?

There is a problem though: Within the descriptor class `Positive` we have no way of knowing to *which* attribute name the descriptor is bound in the `Planet` class. We can distinguish between descriptor instances in `__get__` using the `self` argument, but in Python objects do not know which names have been bound to them. This means we have no way of correlating between descriptor *instances* and the descriptor *names* embedded in the `Planet` class. In other words, none of our descriptor objects know which attribute they represent. This in turn means that we can't store the descriptor value in the `__dict__` of `Pluto`, because we wouldn't know what dictionary key to use.

> This apparent shortcoming of descriptors, not knowing the name of the class attribute to which they are bound, is also evident in the fact that our value validation error message in `__set__` no longer mentions the attribute name; this is a clear regression in capabilities. This is fixable, but the solution will have to wait to the *next* chapter when we look at metaclasses.

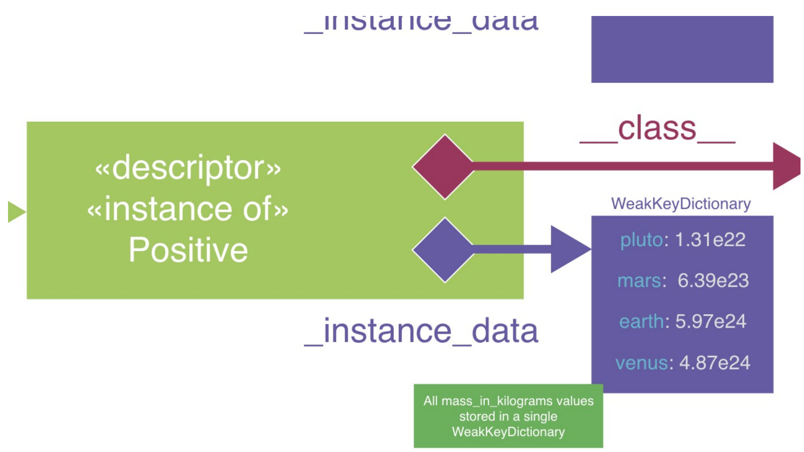
So, how to associate the values with this instances? Let's look at the solution pictorially first, then we'll reiterate with the code.



Instance data for Pluto

We use a special collection type from the Python Standard Library's [`weakref`] package called [`WeakKeyDictionary`]. This works pretty much like a regular dictionary except that it won't retain value objects which are referred to only by the dictionary key references; we say the references are *weak*. A `WeakKeyDictionary` owned by each descriptor instance is used to associate planet instances with the values of the quantity represented by that descriptor — although the descriptor itself doesn't know which quantity is being represented.

For example, the `WeakKeyDictionary` shown here associates planet instances with `mass_in_kilograms` values.



Instance data for planetary mass

Similar `WeakKeyDictionary` instances associate planet instances with `radius_in_metres`, `orbital_period_seconds`, and `surface_temperature_kelvin` values.

Because the dictionary keys are *weak* references, if a planet instance is destroyed — let's pretend the Earth is vapourised to make way for a hyperspace bypass — the corresponding entries in all the weak-key-dictionaries are also removed.

Examining the implementation

Let's look at how this is all put together in code. A `WeakKeyDictionary` instance called `_instance_data` is created in the descriptor initializer:

```
class Positive:

    def __init__(self):
        self._instance_data = WeakKeyDictionary()
```

As a result, our `Planet` class indirectly aggregates *four* such dictionaries, one in each of the four descriptors:

```
class Planet:
    # . . .
    radius_metres = Positive()
    mass_kilograms = Positive()
    orbital_period_seconds = Positive()
    surface_temperature_kelvin = Positive()
```

Within the `__set__()` method we associate the attribute value with the planet instance by inserting a mapping from the instance as key to the attribute as value:

```
class Positive:
   . . .
   def __set__(self, instance, value):
       if value <= 0:
           raise ValueError("Value {} is not positive".format(value))
       self._instance_data[instance] = value
```

As such, a single dictionary will contain all the `radius_metres` values for all `Planet` instances. Another dictionary will contain all `mass_kilograms` values for all `Planet` instances, and so on. We're storing the instance attribute values completely outside the instances, but in such a way that we can reliably retrieve them in `__get__`.

Retrieving descriptors on classes

One aspect of the descriptor protocol we haven't yet addressed is what happens when we retrieve a descriptor from a class. As we've seen instance attribute retrieval works fine:

```
>>> mercury, venus, earth, mars = main()
>>> mars.radius_metres
3389500.0
```

However, class attribute retrieval does not:

```
>>> Planet.radius_metres
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "examples/descriptors/properties_05.py", line 10, in __get__
    return self._instance_data[instance]
  File "/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/weakref.py", li
 345, in __getitem__
```

```
    return self.data[ref(key)]
TypeError: cannot create weak reference to 'NoneType' object
```

In such cases, the `instance` argument of `__get__()` will be set to `None`. Because we cannot create a weak reference to `None`, this causes a failure with the `WeakKeyDictionary` used for attribute storage.

By testing the instance argument against `None` we can detect when a descriptor value is being retrieved via a class attribute.

Here's a modified `Positive` descriptor:

```
class Positive:

    def __init__(self):
        self._instance_data = WeakKeyDictionary()

    def __get__(self, instance, owner):
        if instance is None:
            return self
        return self._instance_data[instance]

    def __set__(self, instance, value):
        if value <= 0:
            raise ValueError("Value {} is not positive".format(value))
        self._instance_data[instance] = value

    def __delete__(self, instance):
        raise AttributeError("Cannot delete attribute")
```

In the `__get__()` implementation when `instance` is `None` we do what seems most natural, returning the descriptor object itself rather than performing the attribute lookup. With the revised code we see more helpful behaviour:

```
>>> Planet.radius_metres
<properties_06.Positive object at 0x103012d68>
```

If you need to query or manipulate the class which contains the descriptor objects you can get hold of this through the `owner` argument of the `__get__` method, which in this case will contain a reference to the `Planet` class. In many cases though, you won't need to use `owner`, so you can do as we've done and just ignore it.

Data versus non-data descriptors

You may have heard the terms “data descriptor” and “non-data descriptor”, but what do they mean?

A *non-data descriptor* is a descriptor which implements only the `__get__()` method and so is read-only. So called *data descriptors* sport both the `__get__()` and `__set__()` methods and are writable.

The distinction is important because of the precedence of attribute lookup. A precedence chain controls attribute lookup according to the following rules:

- If an *instance*’s `__dict__` has an entry with the same name as a *data descriptor*, the data descriptor takes precedence.
- If an *instance*’s `__dict__` has an entry with the same name as a *non-data descriptor*, the dictionary entry takes precedence.

These statements are true, but quite a mouthful.

Another way of looking at it is that attribute lookup proceeds first to data descriptors (such as properties defined in the class), then to instance attributes in **dict**, and then on to non-data descriptors in the class again.

Approaching the difference experimentally

A simple experiment should make things clearer. Here we define a data descriptor, a non-data descriptor and a class which uses them both:

```
# descriptors.py

class DataDescriptor:

    def __get__(self, instance, owner):
        print("DataDescriptor.__get__({!r}, {!r}, {!r})"
              .format(self, instance, owner))

    def __set__(self, instance, value):
        print("DataDescriptor.__set__({!r}, {!r}, {!r})"
              .format(self, instance, value))


class NonDataDescriptor:

    def __get__(self, instance, owner):
        print("NonDataDescriptor.__get__({!r}, {!r}, {!r})"
              .format(self, instance, owner))
```

```
class Owner:

    a = DataDescriptor()
    b = NonDataDescriptor()
```

Let’s try this in a REPL session. After we’ve created an instance of `Owner`, we’ll retrieve the attribute `a`, set an item in the instance dictionary with the same name, and retrieve `a` again:

```
>>> from descriptors import *
>>> obj = Owner()
>>> obj.a
DataDescriptor.__get__(
    <precedence.DataDescriptor object at 0x102071748>,
    <precedence.Owner object at 0x102071550>,
    <class 'precedence.Owner'>)
>>> obj.__dict__['a'] = 196883
>>> obj.a
DataDescriptor.__get__(
    <precedence.DataDescriptor object at 0x102071748>,
    <precedence.Owner object at 0x102071550>,
    <class 'precedence.Owner'>)
```

Since `a` is a data descriptor, the first rule applies and the data descriptor takes precedence when we reference `obj.a`.

Now let’s try that with the non-data attribute `b`:

```
>>> obj.b
NonDataDescriptor.__get__(
    <precedence.NonDataDescriptor object at 0x102071780>,
    <precedence.Owner object at 0x102071550>,
    <class 'precedence.Owner'>)
>>> obj.__dict__['b'] = 744
>>> obj.b
744
```

The first time we access `obj.b` there is no entry of the same name in the instance dictionary. As a result, the non-data descriptor takes precedence. After we’ve added a `b` entry into `__dict__`, the second rule applies and the dictionary entry takes precedence over the non-data descriptor.

Summary

We have seen that the descriptor protocol is itself very simple, which can lead to very concise and declarative code, which hugely reduces duplication. At the same time, implementing descriptors correctly can be tricky and requires careful testing.

Let’s review the key points:

- We demonstrated how to create property descriptors without using decorator syntax by passing property getters and setters directly to the `property()` function. This reinforced the notion that properties create objects called *descriptors* which are bound to *class* attributes.
- We showed how to implement a simple descriptor to perform a basic attribute validation check by creating a class that implemented the *descriptor protocol*.
- We explained how descriptor instances have no knowledge of to which class attributes they are bound. This means that each descriptor instance must store the instance attributes for all descriptor owner instances. This can be achieved using a `WeakKeyDictionary` from the Python Standard Library `weakref` module.
- We saw how to determine whether descriptors are retrieved from their owning *classes* rather than via *instances*, by detecting when the `instance` argument to `__get__()` is set to `None`. A natural course of action in such as case is to return the descriptor instance itself.
- We explained the distinction between data and non-data descriptors, and how this relates to attribute lookup precedence.

We’re not quite done with descriptors yet, and in particular we’d like descriptor instances to know the name of the class attributes to which they have been bound. To solve that problem we need sharper tools, in the form of Python’s metaclasses, which we’ll be covering in the next chapter.

Chapter 5 - Instance Creation

In this chapter module we’ll be taking a deep-dive into exactly what happens when we create a new object. With this knowledge in hand, we’ll be able to exercise fine control over instance creation, which allows us to customize Python objects in powerful ways.

Instance Creation

So what *does* happen when you create an object?

To illustrate, we’ll an 8×8 chess board. Consider this simple class which represents a coordinate on a chess board consisting of a *file* (column) letter from ‘a’ to ‘h’ inclusive and a *rank* (row) number from 1 to 8 inclusive:

```
class ChessCoordinate:

    def __init__(self, file, rank):

        if len(file) != 1:
            raise ValueError("{} component file {!r} does not have a length of one."
                             .format(type(self).__name__, file))

        if file not in 'abcdefgh':
            raise ValueError("{} component file {!r} is out of range."
                             .format(type(self).__name__, file))

        if rank not in range(1, 9):
            raise ValueError("{} component rank {!r} is out of range."
                             .format(type(self).__name__, rank))

        self._file = file
        self._rank = rank

    @property
    def file(self):
        return self._file

    @property
    def rank(self):
        return self._rank

    def __repr__(self):
        return "{}(file={}, rank={})".format(
            type(self).__name__, self.file, self.rank)
```

```
    def __str__(self):
        return '{}{}'.format(self.file, self.rank)
```

This class implements an immutable value type; the initialiser establishes the invariants and the property accessors prevent inadvertent modification of the encapsulated data.

`__init__` is not a constructor

It's easy to think of `__init__()` as the *constructor* implementation, but let's look closely at what happens when we create an instance of `ChessCoordinate` by calling the constructor. We'll use a debugger — PDB in this case, but any debugger will do — to pause on the first line of `__init__()`. Since `__init__()` accepts `self` as an argument, it's clear that the object referred to by `self` *already exists*. That is to say, the object has already been constructed and the job of `__init__()` really is just to initialise the object:

```
>>> white_queen = ChessCoordinate('d', 4)
> /Users/sixtynorth/sandbox/chess_coordinate.py(6)__init__()
-> if len(file) != 1:
(Pdb) self
*** AttributeError: 'ChessCoordinate' object has no attribute '_file'
```

The `AttributeError` indicates that our debugger is having difficulty displaying the uninitialised object. This is because PDB is using `repr` to display it, and our `__repr__()` implementation, quite reasonably, expects that the object has been initialised. However, we can check `type(self)` to see that `self` already has the required type, and we can look at `self.__dict__` to see that the instance dictionary is empty:

```
(Pdb) type(self)
<class '__main__.ChessCoordinate'>
(Pdb) self.__dict__
{}
```

As we continue to step through the initializer, we can watch the instance dictionary get populated through assignments to attributes of `self`. From chapter 3, we know that behind the scenes `object.__setattr__()` is being called:

```
(Pdb) next
> /Users/sixtynorth/sandbox/chess_coordinate.py(10)__init__()
```

```
-> if file not in 'abcdefgh':
(Pdb) next
> /Users/sixtynorth/sandbox/chess_coordinate.py(14)__init__()
-> if rank not in range(1, 9):
(Pdb) next
> /Users/sixtynorth/sandbox/chess_coordinate.py(18)__init__()
-> self._file = file
(Pdb) next
> /Users/sixtynorth/sandbox/chess_coordinate.py(19)__init__()
-> self._rank = rank
(Pdb) self.__dict__
{'_file': 'd'}
(Pdb) next
--Return--
> /Users/sixtynorth/sandbox/chess_coordinate.py(19)__init__()->None
-> self._rank = rank
(Pdb) self.__dict__
{'_file': 'd', '_rank': 4}
```

Note also that `__init__()` doesn't return anything; it simply mutates the instance it has been given.

Construction and `__new__`

So, if `__init__()` isn't responsible for creating the instance, what is? If we use `dir()` to look at the special methods of `ChessCoordinate`, we see one called `__new__()`:

```
>>> dir(ChessCoordinate)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'file',
'rank']
```

We haven't defined `__new__()` ourselves, but we do inherit an implementation from the universal base class, `object`. It is this base-class implementation of `__new__()` which is responsible for allocating our object in this case:

```
>>> ChessCoordinate.__new__ is object.__new__
True
```

But what is the signature of `__new__()`? Don't bother looking in the `help()` because, frankly, the answer isn't very helpful!:

```
>>> help(object.__new__)
Help on built-in function __new__:
```

```
__new__(*args, **kwargs) method of builtins.type instance
    Create and return a new object.  See help(type) for accurate signature
```

Instead, let's figure out the signature of `__new__()` by instrumentation. We'll implement the most basic override of `__new__` which simply delegates to the base-class implementation. We'll add a few print statements so we can easily inspect the arguments and return value:

```
class ChessCoordinate:

    def __new__(cls, *args, **kwargs):
        print("cls =", cls)
        print("args =", repr(args))
        print("kwargs =", repr(kwargs))
        obj = super().__new__(cls)
        print("id(obj) =", id(obj))
        return obj

    def __init__(self, file, rank):
        print("id(self) =", id(self))

        if len(file) != 1:
            raise ValueError("{} component file {!r} does not have a length of one."
                             .format(type(self).__name__, file))

        if file not in 'abcdefgh':
            raise ValueError("{} component file {!r} is out of range."
                             .format(type(self).__name__, file))

        if rank not in range(1, 9):
            raise ValueError("{} component rank {!r} is out of range."
                             .format(type(self).__name__, rank))

        self._file = file
        self._rank = rank

    # Other implementation omitted...
```

> Notice that `__new__()` appears to be implicitly a *class* method. It accepts `cls` as it's first argument, rather than self. In fact `__new__()` is a specially-cased static method that happens to take the type of the class as its first argument — but the distinction isn't important here.

Here's what it looks like when we use this in a REPL:

```
>>> white_queen = ChessCoordinate('d', 4)
cls = <class '__main__.ChessCoordinate'>
args = ('d', 4)
kwargs = {}
id(obj) = 4598210744
id(self) = 4598210744
```

The `cls` argument is the *class* of the new object which will be be allocated. In our case, that will be `ChessCoordinate`, but in the presence of inheritance, it isn't necessarily the case that the `cls` argument will be set to the class enclosing the `__new__()` definition.

In general, besides the class object, `__new__()` also accepts whatever arguments have been passed to the constructor. In this case, we've soaked up any such arguments using `*args` and `**kwargs` although we could have used specific argument names here, just as we have with `__init__()`. We print these additional argument values to the console.

Remember that the purpose of `__new__()` is to allocate a new object. There's no special command for that in Python — all object allocation must ultimately be done by `object.__new__()`. Rather than call the `object.__new__()` implementation directly, we'll call it via `super()`. Should our immediate base class change in the future, this is more maintainable. The return value from the call to `object.__new__()` is a new, uninitialised instance of `ChessCoordinate`. We print its `id()` (remember we can't expect `repr()` to work yet) and then return.

It is this returned object which is then passed as the `self` argument to `__init__()`. We have printed the `id()` of `self` here to demonstrate that this is indeed the case.

Customising allocation

We've shown the essential mechanics of overriding `__new__()`: we accept the class type and the constructor arguments and return an instance of the correct type. Ultimately the only means we have for creating new instances is by calling `object.__new__()`. This is all well and good, but what are some practical uses?

One use for controlling instance creation is a technique called *interning*, which can dramatically reduce memory consumption. We'll demonstrate this by extending our program to allocate some chess boards in the start-of-game configuration. In our implementation each board is represented as a dictionary mapping the name of a piece to one of our `ChessCoordinate` objects. For fun, we've used the Unicode chess code points to represent our pieces. In our program `'♛♜'` means “black queen’s rook” and `'♔♗♙'` mean “white king’s bishop’s pawn”. We need to be this specific because, as you’ll remember, dictionaries require that keys are distinct.

We’ll revise our program by first removing `ChessCoordinate.__new__` and removing the call to `print` in `__init__`. We’ll also add a `starting_board()` function that creates a board in the starting configuration:

```
def starting_board():
    return {'♕♖': ChessCoordinate('a', 1),
            '♕♘': ChessCoordinate('b', 1),
            '♕♗': ChessCoordinate('c', 1),
            '♕♕': ChessCoordinate('d', 1),
            '♔♔': ChessCoordinate('e', 1),
            '♔♗': ChessCoordinate('f', 1),
            '♔♘': ChessCoordinate('g', 1),
            '♔♖': ChessCoordinate('h', 1),
            '♕♖♙': ChessCoordinate('a', 2),
            '♕♘♙': ChessCoordinate('b', 2),
            '♕♗♙': ChessCoordinate('c', 2),
            '♕♕♙': ChessCoordinate('d', 2),
            '♔♔♙': ChessCoordinate('e', 2),
            '♔♗♙': ChessCoordinate('f', 2),
            '♔♘♙': ChessCoordinate('g', 2),
            '♔♖♙': ChessCoordinate('h', 2),
            '♛♜': ChessCoordinate('a', 8),
            '♛♞': ChessCoordinate('b', 8),
            '♛♝': ChessCoordinate('c', 8),
            '♛♛': ChessCoordinate('d', 8),
            '♚♚': ChessCoordinate('e', 8),
            '♚♝': ChessCoordinate('f', 8),
            '♚♞': ChessCoordinate('g', 8),
            '♚♜': ChessCoordinate('h', 8),
            '♛♜♟': ChessCoordinate('a', 7),
            '♛♞♟': ChessCoordinate('b', 7),
            '♛♝♟': ChessCoordinate('c', 7),
            '♛♛♟': ChessCoordinate('d', 7),
            '♚♚♟': ChessCoordinate('e', 7),
            '♚♝♟': ChessCoordinate('f', 7),
            '♚♞♟': ChessCoordinate('g', 7),
            '♚♜♟': ChessCoordinate('h', 7),
           }
```

Let’s go to the REPL and create a board using our new function:

```
>>> board = starting_board()
```

After creating a single board this way, our system reports that our Python process is using about 7 MB of memory.[10] Creating 10,000 chess-boards utilises some 75 MB of memory to store the 320,000 instances of

`ChessCoordinate` contained by the 10,000 dictionaries that have been created to represent all the boards:

```
>>> boards = [starting_board() for _ in range(10000)]
```

Bear in mind though, that there are only 64 distinct positions on a chess board, and given that our `ChessCoordinate` objects are deliberately immutable value types, we should never need more than 64 instances. In our specific case, in fact, we should never need more that 32 instances.

Interning allocated objects

Here are updated definitions for `__new__()` and `__init__()` which allow us to limit allocations to just the necessary ones, one per coordinate:

```
class ChessCoordinate:

    _interned = {}

    def __new__(cls, file, rank):
        if len(file) != 1:
            raise ValueError("{} component file {!r} does not have a length of one."
                             .format(cls.__name__, file))

        if file not in 'abcdefgh':
            raise ValueError("{} component file {!r} is out of range."
                             .format(cls.__name__, file))

        if rank not in range(1, 9):
            raise ValueError("{} component rank {!r} is out of range."
                             .format(cls.__name__, rank))

        key = (file, rank)
        if key not in cls._interned:
            obj = super().__new__(cls)
            obj._file = file
            obj._rank = rank
            cls._interned[key] = obj

        return cls._interned[key]

    def __init__(self, file, rank):
        pass
```

We're now using named positional arguments for the `file` and `rank` arguments to `__new__()`, and we've moved the validation logic from `__init__()` to `__new__()`.

Once the arguments are validated, we use them to create a single `tuple` object from `file` and `rank`. We use this tuple as a key and check if there is an entry against this key tuple in a dictionary called `_interned` which we've attached as a class-attribute. Only if the tuple key is not present in the dictionary do we allocate a new instance; we do this by calling `object.__new__()`. We then configure the new instance — doing the remainder of the work that used to be done in `__init__()` — and insert the newly minted instance into the dictionary. Of course, the instance we return is an element in the dictionary, either the one we just created or one created earlier.

Our `__init__()` method is now empty, and in fact we could remove it entirely.

With these changes in place allocating 10,000 boards takes significantly less memory than before — we're down to about 23 MB.

Interning is a powerful tool for managing memory usage — in fact Python uses it internally for integers and strings — but it should only be used with immutable value types such as our `ChessCoordinate` where instances can safely be shared between data structures.

Now that we understand `__new__()`, in the next chapter we can move on to another advanced Python topic: metaclasses.

Summary

We've covered an crucial topic in this short chapter: the distinction between the allocation and initialization of instances. Let's summarise what we covered:

- We showed how the static method `__new__()` is called to allocate and return a new instance.
  - `__new__()` is implicitly a static method which accepts the class of the new instance as its first argument, and it doesn't require the either the `@classmethod` or `@staticmethod` decorators
- Ultimately, `object.__new__()` is responsible for allocating all instances.

- One use for overriding `__new__()` is to support instance interning which can be useful when certain values of immutable value types are very common, or when the domain of values is small and finite, such as with the squares of a chess board.

This isn't the whole story though, and Python offers yet more control over instance creation at the class level. Before we get to that, though, we need to understand metaclasses.

Chapter 6 - Metaclasses

We've mentioned metaclasses several times. We owe you an explanation!

In this chapter, we'll look at:

- the concept of the class of class objects — *metaclass* for short
- the default metaclass, called `type`
- specifying metaclasses in class definitions
- defining metaclasses, including the special methods of metaclasses
- practical examples of metaclasses which solve some problems we discovered earlier in the book
- how metaclasses interact with inheritance

The class of class objects

To assist us on our journey to understand metaclasses, we need a simple class. We'll use `Widget`, which will be empty:

```
class Widget:
    pass
```

We'll instantiate some widgets:

```
>>> w = Widget()
>>> w
<__main__.Widget object at 0x1006747b8>
```

and introspect their types:

```
>>> type(w)
<class '__main__.Widget'>
```

We all know that in Python, the type of an instance is its class. So what is the type of its *class*?:

```
>>> type(Widget)
<class 'type'>
```

Using `type` as a function and a value

The type of a class is `type`. We've covered this before in chapter 12 of The Python Journeyman, but it bears repeating here. One potentially confusing aspect here is that we are using `type()` as a function to determine the type of an object, but `type` is also being used as a value — it is the type of a class object. However, this duality isn't so unusual; we see exactly the same situation with, say, `list` where it is used as a constructor and a value:

```
>>> a = list("A list")
>>> a
['A', ' ', 'l', 'i', 's', 't']
>>> type(a)
<class 'list'>
```

Given that the type of the `Widget` class is `type` we can say that the *metaclass* of `Widget` is `type`. In general, the type of any class object in Python is its metaclass, and the default metaclass is `type`.

Going further, we can discover that the type of `type` is `type`:

```
>>> type(type)
<class 'type'>
```

We can get the same results by drilling down through special attributes:

```
>>> w.__class__
<class '__main__.Widget'>
>>> w.__class__.__class__
<class 'type'>
>>> w.__class__.__class__.__class__
<class 'type'>
```

Metaclasses and class creation

To understand where metaclasses fit into Python, we must consider how class objects are created. When we define a class in a Python source file, like this:

```
class Widget:
    pass
```

this is actually shorthand for:

```
class Widget(object, metaclass=type)
    pass
```

In addition to passing base classes as positional arguments we can pass the `metaclass` keyword argument. In the same way that the base class is implicitly `object`, the metaclass is implicitly `type`.

Class allocation and initialisation

Roughly speaking when we write a `class` block in our Python source code, it is syntactic sugar for creating a dictionary which is passed to a metaclass to convert the dictionary into a class object. To see how that works, it’s import to understand the several tasks that metaclasses perform during creation of a new class. When we write:

```
class Widget:
    pass
```

what is actually happening is something like this:

```
name = 'Widget'
metaclass = type
bases = ()
kwargs = {}
namespace = metaclass.__prepare__(name, bases, **kwargs)
Widget = metaclass.__new__(metaclass, name, bases, namespace, **kwargs)
metaclass.__init__(Widget, name, bases, namespace, **kwargs)
```

Which is to say that:

- The class name is ‘Widget’
- The metaclass is `type`
- The class has no base classes, other than the implicit `object`
- No keyword arguments were passed to the metaclass — we’ll cover what this means later
- The metaclass’s `__prepare__()` method is called to create a new namespace object, which behaves like a dictionary
- Behind the scenes, the Python runtime populates the namespace dictionary while reading the contents of the class-block
- The metaclass’s `__new__()` method is called to allocate the class object
- The metaclass’s `__init__()` is called to initialise the class object.

The `name`, `bases` and `namespace` arguments contain the information collected during execution of the class definition — normally the class attributes and method definitions inside the class block — although in our case the class-block is logically empty.

Tracing metaclass invocations

By providing our own metaclass we can customise these behaviours. We'll start with a very simple metaclass called `TracingMeta` in a module `tracing.py`. `TracingMeta` prints its method invocations and return values to the console for each of `__prepare__()`, `__new__()` and `__init__()`. Notice that our metaclass needs to be a subclass of an existing metaclass, so we will subclass `type`. Each of our overrides delegates to the base-class `type` to do the actual work required, via a call to `super()`:

```
class TracingMeta(type):

    @classmethod
    def __prepare__(mcs, name, bases, **kwargs):
        print("TracingMeta.__prepare__(name, bases, **kwargs)")
        print("mcs =", mcs)
        print("name =", name)
        print("bases =", bases)
        print("kwargs =", kwargs)
        namespace = super().__prepare__(name, bases)
        print("<-- namespace =", namespace)
        print()
        return namespace

    def __new__(mcs, name, bases, namespace, **kwargs):
        print("TracingMeta.__new__(mcs, name, bases, namespace)")
        print("mcs =", mcs)
        print("name =", name)
        print("bases =", bases)
        print("namespace =", namespace)
        print("kwargs =", kwargs)
        cls = super().__new__(mcs, name, bases, namespace)
        print("<-- cls =", cls)
        print()
        return cls

    def __init__(cls, name, bases, namespace, **kwargs):
        print("TracingMeta.__init__(cls, name, bases, namespace)")
        print("cls =", cls)
        print("name =", name)
        print("bases =", bases)
        print("namespace =", namespace)
        print("kwargs =", kwargs)
        super().__init__(name, bases, namespace)
        print()
```

Notice that although `__new__()` is *implictly* a class-method, we must explicitly decorate the `__prepare__()` class-method with the appropriate decorator.

Now we'll define a class containing a simple method called `action()` which prints a message and a single class attribute `the_answer` with the [cosmically inevitable value 42](). We'll do this at the REPL so you can see clearly *when* the metaclass machinery is invoked:

```
>>> from tracing import TracingMeta
>>> class Widget(metaclass=TracingMeta):
...     def action(message):
...         print(message)
...     the_answer = 42
...
TracingMeta.__prepare__(name, bases, **kwargs)
mcs = <class 'tracing.TracingMeta'>
name = Widget
bases = ()
kwargs = {}
<-- namespace = {}

TracingMeta.__new__(mcs, name, bases, namespace, **kwargs)
mcs = <class 'tracing.TracingMeta'>
name = Widget
bases = ()
namespace = {'__qualname__': 'Widget', 'action': <function Widget.action at 0x1030162f0
 'the_answer': 42, '__module__': 'builtins'}
kwargs = {}
<-- cls = <class 'Widget'>

TracingMeta.__init__(cls, name, bases, namespace, **kwargs)
cls = <class 'Widget'>
name = Widget
bases = ()
namespace = {'__qualname__': 'Widget', 'action': <function Widget.action at 0x1030162f0
 'the_answer': 42, '__module__': 'builtins'}
  kwargs = {}
```

The instant we complete the class definition we can see from the tracing output that Python executes the `__prepare__()`, `__new__()` and `__init__()` methods in turns.

The `__prepare__` method

First of all, lets look at `__prepare__()`, the purpose of which is to produce an initial mapping object to contain the class namespace. The `mcs` argument is a reference to the metaclass itself. This first argument is analagous to the `self` argument passed to instance methods and the `cls` argument passed to class-methods. For metaclass methods, it is conventionally called `mcs`.

The `name` argument contains the name of our `Widget` class as a string.

The `bases` argument is an empty tuple. We didn't declare any base classes for `Widget`, and the ultimate `object` base-class is implicit.

The `kwargs` argument is an empty dictionary. We'll cover the significance of this shortly.

The most important aspect of `__prepare__` is that when it calls its superclass implementation in `type`, the return value is a dictionary — or more generally a mapping type. In this case it's a regular empty dictionary. This dictionary will be the namespace associated with the nascent class.

The `__new__` method

Next we'll look in detail at `__new__()`, the purpose of which is to allocate the new class object. The `mcs` argument is a reference to the metaclass as before. The `name` and `bases` arguments are still the string name of the new class and the tuple of base classes.

The mapping object that we returned from `__prepare__()` is passed as the `namespace` argument to `__new__()`. The Python runtime has populated this dictionary with several entries as it has processed the class definition of `Widget`. Two of the items are our `action()` method and our `the_answer` class attribute. The other two items are `__module__` and `__qualname__` which the Python runtime has added.

The `__module__` attribute is mapped to the name of the module in which the class was defined. Because we used the REPL, this is `builtins` in this case. The `__qualname__` attribute contains the fully-qualified name of the class including parent modules and packages. In this case, it contains just the class name, as the `builtins` module used by the REPL — being the last namespace in the LEGB lookup hierarchy — is available everywhere, .

The `kwargs` dictionary passed to `__new__()` is also still empty.

Within `__new__()` we delegate to the base class `type.__new__()` via a call to `super()`, forwarding the `mcs`, `name`, `bases`, and `namespace` arguments. The object returned by this call *is* the new `Widget` class object that we are in the process of allocating and configuring. The new `Widget` class is what we return from `__new__()`. Note that any changes we wish to make to the contents of the namespace object must have been made *before* this call, as this is the point at which the class object is created. To change the contents of the class namespace after this call, the class object must be manipulated directly.

The `__init__` method

Finally we come to `__init__()` the purpose of which is to *configure* the newly created class object. Note that `__init__` here is an *instance method* of the metaclass, not an explicit class-method like `__prepare__()` or an implicit class-method like `__new__()`. As such it accepts `cls` as its first argument, which is one level less- meta than `mcs` in the same way that `self` is one level less meta than `cls`. The `name`, `bases`, `namespace` and `kwargs` arguments are all as before. Again we delegate to the `type` base class via `super()` although `__init__()` doesn't return anything as it is expected to modify the existing class object that was handed to it. Note that although the namespace object is passed to `__init__`, its contents will already have been used upstream by `__new__` when allocating the class object. Changes to `namespace` will be inneffectual at this juncture, and any changes to the class object must be made by manipulating `cls` directly.

Putting it all together

The key here is that metaclasses give us the opportunity to modify the dictionary of class attributes, which includes methods, before the class is instantiated. We even get opportunity to modify the list of base classes or produce an entirely different class if required, although such uses are rare. We'll look at some complex examples soon to make this clear.

You may be wondering which out of `__prepare__()`, `__new__()`, and `__init__()` you should override. If you don't override `__prepare__()`, the default implementation in `type` will produce a regular dictionary for the namespace object, so you only need to override it if you need the behaviour provided by another mapping type.

Usually, it will only be necessary to override either `__new__()` or `__init__()`, but of these two, only `__new__()` can make decisions *before* the new class is allocated. The distinction between `__new__()` and `__init__()` for metaclasses is exactly the same as it is for regular classes. Later we'll see that it might be wise to prefer configuration in `__init__()` rather than `__new__()` so that metaclasses are more composable.

Passing additional arguments to the metaclass

Earlier, we skipped over the fact that that `__prepare__()`, `__new__()`, and `__init__()` all sport a `**kwargs` parameter to accept arbitrary keyword arguments. These can be supplied in the parameter list when defining a class, and any keyword arguments provided over and above the a named `metaclass` argument will be forwarded to the three special methods.

Here's an example where we've passed a `tension` keyword argument in the parameter list of our `Reticulator` class:

```
>>> class Reticulator(metaclass=TracingMeta, tension=496):
...     def reticulate(self, spline):
...         print(spline)
...     cubic = True
...
TracingMeta.__prepare__(name, bases, **kwargs)
mcs = <class 'tracing.TracingMeta'>
name = Reticulator
bases = ()
kwargs = {'tension': 496}
<-- namespace = {}

TracingMeta.__new__(mcs, name, bases, namespace)
mcs = <class 'tracing.TracingMeta'>
name = Reticulator
bases = ()
namespace = {'reticulate': <function Reticulator.reticulate at 0x103016510>, '__qualnam
_': 'Reticulator', 'cubic': True, '__module__': 'builtins'}
kwargs = {'tension': 496}
<-- cls = <class 'Reticulator'>

TracingMeta.__init__(cls, name, bases, namespace)
cls = <class 'Reticulator'>
name = Reticulator
bases = ()
namespace = {'reticulate': <function Reticulator.reticulate at 0x103016510>, '__qualnam
_': 'Reticulator', 'cubic': True, '__module__': 'builtins'}
kwargs = {'tension': 496}
```

As you can see, the arguments are dutifully forwarded to the metaclass methods, and the argument values could have been used to configure the class object. This allows the class statement to be used as a kind of class *factory*.

An example of metaclass keywords

Here's an interesting example which uses `**kwargs`. The `__new__()` method of the `EntriesMeta` metaclass expects `kwargs` to contain a `num_entries` key which maps to an integer value. This is used to populate the namespace with named entries using the letters of the alphabet. It looks like we only need to override `__new__()` to achieve our aims:

```
class EntriesMeta(type):

    def __new__(mcs, name, bases, namespace, **kwargs):
        print("Entries.__new__(mcs, name, bases, namespace, **kwargs)")
        print("  kwargs =", kwargs)
        num_entries = kwargs['num_entries']
        print("  num_entries =", num_entries)
        namespace.update({chr(i): i for i in range(ord('a'), ord('a')+num_entries)})
        cls = super().__new__(mcs, name, bases, namespace)
        return cls
```

In `__new__()` we:

- Print `kwargs`
- Extract the `num_entries` value from `kwargs`
- Print `num_entries`
- Use a dictionary comprehension to generate a diction mapping letters to number values
- Update the namespace dictionary with our dictionary items
- Pass the modified namespace object on to `type.__new__()` via a call to `super()`

Let's try to use it:

```
>>> from entries import EntriesMeta
>>> class AtoZ(metaclass=EntriesMeta, num_entries=26):
...     pass
...
Entries.__new__(mcs, name, bases, namespace, **kwargs)
  kwargs = {'num_entries': 26}
  num_entries = 26
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: type.__init__() takes no keyword arguments
```

The problem we've set up here is that both `__init__()` and `__new__()` must accept any additional arguments — they must have the same signature. We need to add a do-nothing `__init__()` to keep Python happy:

```
def __init__(cls, name, bases, namespace, **kwargs):
    super().__init__(name, bases, namespace)
```

With this in place, everything works as intended:

```
>>> from entries import EntriesMeta
>>> class AtoZ(metaclass=EntriesMeta, num_entries=26):
...     pass
...
Entries.__prepare__(name, bases, **kwargs)
  mcs = <class 'entries.EntriesMeta'>
  name = AtoZ
  bases = ()
  kwargs = {'num_entries': 26}
<-- namespace = {}

Entries.__new__(mcs, name, bases, namespace, **kwargs)
  mcs = <class 'entries.EntriesMeta'>
  name = AtoZ
  bases = ()
  namespace = {'__module__': 'builtins', '__qualname__': 'AtoZ'}
  kwargs = {'num_entries': 26}
  num_entries = 26
<-- cls = <class 'AtoZ'>

Entries.__init__(cls, name, bases, namespace, **kwargs)
  cls = <class 'AtoZ'>
  name = AtoZ
  bases = ()
  namespace = {'__qualname__': 'AtoZ', 'j': 106, 'q': 113, 't': 116, 'v': 118, 'g': 103
'z': 122, '__module__': 'builtins', 'e': 101, 'k': 107, 'a': 97, 'b': 98, 'c': 99, 'y':
21, 'l': 108, 'i': 105, 'n': 110, 's': 115, 'h': 104, 'm': 109, 'o': 111, 'p': 112, 'w'
119, 'd': 100, 'r': 114, 'f': 102, 'u': 117, 'x': 120}
  kwargs = {'num_entries': 26}
```

We can see the `num_entries` item arrive in `kwargs`, and by the time we get to `__init__()` we can see the namespace object with additional entries.

We can use a regular argument as well as or instead of `**kwargs`. Let's convert `num_entries` to a proper argument:

```
class EntriesMeta(type):

    def __new__(mcs, name, bases, namespace, num_entries, **kwargs):
        print("Entries.__new__(mcs, name, bases, namespace, **kwargs)")
        print("  kwargs =", kwargs)
        print("  num_entries =", num_entries)
        namespace.update({chr(i): i for i in range(ord('a'), ord('a')+num_entries)})
```

```
        cls = super().__new__(mcs, name, bases, namespace)
        return cls
```

This works just as expected:

```
>>> from entries import EntriesMeta
>>> class AtoZ(metaclass=EntriesMeta, num_entries=10):
...     pass
...
Entries.__prepare__(name, bases, **kwargs)
  mcs = <class 'entries.EntriesMeta'>
  name = AtoZ
  bases = ()
  kwargs = {'num_entries': 10}
<-- namespace = {}

Entries.__new__(mcs, name, bases, namespace, **kwargs)
  mcs = <class 'entries.EntriesMeta'>
  name = AtoZ
  bases = ()
  namespace = {'__module__': 'builtins', '__qualname__': 'AtoZ'}
  kwargs = {}
  num_entries = 10
<-- cls = <class 'AtoZ'>

Entries.__init__(cls, name, bases, namespace, **kwargs)
  cls = <class 'AtoZ'>
  name = AtoZ
  bases = ()
  namespace = {'j': 106, 'f': 102, '__module__': 'builtins', 'd': 100, 'b': 98, 'g': 10
 'c': 99, '__qualname__': 'AtoZ', 'i': 105, 'a': 97, 'e': 101, 'h': 104}
  kwargs = {'num_entries': 10}
```

Metaclass methods and visibility

We’ve covered three important special methods for metaclasses, `__prepare__()`, `__new__()` and `__init__()`, but what happens if we include other methods in the metaclass? Let’s see, by adding a method called `metamethod` to the `TracingMeta` metaclass we were experimenting with earlier:

```
class TracingMeta(type):

    @classmethod
    def __prepare__(mcs, name, bases, **kwargs):
        print("TracingMeta.__prepare__(name, bases, **kwargs)")
        print("mcs =", mcs)
        print("name =", name)
        print("bases =", bases)
        print("kwargs =", kwargs)
        namespace = super().__prepare__(name, bases)
        print("<-- namespace =", namespace)
        print()
        return namespace
```

```
    def __new__(mcs, name, bases, namespace, **kwargs):
        print("TracingMeta.__new__(mcs, name, bases, namespace)")
        print("mcs =", mcs)
        print("name =", name)
        print("bases =", bases)
        print("namespace =", namespace)
        print("kwargs =", kwargs)
        cls = super().__new__(mcs, name, bases, namespace)
        print("<-- cls =", cls)
        print()
        return cls

    def __init__(cls, name, bases, namespace, **kwargs):
        print("TracingMeta.__init__(cls, name, bases, namespace)")
        print("cls =", cls)
        print("name =", name)
        print("bases =", bases)
        print("namespace =", namespace)
        print("kwargs =", kwargs)
        super().__init__(name, bases, namespace)
        print()

    def metamethod(cls):
        print("TracingMeta.metamethod(cls)")
        print("cls = ", cls)
        print()
```

Let’s create the class `Widget` with its `TracingMeta` metaclass:

```
>>> from tracing import *
>>> class Widget(metaclass=TracingMeta):
...     pass
...
TracingMeta.__prepare__(name, bases, **kwargs)
mcs = <class 'tracing.TracingMeta'>
name = Widget
bases = ()
kwargs = {}
<-- namespace = {}

TracingMeta.__new__(mcs, name, bases, namespace)
mcs = <class 'tracing.TracingMeta'>
name = Widget
bases = ()
namespace = {'__qualname__': 'Widget', '__module__': 'builtins'}
kwargs = {}
<-- cls = <class 'Widget'>

TracingMeta.__init__(cls, name, bases, namespace)
cls = <class 'Widget'>
name = Widget
bases = ()
namespace = {'__qualname__': 'Widget', '__module__': 'builtins'}
kwargs = {}
```

It turns out that regular methods of the metaclass can be accessed similarly to *class* methods of the `Widget` class, and they will have the class passed to them as the first (and implicit) argument:

```
>>> Widget.metamethod()
TracingMeta.metamethod(cls)
cls =  <class 'Widget'>
```

However, unlike regular class methods we create with the `@classmethod` decorator, we cannot access so-called metamethods via the widget instance:

```
>>> w = Widget()
>>> w.metamethod()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'Widget' object has no attribute 'metamethod'
```

Metamethods are rarely used in practice, although one metamethod in particular has an interesting use: `__call__()`, which we'll look at shortly.

Class method definitions in the regular class and its base classes will take precedence over looking up a method in its metaclass.

Regular methods of the metaclass accept `cls` as their first argument. This makes sense because `cls` (the class) is the 'instance' of the metaclass; it is analagous to `self`. On the other hand, class- methods of the metaclass accept `mcs` as their first argument — the metaclass — analagous to the `cls` argument of a class method in an actual class.

Fine-grained instantiation control with metaclass `__call__()`

For a moment, return your thoughts to the material we covered in chapter 5 when we looked a *instance* allocation with `__new__()`. We know that in order to create instances we call the constructor of the desired class:

```
>>> w = Widget()
```

We have learned that behind the scenes this will call `Widget.__new__()` to allocate a `Widget` followed by `Widget.__init__()` to do any further initialisation. Let's pull back the curtain, and see exactly what *is* "behind the scenes".

The behaviour of calling `__new__()` followed by `__init__()` when we call a constructor is actually the responsibility of `__call__()` on the metaclass. This makes sense when we remember that `__call__()` is a metamethod and therefore can be called like a classmethod, and that `__call__()` makes the objects on which it is defined *callable*, like functions. This is the mechanism by which *classes* in Python become callable, and what we have been referring hitherto as a *constructor* call is in fact the `__call__()` metamethod.

Let's see this in action by overriding `__call__()` in our `TracingMeta` example:

```
class TracingMeta(type):

    @classmethod
    def __prepare__(mcs, name, bases, **kwargs):
        print("TracingMeta.__prepare__(name, bases, **kwargs)")
        print("  mcs =", mcs)
        print("  name =", name)
        print("  bases =", bases)
        print("  kwargs =", kwargs)
        namespace = super().__prepare__(name, bases)
        print("<-- namespace =", namespace)
        print()
        return namespace

    def __new__(mcs, name, bases, namespace, **kwargs):
        print("TracingMeta.__new__(mcs, name, bases, namespace)")
        print("  mcs =", mcs)
        print("  name =", name)
        print("  bases =", bases)
        print("  namespace =", namespace)
        print("  kwargs =", kwargs)
        cls = super().__new__(mcs, name, bases, namespace)
        print("<-- cls =", cls)
        print()
        return cls

    def __init__(cls, name, bases, namespace, **kwargs):
        print("TracingMeta.__init__(cls, name, bases, namespace)")
        print("  cls =", cls)
        print("  name =", name)
        print("  bases =", bases)
        print("  namespace =", namespace)
        print("  kwargs =", kwargs)
        super().__init__(name, bases, namespace)
        print()

    def metamethod(cls):
        print("TracingMeta.metamethod(cls)")
        print("  cls = ", cls)
        print()

    def __call__(cls, *args, **kwargs):
        print("TracingMeta.__call__(cls, *args, **kwargs)")
        print("  cls =", cls)
```

```
        print("  args =", args)
        print("  kwargs =", kwargs)
        print("  About to call type.__call__()")
        obj = super().__call__(*args, **kwargs)
        print("  Returned from type.__call__()")
        print("<-- obj =", obj)
        print()
        return obj
```

We'll also implement a `TracingClass` which will use `TracingMeta` as its metaclass. `TracingClass` overrides `__new__()` and `__init__()` so we can see when they're called:

```
class TracingClass(metaclass=TracingMeta):

    def __new__(cls, *args, **kwargs):
        print("  TracingClass.__new__(cls, args, kwargs")
        print("    cls =", cls)
        print("    args =", args)
        print("    kwargs =", kwargs)
        obj = super().__new__(cls)
        print("  <-- obj =", obj)
        print()
        return obj

    def __init__(self, *args, **kwargs):
        print("  TracingClass.__init__(self, *args, **kwargs")
        print("    self =", self)
        print("    args =", args)
        print("    kwargs =", kwargs)
        print()
```

Notice that when we import the module into the REPL, the metaclass trifecta `__prepare__()`, `__new__()`, and `__init__()` are invoked when the `TracingClass` is defined:

```
>>> from tracing import *
TracingMeta.__prepare__(name, bases, **kwargs)
  mcs = <class 'tracing.TracingMeta'>
  name = TracingClass
  bases = ()
  kwargs = {}
<-- namespace = {}

TracingMeta.__new__(mcs, name, bases, namespace)
  mcs = <class 'tracing.TracingMeta'>
  name = TracingClass
  bases = ()
  namespace = {'__new__': <function TracingClass.__new__ at 0x103016488>, '__init__': <
nction TracingClass.__init__ at 0x103016510>, '__qualname__': 'TracingClass', '__module
': 'tracing'}
  kwargs = {}
<-- cls = <class 'tracing.TracingClass'>

TracingMeta.__init__(cls, name, bases, namespace)
```

```
  cls = <class 'tracing.TracingClass'>
  name = TracingClass
  bases = ()
  namespace = {'__new__': <function TracingClass.__new__ at 0x103016488>, '__init__': <
nction TracingClass.__init__ at 0x103016510>, '__qualname__': 'TracingClass', '__module
': 'tracing'}
  kwargs = {}
```

Now we'll instantiate `TracingClass` with a positional argument and a keyword argument:

```
>>> t = TracingClass(42, keyword="clef")
TracingMeta.__call__(cls, *args, **kwargs)
  cls = <class 'tracing.TracingClass'>
  args = (42,)
  kwargs = {'keyword': 'clef'}
  About to call type.__call__()
TracingClass.__new__(cls, args, kwargs
  cls = <class 'tracing.TracingClass'>
  args = (<class 'tracing.TracingClass'>, 42)
  kwargs = {'keyword': 'clef'}
<-- obj = <tracing.TracingClass object at 0x103012ef0>

TracingClass.__init__(self, *args, **kwargs
  self = <tracing.TracingClass object at 0x103012ef0>
  args = (<class 'tracing.TracingClass'>, 42)
  kwargs = {'keyword': 'clef'}

  Returned from type.__call__()
<-- obj = <tracing.TracingClass object at 0x103012ef0>
```

Look carefully at the control flow here. Our call to the constructor invokes `__call__()` on the metaclass, which receives the arguments we passed to the constructor in addition to the type we're trying to construct.

Our `__call__()` override calls the superclass implementation, which is `type.__call__()` in this case. See how `type.__call__()` in turn calls `TracingClass.__new__()` followed by `TracingClass.__init__()`. In other words, it is `type.__call__()` which orchestrates the default class allocation and initialisation behaviour.

It's very rare to see the `__call__()` metamethod overridden. It's pretty low level in Python terms and provides some of the most basic Python machinery. That said, it can be powerful.

An example of overriding `__call__` on a metaclass

In `keywordmeta.py` we have an example of a metaclass overriding `__call__()` to prevent classes which use the metaclass accepting positional arguments to their constructors:

```
class KeywordsOnlyMeta(type):

    def __call__(cls, *args, **kwargs):
        if args:
            raise TypeError("Constructor for class {!r} does not accept positional argu
nts.".format(cls))
        return super().__call__(cls, **kwargs)


class ConstrainedToKeywords(metaclass=KeywordsOnlyMeta):

    def __init__(self, *args, **kwargs):
        print("args =", args)
        print("kwargs =", kwargs)
```

Even though the `__init__()` method in `ConstrainedToKeywords` accepts positional arguments through `*args`, execution never gets this far as non-empty positional argument lists are intercepted by `__call__()` in the `KeywordsOnlyMeta` metaclass, which causes a `TypeError` to be raised:

```
>>> from keywordmeta import *
>>>
>>> c = ConstrainedToKeywords(23, 45, 96, color='white')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "/Users/rjs/training/tmp/metaclass/keywordmeta.py", line 6, in __call__
    raise TypeError("Constructor for class {!r} does not accept positional arguments."
rmat(cls))
TypeError: Constructor for class <class 'keywordmeta.ConstrainedToKeywords'> does not a
ept positional arguments.
```

Constructor calls which contain only keyword arguments are permitted as designed:

```
>>> c = ConstrainedToKeywords(color='white')
args = (<class 'keywordmeta.ConstrainedToKeywords'>,)
kwargs = {'color': 'white'}
```

We’ve covered a lot of the theory and practice behind metaclasses. Now we’ll build on those ideas with some useful applications.

Practical metaclass examples

Python metaclasses can seem very…well…meta! So it’s time for a section on metaclasses which solve some actual problems you’ve probably encountered.

Preventing duplicate class attributes

At some point, you've probably run into the issue that duplicate class attributes names aren't flagged by Python as errors:

```
class Dodgy:

    def method(self):
        return "first definition"

    def method(self):
        return "second definition"
```

In fact, the second definition takes precedence because it overwrites the first entry in the namespace dictionary as the class definition is processed:

```
>>> from duplicatesmeta import *
>>> dodgy = Dodgy()
>>> dodgy.method()
'second definition
```

Let's write a metaclass which detects and prevents this unfortunate situation from occurring. To do this, rather than using a regular dictionary as the namespace object used during class construction, we need a dictionary which raises an error when we try to assign to an existing key.

Here is just such a dictionary, `OneShotDict`, which is implemented by specialising the built-in `dict` type and overriding the `__init__()` and `__setitem__()` methods. Note that the built-in `dict` has quite a sophisticated initializer which accepts many different forms of arguments, but something much simpler is sufficient in our case:

```
class OneShotDict(dict):

    def __init__(self, existing=None):
        super().__init__()
        if existing is not None:
            for k, v in existing:
                self[k] = v

    def __setitem__(self, key, value):
        if key in self:
            raise KeyError("Cannot assign to existing key {!r} "
                                 "in {!r}".format(key, type(self).__name__))
        super().__setitem__(key, value)
```

Before we use the `OneShotDict` in a metaclass, let's just check that it's working as expected:

```
>>> d = OneShotDict()
>>> d['A'] = 65
>>> d['B'] = 66
>>> d['A'] = 32
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "/Users/sixtynorth/sandbox/metaclasses/orderedmeta.py", line 14, in __setitem__
    raise KeyError("Cannot assign to existing key {!r} in {!r}".format(key, type(self)
name__))
KeyError: Cannot assign to existing key 'A' in 'OneShotDict'
```

We can now design a very simple metaclass which uses `OneShotDict` for the namespace object:

```
class ProhibitDuplicatesMeta(type):

    @classmethod
    def __prepare__(mcs, name, bases):
        return OneShotDict()
```

All we need to do is override the `__prepare__()` classmethod and returning an instance of our specialised dictionary.

If we try to define a class with duplicate methods using this metaclass, we get an error:

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 4, in Dodgy
  File "/Users/sixtynorth/sandbox/metaclasses/duplicatesmeta.py", line 12, in __setitem
    raise ValueError("Cannot assign to existing key {!r} in {!r}".format(key, type(sel
__name__))
ValueError: Cannot assign to existing key 'method' in 'OneShotDict'
```

The main shortcoming here is that the error message isn't hugely informative. Unfortunately, we don't have access to the part of the runtime machinery which reads our class definition and populates the dictionary, so we can't intercept the `KeyError` and emit a more useful error instead. The best we can do — rather than using a general purpose collection like `OneShotDict` — is to create a functional equivalent called something like `OneShotClassNamespace` with a more specific error message. This has the benefit that we can pass in additional diagnostic information, such as the name of the class currently being defined, into the namespace object on construction, which helps us emit a more useful message:

```
class OneShotClassNamespace(dict):

    def __init__(self, name, existing=None):
        super().__init__()
        self._name = name
        if existing is not None:
            for k, v in existing:
                self[k] = v

    def __setitem__(self, key, value):
        if key in self:
            raise TypeError("Cannot reassign existing class "
                            "attribute {!r} of {!r}".format(key, self._name))
        super().__setitem__(key, value)


class ProhibitDuplicatesMeta(type):

    @classmethod
    def __prepare__(mcs, name, bases):
        return OneShotClassNamespace(name)
```

After renaming `OneShotDict` to `OneShotClassNamespace` we adjust its initializer to accept a positional `name` argument which we store as an instance attribute `_name`. In the guard clause of `__setitem__` we change the exception type from `ValueError` to `TypeError` and edit the error message to make it both more germane and more informative. Lastly, we need to remember to forward the class name which is passed to `__prepare__()` to the `OneShotClassNamespace` constructor.

When we try to execute the module containing the class with the duplicate method definition, we get a much more useful error message “Cannot reassign existing class attribute ‘method’ of ‘Dodgy’”:

```
>>> from duplicatesmeta import *
Traceback (most recent call last):
  File "/Users/sixtynorth/sandbox/metaclasses/duplicatesmeta.py", line 38, in <module>
    class Dodgy(metaclass=ProhibitDuplicatesMeta):
  File "/Users/sixtynorth/sandbox/metaclasses/duplicatesmeta.py", line 43, in Dodgy
    def method(self):
  File "/Users/sixtynorth/sandbox/metaclasses/duplicatesmeta.py", line 27, in __setitem
    raise TypeError("Cannot reassign existing class attribute {!r} of {!r}".format(key,
elf._name))
TypeError: Cannot reassign existing class attribute 'method' of 'Dodgy'
```

Much better!

Naming Descriptors using Metaclasses

For our next metaclass example, let's return to the planet example we used in chapter 4 to illustrate descriptors. Here's a reminder of the code we ended up with:

```
from weakref import WeakKeyDictionary


class Positive:

    def __init__(self):
        self._instance_data = WeakKeyDictionary()

    def __get__(self, instance, owner):
        if instance is None:
            return self
        return self._instance_data[instance]

    def __set__(self, instance, value):
        if value <= 0:
            raise ValueError("Value {} is not positive".format(value))
        self._instance_data[instance] = value

    def __delete__(self, instance):
        raise AttributeError("Cannot delete attribute")


class Planet:

    def __init__(self,
                 name,
                 radius_metres,
                 mass_kilograms,
                 orbital_period_seconds,
                 surface_temperature_kelvin):
        self.name = name
        self.radius_metres = radius_metres
        self.mass_kilograms = mass_kilograms
        self.orbital_period_seconds = orbital_period_seconds
        self.surface_temperature_kelvin = surface_temperature_kelvin

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        if not value:
            raise ValueError("Cannot set empty Planet.name")
        self._name = value

    radius_metres = Positive()
    mass_kilograms = Positive()
    orbital_period_seconds = Positive()
    surface_temperature_kelvin = Positive()
```

Recall that we implemented a new descriptor type called `Positive` which would only admit positive numeric values. This saved a lot of boilerplate code in the definition of our `Planet` class, but we lost an important capability along the way, because there is no way for a descriptor instance to know to which class attribute it has been bound. One of the instances of `Positive` is bound to `Planet.radius_metres`, but it has no way of knowing that. The default Python machinery for processing class definitions just doesn't set up that association.

The shortcoming is revealed when we trigger a `ValueError` by trying to assign a non-positive value to one of the attributes. Here we try to give the planet Mercury a nonsensical negative mass:

```
>>> from planet import *
>>> mercury, venus, earth, mars = make_planets()
>>> mercury.mass_kilograms = -10000
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "/Users/rjs/training/tmp/metaclass/planet.py", line 16, in __set__
    raise ValueError("Value {} is not positive".format(value))
ValueError: Value -10000 is not positive
```

The error message doesn't — and in fact *can't* — tell us which attribute triggered the exception.

Now we'll show how we can modify the class creation machinery by defining metaclasses which should be able to intervene in the process of defining the `Planet` class in order to give descriptor instances the right name.

We'll start by introducing a new base-class for our descriptors called `Named`. This is very simple and just has `name` as a public instance attribute. The constructor defines a default value of `None` because we won't be in a position to assign the attribute value until after the descriptor object has been constructed:

```
class Named:

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

We'll modify our existing `Positive` descriptor so it becomes a subclass of `Named` and therefore gains the `name` attribute. Again the constructor arguments defines a default of `None`:

```
class Positive(Named):

    def __init__(self, name=None):
        super().__init__(name)
        self._instance_data = WeakKeyDictionary()

    def __get__(self, instance, owner):
        if instance is None:
            return self
        return self._instance_data[instance]

    def __set__(self, instance, value):
        if value <= 0:
            raise ValueError("Attribute value {} {} is not positive".format(self.name,
lue))
        self._instance_data[instance] = value

    def __delete__(self, instance):
        raise AttributeError("Cannot delete attribute {}".format(self.name))
```

We've modfied the argument list to `__init__()`, ensured that the superclass initialiser is called, and made use of the new `name` attribute in the error message raised by `__set__()` when we try to assign a non-positive number to the descriptor.

Now we need a metaclass which can detect the presence of descriptors which are `Named` and assign class attribute names to them:

```
class DescriptorNamingMeta(type):

    def __new__(mcs, name, bases, namespace):
        for name, attr in namespace.items():
            if isinstance(attr, Named):
                attr.name = name
        return super().__new__(mcs, name, bases, namespace)
```

Again, this is fairly straightforward. In `__new__` we iterate over the names and attributes in the namespace dictionary, and if the attribute is an instance of `Named` we assign the name of the current item to its public `name` attribute.

Having modified the contents of the namespace, we then call the superclass implementation of `__new__()` to actually allocate the new class object.

The only change we need to make to our `Planet` class is to refer to the metaclass on the opening line. There's not need for us to modify our uses of the `Positive` descriptor, the optional `name` argument will default to `None` when the class definition is read, before metaclass `__new__()` is invoked:

```
class Planet(metaclass=DescriptorNamingMeta):

    def __init__(self,
                 name,
                 radius_metres,
                 mass_kilograms,
                 orbital_period_seconds,
                 surface_temperature_kelvin):
        self.name = name
        self.radius_metres = radius_metres
        self.mass_kilograms = mass_kilograms
        self.orbital_period_seconds = orbital_period_seconds
        self.surface_temperature_kelvin = surface_temperature_kelvin

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        if not value:
            raise ValueError("Cannot set empty Planet.name")
        self._name = value

    radius_metres = Positive()
    mass_kilograms = Positive()
    orbital_period_seconds = Positive()
    surface_temperature_kelvin = Positive()
```

By trying to set a non-positive mass for the planet Mercury, we can see that each descriptor objects now knows the name of the attribute to which it has been bound, so can it emit *much* more helpful diagnostic message:

```
>>> from planet import *
>>> mercury, venus, earth, mars = make_planets()
>>> mercury.mass_kilograms
3.3022e+23
>>> mercury.mass_kilograms = -10000
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "/Users/sixtynorth/sandbox/metaclasses/planet_08.py", line 23, in __set__
    raise ValueError("Attribute value {} {} is not positive".format(self.name, value))
ValueError: Attribute value mass_kilograms -10000 is not positive
```

Metaclasses and Inheritance

We'll finish off this module by looking at how metaclasses interact with inheritance.

In `metainheritance.py` let's define two metaclasses related only by the fact that they both subclass `type`:

```
# metainheritance.py

class MetaA(type):
    pass


class MetaB(type):
    pass
```

We'll also define two regular classes, `A` and `B` which use the `MetaA` and `MetaB` as their respective metaclasses:

```
class A(metaclass=MetaA):
    pass


class B(metaclass=MetaB):
    pass
```

Now we'll introduce a third regular class `D` which derives from class `A`:

```
class D(A):
    pass
```

Now, let's introspect the *class* `D` itself — not an instance of `D` — to determine what its metaclass is:

```
>>> type(D)
<class 'metainheritance.MetaA'>
```

The metaclass of class `D` is `MetaA`, which was inherited from regular class `A`. So, metaclasses are inherited.

What happens if we try to create a new class `C` which inherits from both regular classes `A` and `B` with their different metaclasses? Let's give it a go:

```
class C(A, B):
    pass
```

When we try to execute this code by importing it at the REPL, we get a type error:

```
>>> from metainheritance import *
Traceback (most recent call last):
```

```
  File "<input>", line 1, in <module>
  File "/Applications/PyCharm.app/Contents/helpers/pydev/pydev_import_hook.py", line 21
in do_import
    module = self._system_import(name, *args, **kwargs)
  File "/Users/rjs/training/p4/courses/pluralsight/advanced-python/source/advanced-pyth
-m06-metaclasses/examples/metainheritance.py", line 22, in <module>
    class C(A, B):
TypeError: metaclass conflict: the metaclass of a derived class must be a (non-strict)
bclass of the metaclasses of all its bases
```

With the message "metaclass conflict: the metaclass of a derived class must be a (non-strict) subclass of the metaclasses of all its bases" Python is telling us that it doesn't know what to do with the unrelated metaclasses. Which metaclass `__new__()` should be used to allocate the class object?

To resolve this we need a single metaclass — let's call it `MetaC` — which we can create by inheriting from both `MetaA` and `MetaB`:

```
class MetaC(MetaA, MetaB):
    pass
```

In the definition of `C` we must override the metaclass to specify `MetaC`:

```
class C(A, B, metaclass=MetaC):
  pass
```

With these changes we can successfully import `C` and check that it's metaclass is `MetaC`:

```
>>> from metainheritance import *
>>> type(C)
<class 'metainheritance.MetaC'>
```

So we've persuaded Python to accept our code, but our metaclasses are empty so they combine trivially.

Sometimes, metaclasses will combine in straightforward ways. For example, our `ProhibitDuplicatesMeta` which overrides only `__prepare__()`, and our `KeyWordsOnlyMeta` which overrides only `__call__()`, can be combined into the conceptually simple (but horribly named) `ProhibitDuplicatesAndKeyWordsOnlyMeta`:

```
class ProhibitDuplicatesAndKeyWordsOnlyMeta(
    ProhibitDuplicatesMeta,
    KeyWordsOnlyMeta):
pass
```

Designing metaclasses for composition

To cooperate gracefully, non-trivial metaclasses must be designed with this in mind. This isn't always straightforward — or even possible — if the combination makes no sense. An important step in designing cooperative metaclasses is to diligently use `super()` when delegating to "base" classes, because as we learnt in chapter 8 of The Python Journeyman `super()` actually delegates to the next class in the Method Resolution Order (a.k.a. MRO) which accounts for multiple inheritance.

Even though both `TracingMeta` (defined earlier in this chapter) and `DescriptorNamingMeta` (which we used with the planet attributes) both override `__new__()`, they combine in either order because our implementations delegate to the next class in the MRO chain using calls to `super()`. In `planet.py` we can import `TracingMeta` and combine our two metaclasses into `TracingDescriptorNamingMeta` with multiple inheritance:

```
class TracingDescriptorNamingMeta(TracingMeta, DescriptorNamingMeta):
    pass
```

The `Planet` class definition is executed when we import the module. This is when the metaclasses to their work, and we can see that the tracing works as expected:

```
>>> from planet import *
TracingMeta.__prepare__(name, bases, **kwargs)
  mcs = <class 'tracing.TracingMeta'>
  name = TracingClass
  bases = ()
  kwargs = {}
<-- namespace = {}

TracingMeta.__new__(mcs, name, bases, namespace)
  mcs = <class 'tracing.TracingMeta'>
  name = TracingClass
  bases = ()
  namespace = {'__module__': 'tracing', '__qualname__': 'TracingClass', '__init__': <fu
tion TracingClass.__init__ at 0x1032c89d8>, '__new__': <function TracingClass.__new__ a
0x1032c8950>}
  kwargs = {}
<-- cls = <class 'tracing.TracingClass'>

TracingMeta.__init__(cls, name, bases, namespace)
  cls = <class 'tracing.TracingClass'>
  name = TracingClass
  bases = ()
  namespace = {'__module__': 'tracing', '__qualname__': 'TracingClass', '__init__': <fu
tion TracingClass.__init__ at 0x1032c89d8>, '__new__': <function TracingClass.__new__ a
```

```
0x1032c8950>}
  kwargs = {}
```

We can also confirm that the descriptor naming is working by causing a `ValueError` and checking for the descriptor name in the error message:

```
>>> mercury.mass_kilograms = -10000
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'mercury' is not defined
>>> mercury, venus, earth, mars = make_planets()
>>> mercury.mass_kilograms = -10000
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "/Users/rjs/training/planet_09.py", line 24, in __set__
    raise ValueError("Attribute value {} {} is not positive".format(self.name, value))
ValueError: Attribute value mass_kilograms -10000 is not positive
```

The two metaclasses cooperate successfully by using `super()`.

Prefer `__init__` to `__new__`

Another tip for designing metaclasses which cooperate well is to prefer to put code which *configures* the class object in `__init__()` (which is handed to the class object to be configured) rather than in `__new__()` (which is responsible for allocating the new class object).

If any of your cooperative metaclasses expect custom keyword arguments passed from the class statement, you’ll need to ensure that these are forwarded through the MRO chain. You also need to ensure that they are consumed, if necessary, so that subsequent classes in the MRO don’t receive unexpected arguments. All custom keyword arguments must have been consumed by the time `type.__new__()` is called, as it expects no additional arguments.

Ultimately though, if you’re dependent on metaclasses from a third-party framework such as *SQLAlchemy* or *Qt*, there’s a good chance that their metaclasses won’t compose gracefully. You should be able to figure out whether they will compose by reading their source code, if you have access to it.

Summary

We’ve covered a lot of ground in this chapter, and you should now know more than the majority of Python developers about the customisation of class creation using metaclasses.

- All classes have a metaclass which is the type of the class object. The default type of class objects is `type`.
- The metaclass is responsible for processing the class definition from parsing from the source code into a class object.
- The `__prepare__()` metaclass method must return a mapping object which the Python runtime will populate with namespace items collected from parsing and executing the class definition.
- The `__new__()` metaclass method must allocate and return a class object and configure it using the contents of the class namespace, the list of base classes passed from the definition, and any additional keyword arguments passed in the definition.
- The `__init__()` metaclass method can be used to further configure a class object, and must have the same signature as `__new__()`.
- The `__call__()` metaclass method in effect implements the *constructor* for class instances, and is invoked when we construct an instance.
- An important use case for metaclasses is to support so-called named descriptors, whereby we can configure descriptor objects such as properties with the name of the class attribute to which they are assigned.
- Strict rules control how muliple metaclasses interact in the presence of inheritance relationships between the regular classes which use them. Judicial metaclass design using `super()` to delegate via the MRO can yield metaclasses which compose gracefully.

Chapter 7 - Class decorators

In the last chapter we looked at metaclasses, and — while powerful — there's no doubt that metaclasses can be difficult to understand and reason about. Fortunately, Python supports an alternative which is sufficient for many cases where we want to customise classes at the point they are defined. This alternative is the *class decorator*.

It's worth bearing in mind that anything that can be achieved with a class decorator can also be achieved with a metaclass, although the reverse is not true. In other words, class decorators are less powerful than metaclasses although they are much easier to understand. As a result, class decorators should be preferred whenever the desired effect can be achieved with either a metaclass or a class decorator.

A first class decorator

Class decorators work in much the same way as function decorators: they apply a transformation to a class after the class definition body has been processed, but before the definition is bound to the name of the class.

Let's start with a very simple class, which is decorated with a function `my_class_decorator`:

```
@my_class_decorator
class Temperature:

    def __init__(self, kelvin):
        self._kelvin = kelvin

    def get_kelvin(self):
        return self._kelvin

    def set_kelvin(self, value):
        self._kelvin = value
```

Set aside for the moment that these getter and setter methods are hardly the most Pythonic solution; they serve our purpose for the time being.

To understand how class decorators work, `my_class_decorator` is a very simple function which simply iterates over and prints the attributes of the class object:

```
def my_class_decorator(cls):
    for name, attr in vars(cls).items():
        print(name)
    return cls
```

The class decorator must accept the class object as its only argument, (by convention called `cls`) and must return the modified class object, or an alternative class object. This class object will be bound to the class name given in the definition, in this case `Temperature`.

When we import our module into a REPL session, we can see that the decorator is executed when the `Temperature` class is first defined, which is when the module is first imported:

```
>>> from class_decorators import *
__weakref__
get_kelvin
set_kelvin
__dict__
__module__
__init__
```

Enforcing constraints with a class decorator

It’s important for our `Temperature` class to maintain an important invariant of the universe in which we live: that temperatures cannot be lower than absolute zero, which is zero kelvin. Rather than enforce that every method of our class honours this class invariant, we’ll use a class decorator to wrap every method in an invariant-checking proxy.

To do this in a generic way, we need to provide a way of specifying the class invariant to the class decorator. Recall from chapter 3 of “The Python Journeyman” that to do this we define a *decorator factory*. This factory is a function accepting the arguments we need and returning a decorator.

We will define a predicate function which describes the invariant. We’ll then pass that predicate to a function — a *class decorator factory*, if you will — which creates the actual class decorator that processes the definition of the

`Temperature` class. As the `Temperature` class is processed by the class decorator each callable member of the class is identified in turn (these are the methods of the `Temperature` class), and a *function* decorator is applied to each method. When invoked, the method decorators delegate to the original predicate function to verify the invariant. There's a lot going on here, so let's look at the code.

The invariant decorator factory

Here's our `invariant` function which returns a decorator:

```
def invariant(predicate):
    """Create a class decorator which checks a class invariant.

    Args:
        predicate: A callable to which, after every method invocation,
            the object on which the method was called will be passed.
            The predicate should evaluate to True if the class invariant
            has been maintained, or False if it has been violated.

    Returns:
        A class decorator for checking the class invariant tested by
        the supplied predicate function.
    """
    def invariant_checking_class_decorator(cls):
        """A class decorator for checking invariants."""

        method_names = [name for name, attr in vars(cls).items() if callable(attr)]
        for name in method_names:
            _wrap_method_with_invariant_checking_proxy(cls, name, predicate)

        return cls

    return invariant_checking_class_decorator


def _wrap_method_with_invariant_checking_proxy(cls, name, predicate):
    method = getattr(cls, name)
    assert callable(method)

    @functools.wraps(method)
    def invariant_checking_method_decorator(self, *args, **kwargs):
        result = method(self, *args, **kwargs)
        if not predicate(self):
            raise RuntimeError("Class invariant {!r} violated for {!r}".format(predica
__doc__, self))
        return result

    setattr(cls, name, invariant_checking_method_decorator)
```

The `invariant` function — which is in effect a decorator factory — accepts a predicate function which is used to test the invariant. It then returns the invariant checking *class decorator* function. The class decorator accepts the class object as `cls` and builds a list of method names by identifying the callable attributes. Each method is then processed by `_wrap_method_with_invariant_checking_proxy()` which creates a *function decorator* for each method. This function decorator calls the decorated method and then checks that the invariant predicate still holds. Note that we must use `setattr()` to update the class namespace rather than manipulating the class `__dict__` directly as the latter is not mutable.

Decorating `Temperature`

We can now define our `Temperature` class like this:

```
def not_below_absolute_zero(temperature):
    """Temperature not below absolute zero"""
    return temperature._kelvin >= 0


@invariant(not_below_absolute_zero)
class Temperature:

    def __init__(self, kelvin):
        self._kelvin = kelvin

    def get_kelvin(self):
        return self._kelvin

    def set_kelvin(self, value):
        self._kelvin = value
```

Our predicate is defined as a free function `not_below_absolute_zero()` which simply tests the `_kelvin` attribute. We supply this predicate to the `invariant()` class decorator which is applied to the `Temperature` class.

Testing out the decorated class

Creating non-negative temperatures works as expected:

```
>>> from class_decorators import *
>>> t = Temperature(5.0)
```

But an attempt to construct a negative temperature demonstrates that our class decorator has successfully wrapped `__init__()` with the function decorator which checks the invariant:

```
>>> t = Temperature(-1.0)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
```

```
  File "class_decorators.py", line 47, in invariant_checking_method_decorator
    raise RuntimeError("Class invariant {!r} violated for {!r}".format(predicate.__doc
 self))
RuntimeError: Class invariant 'Temperature not below absolute zero' violated for <class
ecorators.Temperature object at 0x103012940>
```

Likewise, the function decorator wraps `set_kelvin()`:

```
>>> s = Temperature(42.0)
>>> s.set_kelvin(-1.0)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "class_decorators.py", line 47, in invariant_checking_method_decorator
    raise RuntimeError("Class invariant {!r} violated for {!r}".format(predicate.__doc
 self))
RuntimeError: Class invariant 'Temperature not below absolute zero.' violated for <clas
decorators.Temperature object at 0x10302ca20>
```

Enforcing constraints for properties

Before we get too excited, let's see what happens if we introduce some properties, defined using the `@property` decorator, into our `Temperature` class:

```
@invariant(not_below_absolute_zero)
class Temperature:

    def __init__(self, kelvin):
        self._kelvin = kelvin

    def get_kelvin(self):
        return self._kelvin

    def set_kelvin(self, value):
        self._kelvin = value

    @property
    def celsius(self):
        return self._kelvin - 273.15

    @celsius.setter
    def celsius(self, value):
        self._kelvin = value + 273.15

    @property
    def fahrenheit(self):
        return self._kelvin * 9/5 - 459.67

    @fahrenheit.setter
    def fahrenheit(self, value):
        self._kelvin = (value + 459.67) * 5/9
```

We've added properties to get and set the temperature in Celsius and Fahrenheit:

```
>>> from class_decorators import *
>>> t = Temperature(42.0)
>>> t.celsius
-231.14999999999998
>>> t.celsius = -100
>>> t.celsius
-100.0
>>> t.celsius = -300
```

At this point we would have expected to have received a `RuntimeError` as -300 celsius is less than absolute zero kelvin. However, no exception is raised and the class invariant has been violated. If we try to *get* the kelvin value via the `get_kelvin()` method we do indeed get an error, but this is too late; class invariant violations should *never* be detected by non-mutating getters or other query methods. A breach in our defences has permitted the object to get into an invalid state.

How is our invariant check being circumvented?

Properties are descriptors

Recall that the `@property` decorator produces a *descriptor* which wraps the getter and setter methods, but because descriptors aren't callable functions they aren't being detected and wrapped by our invariant checking function decorator. To fix this, we need to beef up our machinery to detect and appropriately proxy these descriptors.

If we take a peek at `vars(Temperature)` we can see that `fahrenheit` and `celsius` correspond to `property` objects:

```
>>> from class_decorator import *
>>> from pprint import pprint as pp
>>> pp(vars(Temperature))
mappingproxy({'__dict__': <attribute '__dict__' of 'Temperature' objects>,
              '__doc__': None,
              '__init__': <function Temperature.__init__ at 0x1031bf598>,
              '__module__': 'class_decorators',
              '__weakref__': <attribute '__weakref__' of 'Temperature' objects>,
              'celsius': <property object at 0x1031bb138>,
              'fahrenheit': <property object at 0x1031bb188>,
              'get_kelvin': <function Temperature.get_kelvin at 0x1031bf158>,
```

Ideally, we'd detect any value in this mapping which is a descriptor.[11] Unfortunately for us in this case, descriptors are also used for all functions and methods — for reasons we won't go into even in this advanced book —

so they too would be detected. Instead, we'll go for the simpler approach of looking for instances of `property`.

Detecting and wrapping properties

Our updated class decorator factory function looks like this:

```
def invariant(predicate):
    """Create a class decorator which checks a class invariant.

    Args:
        predicate: A callable to which, after every method invocation,
            the object on which the method was called will be passed.
            The predicate should evaluate to True if the class invariant
            has been maintained, or False if it has been violated.

    Returns:
        A class decorator for checking the class invariant tested by
        the supplied predicate function.
    """
    def invariant_checking_class_decorator(cls):
        """A class decorator for checking invariants."""

        method_names = [name for name, attr in vars(cls).items() if callable(attr)]
        for name in method_names:
            _wrap_method_with_invariant_checking_proxy(cls, name, predicate)

        property_names = [name for name, attr in vars(cls).items() if isinstance(attr,
operty)]
        for name in property_names:
            _wrap_property_with_invariant_checking_proxy(cls, name, predicate)

        return cls

    return invariant_checking_class_decorator
```

We've inserted a new section into the body of the class decorator itself to search for and wrap any class attributes which are properties. This delegates to a new function called `_wrap_property_with_invariant_checking_proxy()`:

```
def _wrap_property_with_invariant_checking_proxy(cls, name, predicate):
    prop = getattr(cls, name)
    assert isinstance(prop, property)

    invariant_checking_proxy = InvariantCheckingPropertyProxy(prop, predicate)

    setattr(cls, name, invariant_checking_proxy)
```

This function retrieves the property from the class and passes it and the predicate function to the constructor of a new class, `InvariantCheckingPropertyProxy`. Finally, we use `setattr()` to replace the property with the proxy. The `InvariantCheckingPropertyProxy` is itself a descriptor:

```
class InvariantCheckingPropertyProxy:

    def __init__(self, referent, predicate):
        self._referent = referent
        self._predicate = predicate

    def __get__(self, instance, owner):
        if instance is None:
            return self
        result = self._referent.__get__(instance, owner)
        if not self._predicate(instance):
            raise RuntimeError("Class invariant {!r} violated for {!r}".format(
                self._predicate.__doc__, instance))
        return result

    def __set__(self, instance, value):
        result = self._referent.__set__(instance, value)
        if not self._predicate(instance):
            raise RuntimeError("Class invariant {!r} violated for {!r}".format(
                self._predicate.__doc__, instance))
        return result

    def __delete__(self, instance):
        result = self._referent.__delete__(instance)
        if not self._predicate(instance):
            raise RuntimeError("Class invariant {!r} violated for {!r}".format(
                self._predicate.__doc__, instance))
        return result
```

The initializer simply stores references to the referent property and the predicate function. The implementations of `__get__()`, `__set__()`, and `__delete__()` forward to the underlying referent property and then check that the class invariant of the instance remains inviolate. A quick test at the REPL shows that everything is working as designed:

```
>>> t = Temperature(42.0)
>>> t.celsius = -300
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "class_decorators.py", line 64, in __set__
    raise RuntimeError("Class invariant {!r} violated for {!r}".format(self._predicate
doc__, instance))
RuntimeError: Class invariant 'Temperature not below absolute zero' violated for <class
ecorators.Temperature object at 0x1030226d8>
>>> t.fahrenheit = 100
>>> t.celsius
37.77777777777783
```

Chaining class decorators

For a final demonstration, we’ll show that we can chain class decorators just as we can with function decorators:

```
def not_below_absolute_zero(temperature):
    """Temperature not below absolute zero"""
    return temperature._kelvin >= 0

def below_absolute_hot(temperature):
    """Temperature below absolute hot"""
    # See http://en.wikipedia.org/wiki/Absolute_hot
    return temperature._kelvin <= 1.416785e32

@invariant(below_absolute_hot)
@invariant(not_below_absolute_zero)
class Temperature:

    def __init__(self, kelvin):
        self._kelvin = kelvin

    def get_kelvin(self):
        return self._kelvin

    def set_kelvin(self, value):
        self._kelvin = value

    @property
    def celsius(self):
        return self._kelvin - 273.15

    @celsius.setter
    def celsius(self, value):
        self._kelvin = value + 273.15

    @property
    def fahrenheit(self):
        return self._kelvin * 9/5 - 459.67

    @fahrenheit.setter
    def fahrenheit(self, value):
        self._kelvin = (value + 459.67) * 5/9
```

We’ve added a second invariant to be maintained to ensure that the temperature is below the hypothetical value [‘absolute hot’](http://en.wikipedia.org/wiki/Absolute_hot). At the REPL we can see that both constraints are enforced when we call instance methods such as `set_kelvin()`:

```
>>> t = Temperature(37.5)
>>> t.set_kelvin(-300)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "class_decorators.py", line 39, in invariant_checking_method_decorator
    result = method(self, *args, **kwargs)
  File "class_decorators.py", line 41, in invariant_checking_method_decorator
    raise RuntimeError("Class invariant {!r} violated for {!r}".format(predicate.__doc_
 self))
RuntimeError: Class invariant 'Temperature not below absolute zero' violated for <class
```

```
ecorators.Temperature object at 0x103025780>
>>> t.set_kelvin(1e33)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "class_decorators.py", line 41, in invariant_checking_method_decorator
    raise RuntimeError("Class invariant {!r} violated for {!r}".format(predicate.__doc_
 self))
RuntimeError: Class invariant 'Temperature below absolute hot' violated for <class_deco
tors_4.Temperature object at 0x103025780>
```

Chained class decorators and properties

Our class decorator has no problem decorating the already decorated methods on its second invocation. The proxied properties are another story though. Although the lower bound check works as before:

```
>>> t.celsius = -300
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "class_decorators_4.py", line 64, in __set__
    raise RuntimeError("Class invariant {!r} violated for {!r}".format(self._predicate
doc__, instance))
RuntimeError: Class invariant 'Temperature not below absolute zero' violated for <class
ecorators_4.Temperature object at 0x103025780>
```

However, the upper bound does *not* work:

```
>>> t.celsius = 1e34
>>>
```

The problem here is that our class decorator is detecting specifically `property` instances with this fragment:

```
property_names = [name for name, attr in vars(cls).items() if isinstance(attr, property
for name in property_names:
    _wrap_property_with_invariant_checking_proxy(cls, name, predicate)
```

However, our `InvariantCheckingPropertyProxy` does not satisfy the `isinstance()` check, so our proxy which enforces the `not_below_absolute_zero` invariant is applied to the genuine `property` although the `below_absolute_hot` proxy is not applied.

For a solution to this problem we'll use Python's abstract base-class mechanism, a topic we'll explore in the next chapter.

Summary

- Class decorators provide a simpler alternative to metaclasses for processing class definitions prior to the definition being bound to the class name.
- Remember, though, that class decorators are less powerful than metaclasses.
- Use class decorators when you can, and metaclasses when you must.
- Class decorators can be chained just like function decorators.

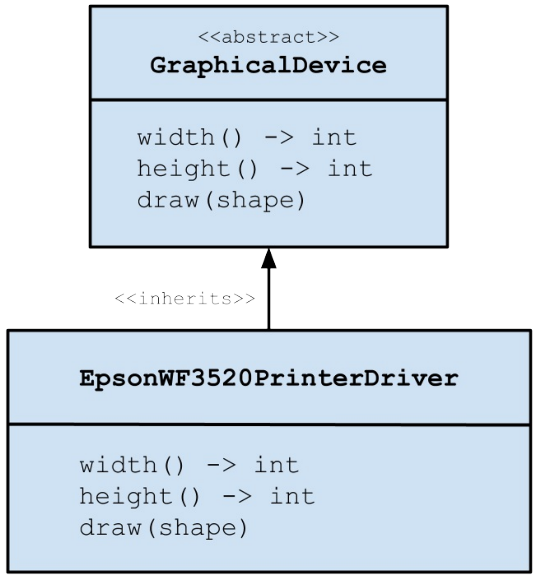
Chapter 8 - Abstract Base Classes

In this chapter we'll be investigating the abstract base-class mechanism in Python as originally defined in PEP 3119. In *The Python Journeyman* we used some abstract base classes, such as `collections.abc.Sequence`, when implementing a sorted-set collection type. This time, we'll look at the tools provided in the Python Standard Library `abc` module for creating abstract base-classes of your own design.

If you're coming to Python from another object-oriented language such as Java, C++, or C# you may have preconceived ideas of what an abstract base class is, and how to use one. Beware though! The abstract base-class mechanism in Python is much more flexible and can work in what may seem to be very surprising ways, so pay attention!

What is an abstract base-class?

Thinking for a moment beyond the confines of Python, what is, in general, an abstract base-class? The clue is in the name: *base* refers to the fact that the class is intended to be the target of an inheritance relationship; that is, we expect another class to derive from the base-class. For instance, `GraphicalDevice` here is intended as the base-class for other classes such as our `EpsonWF3520PrinterDriver`.



Graphical device inheritance

Abstract refers to the fact that the class cannot be instantiated in isolation; that is, it makes no sense to create an object of the type of the base-class alone. It only makes sense to instantiate the class as part of an object of a derived type. Ideally, it should should not be *possible* to instantiate an abstract base-class directly. The opposite of abstract is *concrete*, and in this example the printer driver is a concrete class, so it makes sense to instantiate it.

Why are abstract base-classes useful?

The rationale for any abstract base-class is to define an interface which derived classes *must* implement. This allows client code to be written against the base-class interface. In this example the printer driver must override the three abstract methods of `GraphicalDevice`. Done diligently, this leads to a highly desirable property of class hierarchies called *Liskov Substitutability*, a design principle which states that subclasses, from the point of view of client code, should be interchangeable. In other words, client code developed against an abstract interface should not require knowledge of specific concrete types, only of the capabilities as promised by the abstract base-class. So, for example, code written against the interface of `GraphicalDevice` should be able to render to an Epson WF3520 printer, or an 1080p LCD

Display without modification — we can substitute one concrete class for another.

Abstract base-classes differ from pure interfaces such as those we have in languages like Java, insofar as they can also contain implementation code which is to be shared by all derived classes.

What about duck typing?

Why do we need to define named interfaces when we have duck typing? Isn't it sufficient to know whether a particular object responds to the interface of a duck, and behaves like a duck, without actually knowing that it *is* a duck?

In Python, this is true both in theory and practice, but determining whether a particular object supports the required interface in advance of exercising that interface can be quite awkward. For example, what does it mean in Python to be a *mutable sequence*? We know that `list` is a mutable sequence, but we cannot assume that all mutable sequences are lists. In fact, the mutable sequence protocol requires at least sixteen methods are implemented. When relying on duck-typing it can be difficult to be sure that you've met the requirements, and if clients *do* need to determine whether a particular object is admissable as a mutable sequence with a look-before-you-leap approach, the check is messy and awkward to perform robustly.

Abstract base-classes in Python

Abstract base-classes in Python serve two purposes: First of all they provide a mechanism for defining protocols or interfaces, and ensuring that implementers of those protocols meet some minimum requirements. Secondly they provide a means for easily determining whether an arbitrary class or instance meets the requirements of a specific protocol.

For instance, we can determine that `list` is mutable sequence using the built-in `issubclass()` function:

```
>>> from collections.abc import MutableSequence
>>> issubclass(list, MutableSequence)
True
```

This much may not be surprising, but let's look at the base-class of `list`. In fact, we'll look at the transitive base-classes of list by examining it's method resolution order using the `__mro__` attribute:

```
>>> list.__mro__
(<class 'list'>, <class 'object'>)
```

This reveals that `list` has only one base-class, `object`, and that `MutableSequence` is nowhere to be seen. Further reflection — if you'll excuse the pun — might lead you to wonder how it is that such a fundamental type as Python's `list` can be a subclass of a type defined in a *library* module.

We've started out with this curious example so as to efficiently disabuse migrants from other programming languages of any existing notions of what abstract base-classes are, how they work, how they are used, and why they are useful! That done, we will dig further into the mechanism.

Abstract base-classes are indeed abstract

Let's establish that `MutableSequence` is indeed abstract, by attempting to directly instantiate it:

```
>>> ms = MutableSequence()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class MutableSequence with abstract methods __del
em__, __getitem__, __len__, __setitem__, insert
```

This fails with a useful `TypeError` explaining the five methods we need to implement the *Mutable Sequence* protocol. The reason we don't need to implement all sixteen is that eleven of them can be implemented in terms of the other five, and the `MutableSequence` abstract base-class contains code to do exactly this. Note, however, that these implementations may not be the most efficient since they can't exploit knowledge of the concrete class, but must instead work entirely through the interface of the abstract class.

Defining subclasses with `__subclasscheck__()`

Now to the question of how `issubclass(list, MutableSequence)` returns `True`. What happens is that when we call `issubclass(list, MutableSequence)` the built-in `issubclass()` function checks for the existence of a method called `__subclasscheck__()` on the *type* of `MutableSequence` (which is to say, on the metaclass of `MutableSequence`), and if it is present it is called with the subclass `list` as an argument. In other wordst:

```
issubclass(list, MutableSequence)
```

is roughly equivalent to:

```
if hasattr(type(MutableSequence), '__subclasscheck__'):
    return type(MutableSequence).__subclasscheck__(list)
# normal issubclass() behaviour...
```

It's up to the metaclass of `MutableSequence` to determine whether or not `list` is a subclass of `MutableSequence`, rather than `list` being required to know that `MutableSequence` is one of its base-classes. This allows `list` to be a subclass of `MutableSequence` *without* `MutableSequence` being a superclass of list (in the normal sense of being the target of an inheritance relationship). This unusual asymmetry between the superclass and subclass relationships in Python can be pretty mind-boggling if you're coming from, say, Java, but it's also incredibly powerful. We describe such base-classes as *virtual* base-classes, which is nothing at all to do with an identically named concept in C++.

A simple example

The `__subclasscheck__()` method on the metaclass of the virtual base class can do pretty much anything it likes to determine whether its argument is to be considered a subclass. Consider the code in `weapons.py`:

```
class SwordMeta(type):
    def __subclasscheck__(cls, sub):
        return (hasattr(sub, 'swipe') and callable(sub.swipe)
                and
                hasattr(sub, 'sharpen') and callable(sub.sharpen))


class Sword(metaclass=SwordMeta):
    pass


class BroadSword:

    def swipe(self):
        print("Swoosh!")
```

```
    def sharpen(self):
        print("Shink!")


class SamuraiSword:

    def swipe(self):
        print("Slice!")

    def sharpen(self):
        print("Shink!")


class Rifle:

    def fire(self):
        print("Bang!")
```

In this module we have defined a `Sword` class with a metaclass `SwordMeta`. `SwordMeta` defines the `__subclasscheck__()` method to check for the existence of callable `swipe` and `sharpen` attributes on the class. In this situation `Sword` will play the role of a virtual base-class. A few simple tests at the REPL confirm that `BroadSword` and `SamuraiSword` are indeed considered subclasses of `Sword` even though there is no explicit relationship through inheritance:

```
>>> issubclass(BroadSword, Sword)
True
>>> issubclass(SamuraiSword, Sword)
True
>>> issubclass(Rifle, Sword)
False
```

`__instancecheck__()`

This isn't the whole story though, as tests of *instances* using `isinstance()` will return inconsistent results:

```
>>> samurai_sword = SamuraiSword()
>>> isinstance(samurai_sword, Sword)
False
```

This is because the `isinstance()` machinery checks for the existence of the `__instancecheck__()` metamethod which we have not yet implemented. Let's do so now:

```
class SwordMeta(type):

  def __instancecheck__(cls, instance):
      return issubclass(type(instance), cls)
```

```
    def __subclasscheck__(cls, sub):
        return (hasattr(sub, 'swipe') and callable(sub.swipe)
                and
                hasattr(sub, 'sharpen') and callable(sub.sharpen))
```

Our `__instancecheck__()` implementation simply delegates to `issubclass()`. With this change in place, our call to `isinstance()` produces a result consistent with the result from `issubclass()`:

```
>>> samurai_sword = SamuraiSword()
>>> isinstance(samurai_sword, Sword)
True
```

This surprising technique is used in Python for some of the collection abstract base-classes, including `Sized`:

```
>>> from collections.abc import Sized
>>>
>>> class SizedCollection:
...     def __init__(self, size):
...         self._size = size
...     def __len__(self):
...         return self._size
...
>>>
>>> issubclass(SizedCollection, Sized)
True
```

After importing `Sized` from `collections.abc` we define a new class `SizedCollection` which is *not* related to `Sized` through inheritance. `SizedCollection` stores an integer size and has a `__init__()` to initialize the size, and `__len__()` to allow the size to be retrieved with the built-in `len()` function.

In this case, implementing a `__len__()` method is sufficient to be considered a subclass of the `Sized` abstract base-class.

It's worth bearing in mind that our implementations of `SwordMeta.__instancecheck__()` and `SwordMeta.__subclasscheck__()` are somewhat naïve, as they make no attempt to check the regular, non-virtual base-classes of the objects being tested. This could lead to some surprising behaviour. Bear in mind that correctly overriding `__subclasscheck__()` and `__instancecheck__()` on your own metaclasses is difficult. Don't worry though, we'll be presenting some more digestible alternatives shortly.

Non-transitivity of subclass relationships

Overriding `__subclasscheck__()` affords class implementers a great deal of flexibility. So much flexibility in fact that not only should you not expect symmetry between the superclass and subclass relationships, you shouldn't expect transitivity of the subclass relationship. What this means is that if C is a subclass of B, and B is subclass of A, it doesn't necessarily follow that C is a subclass of A.

One glaring example from Python revolves around the `Hashable` virtual base-class from `collections.abc`:

```
>>> from collections.abc import Hashable
>>> issubclass(object, Hashable)
True
>>> issubclass(list, object)
True
>>> issubclass(list, Hashable)
False
```

This occurs because `list` — which, remember, is a mutable collection — disables hashing by removing `__hash__()`. This method would otherwise be inherited from `object`, and the `Hashable` abstract base-class checks for it through its `__subclasscheck__()`:

```
>>> object.__hash__
<slot wrapper '__hash__' of 'object' objects>
>>>
>>> list.__hash__
>>>
>>> list.__hash__ is None
True
```

Some further investigation reveals that the `list` class sets the `__hash__` attribute to `None`. The `Hashable.__subclasscheck__()` implementation checks for this eventuality and uses it to signal non-hashability.

This example is also interesting because it demonstrates the fact that even the ultimate *base-class* `object` can be considered a subclass of `Hashable` — underlying the lack of symmetry between superclass and subclass relationships in Python.

Method resolution and virtual base-classes

It’s worth bearing in mind that, unlike regular base-classes, virtual base-classes don’t play a role in method resolution. We’ll demonstrate this by adding a `thrust()` method to the `Sword` virtual base-class:

```
class Sword(metaclass=SwordMeta):

    def thrust(self):
        print("Thrusting...")
```

Attempts to invoke this method on subclasses raise an `AttributeError`, and checking further, we can see that `Sword` is not present in the MRO for `BroadSword`:

```
>>> broad_sword = BroadSword()
>>> isinstance(broad_sword, Sword)
True
>>> broad_sword.swipe()
Swipe!
>>> broad_sword.thrust()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'BroadSword' object has no attribute 'thrust'
>>>
>>> BroadSword.__mro__
(<class 'BroadSword'>, <class 'object'>)
```

For this reason, it’s not possible to call virtual base-class methods using `super()` since `super()` works by searching the MRO.

Library support for abstract base-classes

We’ve pointed out that implementing `__subclasscheck__()` and `__instancecheck__()` correctly can be awkward to get right. Fortunately, the standard library provides support for implementing abstract base-classes with the `abc` module — and particularly the `ABCMeta` metaclass — along with some other useful pieces of infrastructure including the `ABC` base-class and the `@abstractmethod` decorator. We’ll cover each of these in detail now.

The ABCMeta metaclass

The `ABCMeta` metaclass implements reliable `__subclasscheck__()` and `__instancecheck__()` methods along with some other handy capabilities we’ll come to shortly. We’ll eventually use `ABCMeta` in our `Sword` example, allowing us to dispose of `SwordMeta`. But first we're going to cannibalise `ABCMeta` so that we can see how it works:

```
from abc import ABCMeta

class Sword(metaclass=ABCMeta):
    pass
```

`ABCMeta` doesn't, of course, know what it means to be a sword, so the test that was previously in `SwordMeta.__subclasscheck__()` needs to be relocated elsewhere. The `ABCMeta.__subclasscheck__()` method calls the special `__subclasshook__()` method on our actual class to perform the test.

In fact, all Python objects have the `__subclasshook__()` classmethod which accepts the potential subclass as its only argument. The method should return `True`, `False`, or `NotImplemented`. We can see this by calling it on `object`:

```
>>> object.__subclasshook__()
NotImplemented
```

For classes where `__subclasshook__()` returns `NotImplemented` the subclass test continues with the usual mechanism of testing the non-virtual base-classes. Boolean return values, on the other hand, definitively indicate whether the argument class is to be considered a subclass of the base-class or not.

Let's try implementing `__subclasshook__()` for our `Sword` class with pretty much the same definition we used for `__subclasscheck__()` previously:

```
class Sword(metaclass=ABCMeta):

    @classmethod
    def __subclasshook__(cls, sub):
        return (hasattr(sub, 'swipe') and callable(sub.swipe)
                and
                hasattr(sub, 'sharpen') and callable(sub.sharpen))
```

This class method, in conjunction with `ABCMeta`, is sufficient to make `issubclass()` and `isinstance()` work:

```
>>> issubclass(SamuraiSword, Sword)
True
>>> issubclass(Rifle, Sword)
False
>>> broad_sword = BroadSword()
```

```
>>> isinstance(broad_sword, Sword)
True
```

`SamuraiSword` is a subclass of `Sword`. `Rifle` is not a subclass of `Sword`, and an instance of `BroadSword` is an instance of a `Sword`.

Virtual subclass registration

It's also possible to directly register a class as a virtual subclass of an abstract base-class whose metaclass is `ABCMeta` by using the `register()` metamethod. For example, let's create an abstract base-class `Text` and register the built-in type `str` as a virtual subclass:

```
>>> class Text(metaclass=ABCMeta):
...     pass
...
>>> Text.register(str)
<class 'str'>
```

Notice that the `register()` metamethod returns the class which was registered — a point we'll return to in a moment. Now we can even retrofit base-classes (albeit virtual ones) to the built-in types:

```
>>> issubclass(str, Text)
True
>>> isinstance("Is this text?", Text)
True
```

Here were demonstrate that the built-in `str` class is now considered a subclass of our our `Text` class defined at the REPL, and `str` objects are instances of `Text`.

Using `register` as a decorator

Because the `register()` metamethod returns its argument, we can even use it as a class decorator. Let's register a class `Prose` as a virtual subclass of `Text`:

```
>>> @Text.register
... class Prose:
...     pass
...
>>> issubclass(Prose, Text)
True
```

Combining subclass registration with `__subclasshook__()`

You need to take care when combining virtual subclass registration with the `__subclasshook__()` technique because the result of `__subclasshook__()` takes precedence over the subclass registry. Returning `True` or `False` from `__subclasshook__()` is taken as a definite answer. If you also want registration to be accounted for you should return `NotImplemented` to indicate "not sure".

To see this in action, let's add a `LightSaber` which has no `sharpen()` method to our example. This class won't satisfy the `__subclasshook__()` test we defined in `Sword`, but we still want it identified as a virtual subclass of `Sword`, so we've registered it using the decorator form of `Sword.register`:

```
@Sword.register
class LightSaber:

    def swipe(self):
        print("Ffffkrrrrshhzzzwooooom..woom..woooom..")
```

Even though we've registered `LightSaber` with `Sword` the subclass test returns `False`:

```
>>> issubclass(LightSaber, Sword)
False
```

To fix this, we need to ensure that `__subclasshook__()` never returns `False` because doing so causes the `__subclasscheck__()` implementation in `ABCMeta` to skip the check for registered subclasses. Instead, in the case of a negative result, we should return `NotImplemented`:

```
class Sword(metaclass=ABCMeta):

    @classmethod
    def __subclasshook__(cls, sub):
        return ((hasattr(sub, 'swipe') and callable(sub.swipe)
                 and
                 hasattr(sub, 'sharpen') and callable(sub.sharpen))
                or NotImplemented)
```

With this change in place — which exploits shortcut evaluation of the logical operators — subclass detection now works as expected for implicitly detected subclasses, explicitly registered subclasses, and non-subclasses:

```
>>> issubclass(BroadSword, Sword)
True
>>> issubclass(LightSaber, Sword)
```

```
True
>>> issubclass(Rifle, Sword)
False
```

Bear in mind that this somewhat contrived example is designed to demonstrate that you should take care with subclass registration. How useful is our virtual base-class `Sword` now? The answer is “not very”. Being an instance of `Sword` is no longer a useful predicate for the object in question since we can’t guarantee the presence of the `sharpen()` method, which was — if you’ll excuse the pun — the whole point of the sword.

The ABC convenience base-class

The `abc` module contains a class `ABC` which is simply a regular class that has `ABCMeta` as its metaclass:

```
class ABC(metaclass=ABCMeta):
    """Helper class that provides a standard way to create an ABC using
    inheritance.
    """
    pass
```

This makes it even easier to declare abstract base-classes without having to put the metaclass mechanism on show. This may be an advantage when coding for audiences who haven’t been exposed to the concept of metaclasses. Using `ABC` our `Sword` class becomes:

```
from abc import ABC


class Sword(ABC):

    @classmethod
    def __subclasshook__(cls, sub):
        return ((hasattr(sub, 'swipe') and callable(sub.swipe)
                 and
                 hasattr(sub, 'sharpen') and callable(sub.sharpen))
                or NotImplemented)
```

The `abstractmethod` decorator

Finally, we reach the aspect of Python’s abstract base-classes which most immediately springs to mind when we talk about abstract base-classes in general: The ability to declare *abstract methods*.

An abstract method is a method which is declared but which doesn't necessarily have a useful definition. Abstract methods *must* be overridden in derived concrete classes, and the presence of an abstract method will prevent its host class being instantiated.

Abstract methods should be decorated with the `@abstractmethod` decorator, and their abstractness will only be enforced if the metaclass of the host class is `ABCMeta`:

```
from abc import (ABC, abstractmethod)

class AbstractBaseClass(ABC):  # metaclass is ABCMeta

    @abstractmethod
    def an_abstract_method(self):
        raise NotImplementedError  # Method body syntactically required.
```

Let's add abstract methods for swiping and thrusting to our `Sword` abstract base-class. We'll also update `__subclasshook__()` to match:

```
from abc import ABC, abstractmethod


class Sword(ABC):

    @classmethod
    def __subclasshook__(cls, sub):
    return ((hasattr(sub, 'swipe') and callable(sub.swipe)
             and
             hasattr(sub, 'thrust') and callable(sub.thrust)
             and
             hasattr(sub, 'parry') and callable(sub.parry)
             and
             hasattr(sub, 'sharpen') and callable(sub.sharpen))
            or NotImplemented)

    @abstractmethod
    def swipe(self):
        raise NotImplementedError

    @abstractmethod
    def thrust(self):
        print("Thrusting...")

    @abstractmethod
    def parry(self):
        raise NotImplementedError
```

Python syntax requires that we provide an implementation for each method, so for `swipe()` and `parry()` we raise a `NotImplementedError` but for `thrust()` we provide a default behaviour of printing "Thrusting!".

> We’ll take this opportunity to remind you of the distinction between `NotImplemented` and `NotImplementedError`. `NotImplemented` is a value returnable from predicate functions which are unable to make a determination of `True` or `False`. On the other hand, `NotImplementedError` is an exception type to be raised in place of missing code.

We’ll now make `BroadSword` a *real* subclass of `Sword`, by explicitly adding `Sword` to the list of base-classes:

```
class BroadSword(Sword):

    def swipe(self):
        print("Swipe!")

    def sharpen(self):
        print("Shink!")
```

We cannot instantiate `BroadSword` as it too is still abstract:

```
>>> broad_sword = BroadSword()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: Can't instantiate abstract class BroadSword with abstract methods parry, thr
```

We must implement all abstract methods for the class to be considered concrete:

```
class BroadSword(Sword):

    def swipe(self):
        print("Swoosh!")

    def thrust(self):
        super().thrust()

    def parry(self):
        print("Parry!")

    def sharpen(self):
        print("Shink!")
```

Notice that in `thrust()` we call the implementation provided in the abstract class, using `super()`:

```
>>> broad_sword = BroadSword()
>>> broad_sword.swipe()
Swipe!
```

Note however, that the requirement to implement abstract methods doesn't extend to *virtual* subclasses, only to *real* subclasses. We can still instantiate `SamuraiSword` successfully, even though it doesn't implement any of the abstract methods:

```
>>> samurai_sword = SamuraiSword()
>>> samurai_sword
<SamuraiSword object at 0x103022160>
```

Combining abstractmethod with other decorators

What if you need to combine `@abstractmethod` with other decorators? The `@abstractmethod` decorator can be combined with the `@staticmethod`, `@classmethod`, and `@property` decorators, although care must be taken that `@abstractmethod` is the *innermost* decorator. For properties you can independently mark the getters and setters as abstract:

```
class AbstractBaseClass(ABC):

    @staticmethod
    @abstractmethod
    def an_abstact_static_method():
        raise NotImplementedError

    @classmethod
    @abstractmethod
    def an_abstract_class_method(cls):
        raise NotImplementedError

    @property
    @abstractmethod
    def an_abstract_property(self):
        raise NotImplementedError

    @an_abstract_property.setter
    @abstractmethod
    def an_abstract_property(self, value):
        raise NotImplementedError
```

Recall the properties are implemented using descriptors. When implementing your own descriptors you need to do a little extra work to ensure your descriptor plays nicely with `@abstractmethod`. Let's look at that now.

Propagating abstractness through descriptors

Any descriptors which are implemented in terms of abstract methods should identify themselves as abstract by exposing a `__isabstractmethod__` attribute which should evaluate to `True`:

```python
class MyDataDescriptor(ABC):

    @abstractmethod
    def __get__(self, instance, owner):
        # ...
        pass

    @abstractmethod
    def __set__(self, instance, value):
        # ...
        pass

    @abstractmethod
    def __delete__(self, instance):
        # ...
        pass

    @property
    def __isabstractmethod__(self):
        return True  # or False if not abstract
```

This `__isabstractmethod__` attribute can either be implemented as a class attribute of the descriptor class, or it can itself be implemented as a property, as we have done here. Implementing it as a property is useful in cases where abstractness needs to be determined at runtime, as we'll see in an example shortly.

Let's see this in action with a very simple example. We'll define a class called `AbstractBaseClass` which inherits from `ABC`. Within this class we'll define two properties called `abstract_property` and `concrete_property` using the appropriate combinations of decorators:

```python
>>> from abc import (ABC, abstractmethod)
>>> class AbstractBaseClass(ABC):
...     @property
...     @abstractmethod
...     def abstract_property(self):
...         raise NotImplementedError
...     @property
...     def concrete_property(self):
...         return "sand, cement, water"
...
>>>
```

By querying the descriptor objects created by the property decorators we can inspect the `__isabstractmethod__` attributes of the two properties, and see that they have been marked as abstract and non-abstract (or concrete) as necessary:

```
>>> AbstractBaseClass.abstract_property.__isabstractmethod__
True
>>> AbstractBaseClass.concrete_property.__isabstractmethod__
False
```

This happens because the `__isabstractmethod__` flag is inspected by the `ABCMeta` implementation when the metaclass creates the actual class object which hosts the property.

So much for the theory, let's see this in practice.

Fixing our `@invariant` class decorator with ABCs

We rounded off chapter 7 on class decorators by building a class decorator for checking class invariants after every method call and property access. This worked fine for both methods and properties with a single application of the decorator, but with chained `@invariant` decorators the checking didn't work as planned for properties; only the innermost `@invariant` was taking effect. Let's recap.

Although the innermost lower bound check works:

```
>>> t = Temperature(42)
>>> t.celsius = -300
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "class_decorators_4.py", line 64, in __set__
    raise RuntimeError("Class invariant {!r} violated for {!r}".format(self._predicate
doc__, instance))
RuntimeError: Class invariant 'Temperature not below absolute zero' violated for <class
ecorators_4.Temperature object at 0x103025780>
```

Violating the upper bound is not detected as we had intended:

```
>>> t.celsius = 1e34
>>>
```

The problem here is that our class decorator is detecting specifically `property` instances with this fragment:

```
property_names = [name for name, attr
                  in vars(cls).items()
                  if isinstance(attr, property)]
for name in property_names:
    _wrap_property_with_invariant_checking_proxy(cls, name, predicate)
```

Because our property wrappers of type `InvariantCheckingPropertyProxy` are not detected as instances of `property` they are not wrapped a second time, and the invariant specified in the outermost decorator is not enforced.

Abstract base-classes to the rescue

We promised to use abstract base-classes to fix this problem, so let's go! We'll introduce a new abstract base-class called `PropertyDataDescriptor` which inherits from the `ABC` convenience class and which contains three abstract methods which define the data descriptor protcol. It also includes the abstract property `__isabstractmethod__()` for correct propagation of abstractness:

```
class PropertyDataDescriptor(ABC):

    @abstractmethod
    def __get__(self, instance, owner):
        raise NotImplementedError

    @abstractmethod
    def __set__(self, instance, value):
        raise NotImplementedError

    @abstractmethod
    def __delete__(self, instance):
        raise NotImplementedError

    @property
    @abstractmethod
    def __isabstractmethod__(self):
        raise NotImplementedError
```

Note that becuase `__isabstractmethod__` needs to look like an abstract *attribute*, we have implemented it by applying the `@abstractmethod` and `@property` decorators in that order.

Having defined an abstract *base* class, we now need some subclasses. The first will be a *virtual* subclass — the built-in `property` class — which we'll register with the base-class:

```
PropertyDataDescriptor.register(property)
```

The second subclass will be a real subclass. We'll modify our existing property proxy `InvariantCheckingPropertyProxy` to inherit from

`PropertyDataDescriptor`, which will also require that we override the `__isabstractmethod__` property:

```
class InvariantCheckingPropertyProxy(PropertyDataDescriptor):

    def __init__(self, referent, predicate):
        self._referent = referents
        self._predicate = predicate

    def __get__(self, instance, owner):
        if instance is None:
            return self
        result = self._referent.__get__(instance, owner)
        if not self._predicate(instance):
    raise RuntimeError("Class invariant {!r} violated for {!r}".format(
                            self._predicate.__doc__, instance))
        return result

    def __set__(self, instance, value):
        result = self._referent.__set__(instance, value)
        if not self._predicate(instance):
    raise RuntimeError("Class invariant {!r} violated for {!r}".format(
                            self._predicate.__doc__, instance))
        return result

    def __delete__(self, instance):
        result = self._referent.__delete__(instance)
        if not self._predicate(instance):
    raise RuntimeError("Class invariant {!r} violated for {!r}".format(
                            self._predicate.__doc__, instance))
        return result

    @property
    def __isabstractmethod__(self):
        return self._referent.__isabstractmethod__
```

Finally, we need to update the search-and-wrap logic in `invariant_checking_class_decorator()` to use the more general test for instances of `PropertyDataDescriptor` rather than the more specific test for just `property`.

```
def invariant(predicate):
    """Create a class decorator which checks a class invariant.

    Args:
        predicate: A callable to which, after every method invocation,
            the object on which the method was called will be passed.
            The predicate should evaluate to True if the class invariant
            has been maintained, or False if it has been violated.

    Returns:
        A class decorator for checking the class invariant tested by
        the supplied predicate function.
    """
    def invariant_checking_class_decorator(cls):
        """A class decorator for checking invariants."""
```

```
        method_names = [name for name, attr
                        in vars(cls).items()
                        if callable(attr)]
        for name in method_names:
            _wrap_method_with_invariant_checking_proxy(cls, name, predicate)

        property_names = [name for name, attr
                          in vars(cls).items()
                          if isinstance(attr, PropertyDataDescriptor)]
        for name in property_names:
            _wrap_property_with_invariant_checking_proxy(cls, name, predicate)

        return cls

    return invariant_checking_class_decorator
```

With these changes in place both invariants are enforced on property writes:

```
>>> t = Temperature(42)
>>>
>>> t.celsius = -300
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "class_decorators.py", line 86, in __set__
    result = self._referent.__set__(instance, value)
  File "class_decorators.py", line 88, in __set__
    raise RuntimeError("Class invariant {!r} violated for {!r}".format(self._predicate
doc__, instance))
RuntimeError: Class invariant 'Temperature not below absolute zero' violated for <class
ecorators.Temperature object at 0x103012cc0>
>>>
>>> t.celsius = 1e34
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "class_decorators.py", line 88, in __set__
    raise RuntimeError("Class invariant {!r} violated for {!r}".format(self._predicate
doc__, instance))
RuntimeError: Class invariant 'Temperature below absolute hot' violated for <class_deco
tors.Temperature object at 0x103012cc0>
```

We create a temperature of 42 kelvin, then attempt to modify the temperature through the `celsius` setter with a temperature of -300 celsius (which is below absolute zero). This now fails as designed, signalling violation of the ‘Temperature not below absolute zero’ invariant. Then we test the other invariant, by assigning a temperature of 1 × 10^34 through the `celsius` setter. This also fails as designed, signalling violation ‘Temperature below absolute hot’ class invariant.

There’s a lot going on in this code with decorators, metaclasses, abstract base-classes, and descriptors, and it may seem somewhat complicated. All this complexity is well encapsulated in the `invariant` class decorator, however, so take a step back and enjoy the simplicity of the client code in the `Temperature` class.

Summary

In this chapter we've explained Python's system of abstract base-classes, which is rather somewhat more flexible that similar concepts in other languages. In particlar we covered these topics:

- Subclass/instance checking
    - How the behaviour of the built-in `issubclass()` and `isinstance()` functions can be specialised for a base-class by defining the `__subclasscheck__()` and `__instancecheck__()` methods on the metaclass of that base-class.
    - Specialised subclass checks allow us to centralize the definition of what it means to be a subclass by gathering look-before-you-leap protocol checks into one place. Any class which implements the required protocol will become at least a virtual subclass of a virtual base-class.
- The standard library `abc` module contains tools for assisting in the definition of abstract base-classes.
    - Most important amongst those tools is the `ABCMeta` metaclass which can be used as the metaclass for abstract base-classes.
    - Slightly more conveniently, you can simply inherit from the `ABC` class which has `ABCMeta` as its metaclass.
    - `ABCMeta` provides default implementations of both `__subclasscheck__()` and `__instancecheck__()` which support two means of identifying subclasses: A special `__subclasshook__()` classmethod on abstract base-classes and a registration method.
    - `__subclasshook__()` accepts a candidate subclass as its only argument and should return `True` or `NotImplemented`. `False` should only be returned if it is desired to disable subclass registration.
    - Passing any class — even a built-in class — to the `register()` metamethod of an abstract base-class will register the argument as a *virtual* subclass of the base-class.

- An `@abstractmethod` decorator can be used to prevent instantiation of abstract classes. It requires methods marked as such to be overridden in real — although not virtual — subclasses.
- The `@abstractmethod` decorator can be combined with other decorators such as `@staticmethod`, `@classmethod`, and `@property`, but `@abstractmethod` should always be the innermost decorator.
- Descriptors should propagate abstractness from underlying methods by exposing the `__isabstractmethod__` attribute.

Afterword: Continue the journey

Python is a large and complicated language with many moving parts. We find it remarkable that much of this complexity is hidden so well in Python. We hope in this book we’ve given you deeper insight into some important mechanisms in Python which, while a bit trickier to understand, can deliver great expressive power. You have reached the end of this book on advanced Python, and now is the time to take what you have learned and apply these powerful techniques in your work and play with Python. No matter where your journey goes, though, remember that, above all else, it’s great *fun* to write Python software, so enjoy yourselves!

Notes

1 At least Python 3.5, though any version greater than that will work as well (e.g. 3.6).↩

2 See https://mail.python.org/pipermail/python-ideas/2009-October/006157.html↩

3 See http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.6084&rep=rep1&type=pdf↩

4 A widespread misconception is that `else` clauses associated with for-loops are executed if and only if the iterator which the for-loop is consuming is initially empty. This is not the case, the `else` clause will be executed *when* the iterator is exhausted, but only if the for-loop was also exited 'naturally', without invoking `break`. As such, the `else` clause is useful to handle the "not found" case when searching for something:↩

5 Note we are required to pass the `length` parameter to specify how many bytes we want in the result and the `byteorder` parameter to specify whether we want the bytes returned in big-endian order with the most significant byte first, or little-endian order with the least significant byte first↩

6 We've taken additional steps in this book to make the output even more readable by wrapping the interleaved lines of indices and buffer data to make it more readable. The code we present outputs all indices on one line, and all buffer bytes on the following line.↩

7 The word *cast* is a synonym for *type-conversion* used in many different programming languages. The exact meaning is language dependent, but it typically refers to reinterpreting the bit-level data as a different type. We don’t often talk about casting in Python, as Python is strongly typed, but this one place where the notion of casting crops up.↩

8 From Python 3.7 dictionary insertion order will be preserved, according to dictat from Python’s Benevolent Dictator for Life (BDFL), Guido van Rossum, on the Python mailing list in December 2017. https://mail.python.org/pipermail/python-dev/2017-December/151283.html↩

9 You’ll notice that in our view, whatever the International Astronomical Union says, Pluto **is** a planet!↩

10 Of course your system will likely report different numbers. A lot of factors are involved in memory usage. The real point is that allocating 10,000 board should show a marked increase in memory usage.↩

11 Recall from chapter 4 that a descriptor is any object supporting any of the `__get__()`, `__set__()`, or `__delete__()` methods.↩