Mastering Python

Python is a dynamic programming language. It is known for its high readability and hence it is often the first language learned by new programmers. Python being multi-paradigm, it can be used to achieve the same thing in different ways and it is compatible with different platforms. Even if you find writing Python code easy, writing code that is efficient, easy to maintain, and reuse is not so straightforward.

This book is an authoritative guide that will help you learn new advanced methods in a clear and contextualized way. It starts off by creating a project-specific environment using venv, introducing you to different Pythonic syntax and common pitfalls before moving on to cover the functional features in Python. Gradually, you will move towards more advanced techniques such as coroutines, async I/O, metaclasses, and extensions written in C/C++. In addition to that we will see how these can be used for debugging, profiling, and performance improvements. You will learn how to optimize application performance so that it works efficiently across multiple machines and Python versions. By the end of the book, you will be able to write more advanced scripts and take on bigger challenges.

Who this book is written for
Almost anyone can learn to write working script and create high quality code but they might lack a structured understanding of what it means to be Pythonic. If you are a Python programmer who wants to code efficiently by getting the syntax and usage of a few intricate Python techniques exactly right, this book is for you.

What you will learn from this book
- Learn what it means to write Pythonic code
- Use the functional programming paradigm effectively and how it relates to Lambda Calculus
- Get familiar with the different ways the decorators can be used and created
- Understand the power of generators, coroutines, and asynchronous I/O
- Create metaclasses and learn how they are the building blocks of Python classes
- Generate useful and readable documentation out of documents and code using Sphinx
- Learn how to track and optimize application performance, both memory and CPU
- Use the multiprocessing library, not just locally but also distributed across multiple machines
- Get a basic understanding of packaging and creating your own libraries/applications


In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 5 'Decorators – Enabling Code Reuse by Decorating'
- A synopsis of the book’s content
- More information on Mastering Python
About the Author

Rick van Hattem is an experienced programmer, entrepreneur, and software/database architect with over 20 years of programming experience, including 15 with Python. Additionally, he has a lot of experience with high-performance architectures featuring large amounts of concurrent users and/or data.

Rick has founded several start-ups and has done consulting for many companies, including a few Y Combinator start-ups and several large companies. One of the startups he founded, Fashiolista.com, is one of the largest social networks for fashion in the world, featuring millions of users and the performance challenges to accompany those.

Rick was one of the reviewers on the book *PostgreSQL Server Programming*, Packt Publishing.
Preface

Python is a language that is easy to learn and both powerful and convenient from the start. Mastering Python, however, is a completely different question.

Every programming problem you will encounter has at least several possible solutions and/or paradigms to apply within the vast possibilities of Python. This book will not only illustrate a range of different and new techniques but also explain where and when a method should be applied.

This book is not a beginner's guide to Python 3. It is a book that can teach you about the more advanced techniques possible within Python. Specifically targeting Python 3.5 and up, it also demonstrates several Python 3.5-only features such as async def and await statements.

As a Python programmer with many years of experience, I will attempt to rationalize the choices made in this book with relevant background information. These rationalizations are in no way strict guidelines, however. Several of these cases boil down to personal style in the end. Just know that they stem from experience and are, in many cases, the solutions recommended by the Python community.

Some of the references in this book might not be obvious to you if you are not a fan of Monty Python. This book extensively uses spam and eggs instead of foo and bar in code samples. To provide some background information, I recommend watching the "Spam" sketch by Monty Python. It is positively silly!

What this book covers

Chapter 1, Getting Started – One Environment per Project, introduces virtual Python environments using virtualenv or venv to isolate the packages in your Python projects.

Chapter 2, Pythonic Syntax, Common Pitfalls, and Style Guide, explains what Pythonic code is and how to write code that is Pythonic and adheres to the Python philosophy.
Chapter 3, Containers and Collections – Storing Data the Right Way, is where we use the many containers and collections bundled with Python to create code that is fast and readable.

Chapter 4, Functional Programming – Readability Versus Brevity, covers functional programming techniques such as list/dict/set comprehensions and lambda statements that are available in Python. Additionally, it illustrates their similarities with the mathematical principles involved.

Chapter 5, Decorators – Enabling Code Reuse by Decorating, explains not only how to create your own function/class decorators, but also how internal decorators such as property, staticmethod, and classmethod work.

Chapter 6, Generators and Coroutines – Infinity, One Step at a Time, shows how generators and coroutines can be used to lazily evaluate structures of infinite size.

Chapter 7, Async IO – Multithreading without Threads, demonstrates the usage of asynchronous functions using async def and await so that external resources no longer stall your Python processes.

Chapter 8, Metaclasses – Making Classes (Not Instances) Smarter, goes deeper into the creation of classes and how class behavior can be completely modified.

Chapter 9, Documentation – How to Use Sphinx and reStructuredText, shows how you can make Sphinx automatically document your code with very little effort. Additionally, it shows how the Napoleon syntax can be used to document function arguments in a way that is legible both in the code and the documentation.

Chapter 10, Testing and Logging – Preparing for Bugs, explains how code can be tested and how logging can be added to enable easy debugging in case bugs occur at a later time.

Chapter 11, Debugging – Solving the Bugs, demonstrates several methods of hunting down bugs with the use of tracing, logging, and interactive debugging.

Chapter 12, Performance – Tracking and Reducing Your Memory and CPU Usage, shows several methods of measuring and improving CPU and memory usage.

Chapter 13, Multiprocessing – When a Single CPU Core Is Not Enough, illustrates that the multiprocessing library can be used to execute your code, not just on multiple processors but even on multiple machines.
Chapter 14, Extensions in C/C++, System Calls, and C/C++ Libraries, covers the calling of C/C++ functions for both interoperability and performance using Ctypes, CFFI, and native C/C++.

Chapter 15, Packaging – Creating Your Own Libraries or Applications, demonstrates the usage of setuptools and setup.py to build and deploy packages on the Python Package Index (PyPI).
In this chapter, you are going to learn about Python decorators. Decorators are essentially function/class wrappers that can be used to modify the input, output, or even the function/class itself before executing it. This type of wrapping can just as easily be achieved by having a separate function that calls the inner function, or via mixins. As is the case with many Python constructs, decorators are not the only way to reach the goal but are definitely convenient in many cases.

While you can live perfectly without knowing too much about decorators, they give you a lot of "reuse power" and are therefore used heavily in framework libraries such as web frameworks. Python actually comes bundled with some useful decorators, most notably the property decorator.

There are, however, some particularities to take note of: wrapping a function creates a new function and makes it harder to reach the inner function and its properties. One example of this is the help(function) functionality of Python; by default, you will lose function properties such as the help text and the module the function exists in.

This chapter will cover the usage of both function and class decorators as well as the intricate details you need to know when decorating functions within classes.

The following are the topics covered:

- Decorating functions
- Decorating class functions
- Decorating classes
- Using classes as decorators
- Useful decorators in the Python standard library
Decorators – Enabling Code Reuse by Decorating

Decorating functions
Essentially, a decorator is nothing more than a function or class wrapper. If we have a function called `spam` and a decorator called `eggs`, then the following would decorate `spam` with `eggs`:

```python
spam = eggs(spam)
```

To make the syntax easier to use, Python has a special syntax for this case. So, instead of adding a line such as the preceding one below the function, you can simply decorate a function using the `@` operator:

```python
@eggs
def spam():
    pass
```

The decorator simply receives the function and returns a—usually different—function. The simplest possible decorator is:

```python
def eggs(function):
    return function
```

Looking at the earlier example, we realize that this gets `spam` as the argument for `function` and returns that function again, effectively changing nothing. Most decorators nest functions, however. The following decorator will print all arguments sent to `spam` and pass them to `spam` unmodified:

```python
>>> import functools

>>> def eggs(function):
...    @functools.wraps(function)
...    def _eggs(*args, **kwargs):
...        print('%r got args: %r and kwargs: %r' % (
...            function.__name__, args, kwargs))
...        return function(*args, **kwargs)
...    return _eggs

>>> @eggs
... def spam(a, b, c):
...    pass
```

```python
>>> def eggs(function):
...    @functools.wraps(function)
...    def _eggs(*args, **kwargs):
...        print('%r got args: %r and kwargs: %r' % (
...            function.__name__, args, kwargs))
...        return function(*args, **kwargs)
...    return _eggs

>>> @eggs
... def spam(a, b, c):
...    pass
```
... return a * b + c

>>> spam(1, 2, 3)
'spam' got args: (1, 2, 3) and kwargs: {}
5

This should indicate how powerful decorators can be. By modifying *args and **kwargs, you can add, modify and remove arguments completely. Additionally, the return statement can be modified as well. Instead of return function(...), you can return something completely different if you wish.

**Why functools.wraps is important**

Whenever you are writing a decorator, always be sure to add functools.wraps to wrap the inner function. Without wrapping it, you will lose all properties from the original function, which can lead to confusion. Take a look at the following code without functools.wraps:

```python
>>> def eggs(function):
...    def _eggs(*args, **kwargs):
...        return function(*args, **kwargs)
...    return _eggs

>>> @eggs
... def spam(a, b, c):
...    '''The spam function Returns a * b + c'''
...    return a * b + c

>>> help(spam)
Help on function _eggs in module ...:
<BLANKLINE>_eggs(*args, **kwargs)
<BLANKLINE>

>>> spam.__name__
'_eggs'
```

---

[105]
Decorators – Enabling Code Reuse by Decorating

Now, our spam method has no documentation anymore and the name is gone. It has been renamed to _eggs. Since we are indeed calling _eggs, this is understandable, but it’s very inconvenient for code that relies on this information. Now we will try the same code with the minor difference; we will use functools.wraps:

```python
>>> import functools

>>> def eggs(function):
...     @functools.wraps(function)
...     def _eggs(*args, **kwargs):
...         return function(*args, **kwargs)
...     return _eggs

>>> @eggs
... def spam(a, b, c):
...     '''The spam function Returns a * b + c'''
...     return a * b + c

>>> help(spam)
Help on function spam in module ...:
<BLANKLINE>
spam(a, b, c)
    The spam function Returns a * b + c
<BLANKLINE>

>>> spam.__name__
'spam'
```

Without any further changes, we now have documentation and the expected function name. The working of functools.wraps is nothing magical though; it simply copies and updates several attributes. Specifically, the following attributes are copied:

- __doc__
- __name__
- __module__
- __annotations__
- __qualname__
Additionally, \_\_dict\_ is updated using \_eggs\\.\_\_dict\_.update(spam.\_\_dict\_), and a new property called \_\_wrapped\_ is added, which contains the original (\_\_wrapped\_ spam in this case) function. The actual \_\_wrapped\_ function is available in the \_\_wrapped\_ file of your Python distribution.

How are decorators useful?

The use cases for decorators are plentiful, but some of the most useful cases are with debugging. More extensive examples of this will be covered in Chapter 11, Debugging – Solving the Bugs but I can give you a sneak preview of how to use decorators to keep track of what your code is doing.

Let's assume you have a bunch of functions that may or may not be called, and you're not entirely sure what kind of input and output each of these is getting. In this case, you could, of course, modify the function and add some print statements at the beginning and the end to print the output. This quickly gets tedious, however, and it's one of those cases where a simple decorator will make it easy to do the same thing.

For this example, we are using a very simple function, but we all know that in real life, we're not always that lucky:

```python
>>> def spam(eggs):
...     return 'spam' * (eggs % 5)
...     output = spam(3)
>>> output = spam(3)
spam(3): 'spamspamspam'
```

While this works, wouldn't it be far nicer to have a little decorator that takes care of this problem?

```python
>>> def debug(function):
...     @functools.wraps(function)
...     def _debug(*args, **kwargs):
```

```python
...     output = 'spam' * (eggs % 5)
...     print('spam(%r): %r' % (eggs, output))
...     return output
...
>>> output = spam(3)
spam(3): 'spamspamspam'
```
Decorators – Enabling Code Reuse by Decorating

```python
...     output = function(*args, **kwargs)
...     print('%s(%r, %r): %r' % (function.__name__, args, kwargs, output))
...     return output
...     return _debug
...
>>> @debug
... def spam(eggs):
...     return 'spam' * (eggs % 5)
...
>>> output = spam(3)
spam((3,), {}): 'spamspamspam'
```

Now we have a decorator that we can easily reuse for any function that prints the input, output, and function name. This type of decorator can also be very useful for logging applications, as we will see in Chapter 10, Testing and Logging – Preparing for Bugs. It should be noted that you can use this example even if you are not able to modify the module containing the original code. We can wrap the function locally and even monkey-patch the module if needed:

```python
import some_module

# Regular call
some_module.some_function()

# Wrap the function
debug_some_function = debug(some_module.some_function)

# Call the debug version
debug_some_function()

# Monkey patch the original module
some_module.some_function = debug_some_function

# Now this calls the debug version of the function
some_module.some_function()
```

Naturally, monkey-patching is not a good idea in production code, but it can be very useful when debugging.
Memoization using decorators

Memoization is a simple trick for making some code run a bit faster. The basic trick here is to store a mapping of the input and expected output so that you have to calculate a value only once. One of the most common examples of this technique is when demonstrating the naïve (recursive) Fibonacci function:

```python
>>> import functools

>>> def memoize(function):
...     function.cache = dict()
...
...     @functools.wraps(function)
...     def _memoize(*args):
...         if args not in function.cache:
...             function.cache[args] = function(*args)
...         return function.cache[args]
...     return _memoize

>>> @memoize
... def fibonacci(n):
...     if n < 2:
...         return n
...     else:
...         return fibonacci(n - 1) + fibonacci(n - 2)

>>> for i in range(1, 7):
...     print('fibonacci %d: %d' % (i, fibonacci(i)))
fibonacci 1: 1
fibonacci 2: 1
fibonacci 3: 2
fibonacci 4: 3
fibonacci 5: 5
fibonacci 6: 8

>>> fibonacci.__wrapped__.cache
{(5,): 5, (0,): 0, (6,): 8, (1,): 1, (2,): 1, (3,): 2, (4,): 3}
```
While this example would work just fine without any memoization, for larger numbers, it would kill the system. For \( n=2 \), the function would execute \( \text{fibonacci}(n-1) \) and \( \text{fibonacci}(n-2) \) recursively, effectively giving an exponential time complexity. Also, effectively for \( n=30 \), the Fibonacci function is called 2,692,537 times which is still doable nonetheless. At \( n=40 \), it is going to take you quite a very long time to calculate.

The memoized version, however, doesn't even break a sweat and only needs to execute 31 times for \( n=30 \).

This decorator also shows how a context can be attached to a function itself. In this case, the cache property becomes a property of the internal (wrapped \( \text{fibonacci} \)) function so that an extra \text{memoize} \( \) decorator for a different object won't clash with any of the other decorated functions.

Note, however, that implementing the memoization function yourself is generally not that useful anymore since Python introduced \text{lru_cache} (least recently used cache) in Python 3.2. The \text{lru_cache} is similar to the preceding \text{memoize} function but a bit more advanced. It only maintains a fixed (128 by default) cache size to save memory and uses some statistics to check whether the cache size should be increased.

To demonstrate how \text{lru_cache} works internally, we will calculate \( \text{fibonacci}(100) \), which would keep our computer busy until the end of the universe without any caching. Moreover, to make sure that we can actually see how many times the \( \text{fibonacci} \) function is being called, we'll add an extra decorator that keeps track of the count, as follows:

```python
>>> import functools

# Create a simple call counting decorator
>>> def counter(function):
...     function.calls = 0
...     @functools.wraps(function)
...     def _counter(*args, **kwargs):
...         function.calls += 1
...         return function(*args, **kwargs)
...     return _counter

# Create a LRU cache with size 3
>>> @functools.lru_cache(maxsize=3)
... @counter
... def fibonacci(n):
```
...  if n < 2:
...      return n
...  else:
...      return fibonacci(n - 1) + fibonacci(n - 2)

>>> fibonacci(100)
354224848179261915075

# The LRU cache offers some useful statistics
>>> fibonacci.cache_info()
CacheInfo(hits=98, misses=101, maxsize=3, currsize=3)

# The result from our counter function which is now wrapped both by
# our counter and the cache
>>> fibonacci.__wrapped__.__wrapped__.calls
101

You might wonder why we need only 101 calls with a cache size of 3. That's because
we recursively require only \( n - 1 \) and \( n - 2 \), so we have no need of a larger cache
in this case. With others, it would still be useful though.

Additionally, this example shows the usage of two decorators for a single function.
You can see these as the layers of an onion. The first one is the outer layer and it
works towards the inside. When calling \( \text{fibonacci} \), \( \text{lru}
\text{cache} \) will be called first
because it's the first decorator in the list. Assuming there is no cache available yet,
the \text{counter} decorator will be called. Within the counter, the actual \( \text{fibonacci} \)
function will be called.

Returning the values works in the reverse order, of course; \( \text{fibonacci} \) returns its
value to \text{counter}, which passes the value along to \text{lru}
\text{cache}.

**Decorators with (optional) arguments**
The previous examples mostly used simple decorators without any arguments. As
we have already seen with \text{lru}
\text{cache}, decorators can accept arguments as well since
they are just regular functions, but this adds an extra layer to a decorator. This means
that adding an argument can be as simple as the following:

```python
>>> import functools

>>> def add(extra_n=1):
```

---

[111]
Decorators – Enabling Code Reuse by Decorating

... 'Add extra_n to the input of the decorated function'
...
...
... # The inner function, notice that this is the actual
...
... # decorator
...
... def _add(function):
...
...     # The actual function that will be called
...
...     @functools.wraps(function)
...
...     def __add(n):
...
...         return function(n + extra_n)
...
...
...     return __add
...
...
...     return _add
...

>>> @add(extra_n=2)
... def eggs(n):
...     return 'eggs' * n

>>> eggs(2)
'eggsseggseggseggs'

Optional arguments are a different matter, however, because they make the extra function layer optional. With arguments, you need three layers, but without arguments, you need only two layers. Since decorators are essentially regular functions that return functions, the difference would be to return the sub-function or the sub-sub-function, based on the parameters. This leaves just one issue—detecting whether the parameter is a function or a regular parameter. To illustrate, with the parameters the actual call looks like the following:

add(extra_n=2)(eggs)(2)

Whereas the call without arguments would look like this:

add(eggs)(2)

To detect whether the decorator was called with a function or a regular argument as a parameter, we have several options, none of which are completely ideal in my opinion:

- Using keyword arguments for decorator arguments so that the regular argument will always be the function
- Detecting whether the first and only argument is callable
In my opinion, the first one—using keyword arguments—is the better of the two options because it is somewhat more explicit and leaves less room for confusion. The second option could be problematic if, for some reason, your argument is callable as well.

Using the first method, the normal (non-keyword) argument has to be the decorated function and the other two checks can still apply. We can still check whether the function is indeed callable and whether there is only a single argument available. Here is an example using a modified version of the previous example:

```python
>>> import functools

>>> def add(*args, **kwargs):
...     'Add n to the input of the decorated function'
...     # The default kwargs, we don't store this in kwargs
...     # because we want to make sure that args and kwargs
...     # can't both be filled
...     default_kwargs = dict(n=1)
...     # The inner function, notice that this is actually a
...     # decorator itself
...     def _add(function):
...         # The actual function that will be called
...         @functools.wraps(function)
...         def __add__(n):
...             default_kwargs.update(kwargs)
...             return function(n + default_kwargs['n'])
...         return __add
...     if len(args) == 1 and callable(args[0]) and not kwargs:
...         # Decorator call without arguments, just call it
...         # ourselves
...         return _add(args[0])
...     elif not args and kwargs:
...         # Decorator call with arguments, this time it will
...         # automatically be executed with function as the
```

[113]
Decorators – Enabling Code Reuse by Decorating

... # first argument
... default_kwargs.update(kwargs)
... return _add
... else:
...     raise RuntimeError('This decorator only supports '
...                        'keyword arguments')

>>> @add
... def spam(n):
...     return 'spam' * n

>>> @add(n=3)
... def eggs(n):
...     return 'eggs' * n

>>> spam(3)
'spamspamspamspam'

>>> eggs(2)
'eggseggeggseggeggseggs'

>>> @add(3)
... def bacon(n):
...     return 'bacon' * n
Traceback (most recent call last):
...
RuntimeError: This decorator only supports keyword arguments

Whenever you have the choice available, I recommend that you either have a decorator with arguments or without them, instead of having optional arguments. However, if you have a really good reason for making the arguments optional, then you have a relatively safe method of making this possible.
Creating decorators using classes

Similar to how we create regular function decorators, it is also possible to create decorators using classes instead. After all, a function is just a callable object and a class can implement the callable interface as well. The following decorator works similarly to the `debug` decorator we used earlier, but uses a class instead of a regular function:

```python
>>> import functools

>>> class Debug(object):
...     def __init__(self, function):
...         self.function = function
...         # functools.wraps for classes
...         functools.update_wrapper(self, function)
...     
...     def __call__(self, *args, **kwargs):
...         output = self.function(*args, **kwargs)
...         print('%s(%r, %r): %r' % (
...             self.function.__name__, args, kwargs, output))
...         return output

>>> @Debug
... def spam(eggs):
...     return 'spam' * (eggs % 5)

>>> output = spam(3)
spam((3,), {}): 'spamspamspam'

The only notable difference between functions and classes is that `functools.wraps` is now replaced with `functools.update_wrapper` in the `__init__` method.
Decorating class functions

Decorating class functions is very similar to regular functions, but you need to be aware of the required first argument, self — the class instance. You have most likely already used a few class function decorators. The `classmethod`, `staticmethod`, and `property` decorators for example, are used in many different projects. To explain how all this works, we will build our own versions of the `classmethod`, `staticmethod`, and `property` decorators. First, let's look at a simple decorator for class functions to show the difference from regular decorators:

```python
>>> import functools

>>> def plus_one(function):
...     @functools.wraps(function)
...     def _plus_one(self, n):
...         return function(self, n + 1)
...     return _plus_one

>>> class Spam(object):
...     @plus_one
...     def get_eggs(self, n=2):
...         return n * 'eggs'

>>> spam = Spam()
>>> spam.get_eggs(3)
'eggseggseggseggs'
```

As is the case with regular functions, the class function decorator now gets passed along self as the instance. Nothing unexpected!

Skipping the instance – `classmethod` and `staticmethod`

The difference between a `classmethod` and a `staticmethod` is fairly simple. The `classmethod` passes a class object instead of a class instance (self), and `staticmethod` skips both the class and the instance entirely. This effectively makes `staticmethod` very similar to a regular function outside of a class.
Before we recreate `classmethod` and `staticmethod`, we need to take a look at the expected behavior of these methods:

```python
>>> import pprint

>>> class Spam(object):
...    def some_instancemethod(self, *args, **kwargs):
...        print('self: %r' % self)
...        print('args: %s' % pprint.pformat(args))
...        print('kwargs: %s' % pprint.pformat(kwargs))
...
...    @classmethod
...    def some_classmethod(cls, *args, **kwargs):
...        print('cls: %r' % cls)
...        print('args: %s' % pprint.pformat(args))
...        print('kwargs: %s' % pprint.pformat(kwargs))
...
...    @staticmethod
...    def some_staticmethod(*args, **kwargs):
...        print('args: %s' % pprint.pformat(args))
...        print('kwargs: %s' % pprint.pformat(kwargs))

# Create an instance so we can compare the difference between executions with and without instances easily
>>> spam = Spam()

# With an instance (note the lowercase spam)
>>> spam.some_instancemethod(1, 2, a=3, b=4)
sel: <...Spam object at 0x...>
args: (1, 2)
kwargs: {'a': 3, 'b': 4}

# Without an instance (note the capitalized Spam)
>>> Spam.some_instancemethod()
```
Traceback (most recent call last):
...

TypeError: some_instancemethod() missing 1 required positional argument: 'self'

# But what if we add parameters? Be very careful with these!
# Our first argument is now used as an argument, this can give
# very strange and unexpected errors
>>> Spam.some_instancemethod(1, 2, a=3, b=4)
   self: 1
   args: (2,)
   kwargs: {'a': 3, 'b': 4}

# Classmethods are expectedly identical
>>> spam.some_classmethod(1, 2, a=3, b=4)
   cls: <class '...Spam'>
   args: (1, 2)
   kwargs: {'a': 3, 'b': 4}

>>> Spam.some_classmethod()
   cls: <class '...Spam'>
   args: ()
   kwargs: {}

>>> Spam.some_classmethod(1, 2, a=3, b=4)
   cls: <class '...Spam'>
   args: (1, 2)
   kwargs: {'a': 3, 'b': 4}

# Staticmethods are also identical
>>> spam.some_staticmethod(1, 2, a=3, b=4)
   args: (1, 2)
   kwargs: {'a': 3, 'b': 4}

>>> Spam.some_staticmethod()
>>> Spam.some_staticmethod(1, 2, a=3, b=4)
args: (1, 2)
kwargs: {'a': 3, 'b': 4}

Note that calling `some_instancemethod` without an instance results in an error whereby `self` is missing. As expected (since we didn't instantiate the class in that case), for the version with the arguments, it seems to work but it is actually broken. This is because the first argument is now assumed to be `self`. This is obviously incorrect in this case, where you pass an integer, but if you had passed along some other class instance, this could be a source of very strange bugs. Both `classmethod` and `staticmethod` handle this correctly.

Before we can continue with decorators, you need to be aware of how Python descriptors function. Descriptors can be used to modify the binding behavior of object attributes. This means that if a descriptor is used as the value of an attribute, you can modify which value is being set, get, and deleted when these operations are called on the attribute. Here is a basic example of this behavior:

```python
>>> class MoreSpam(object):
...     def __init__(self, more=1):
...         self.more = more
...     def __get__(self, instance, cls):
...         return self.more + instance.spam
...     def __set__(self, instance, value):
...         instance.spam = value - self.more
```

```python
>>> class Spam(object):
...     more_spam = MoreSpam(5)
...     def __init__(self, spam):
...         self.spam = spam
```
As you can see, whenever we set or get values from more_spam, it actually calls __get__ or __set__ on MoreSpam. A very useful feat for automatic conversions and type checking, the property decorator we will see in the next paragraph is just a more convenient implementation of this technique.

Now that we know how descriptors work, we can continue with creating the classmethod and staticmethod decorators. For these two, we simply need to modify __get__ instead of __call__ so that we can control which type of instance (or none at all) is passed along:

```python
import functools

class ClassMethod(object):
    def __init__(self, method):
        self.method = method

    def __get__(self, instance, cls):
        @functools.wraps(self.method)
        def method(*args, **kwargs):
            return self.method(cls, *args, **kwargs)
        return method

class StaticMethod(object):
    def __init__(self, method):
        self.method = method

    def __get__(self, instance, cls):
        return self.method
```
The `ClassMethod` decorator still features a sub-function to actually produce a working decorator. Looking at the function, you can most likely guess how it functions. Instead of passing `instance` as the first argument to `self.method`, it passes `cls`.

`StaticMethod` is even simpler, because it completely ignores both the `instance` and the `cls`. It can just return the original method unmodified. Because it returns the original method without any modifications, we have no need for the `functools.wraps` call either.

### Properties – smart descriptor usage

The `property` decorator is probably the most used decorator in Python land. It allows you to add getters/setters to existing instance properties so that you can add validators and modify your values before setting them to your instance properties. The `property` decorator can be used both as an assignment and as a decorator. The following example shows both syntaxes so that we know what to expect from the `property` decorator:

```python
>>> class Spam(object):
...     def get_eggs(self):
...         print('getting eggs')
...         return self._eggs
...
...     def set_eggs(self, eggs):
...         print('setting eggs to %s' % eggs)
...         self._eggs = eggs
...
...     def delete_eggs(self):
...         print('deleting eggs')
...         del self._eggs
...
...     eggs = property(get_eggs, set_eggs, delete_eggs)
...
...     @property
...     def spam(self):
...         print('getting spam')
...         return self._spam
...
```
Decorators – Enabling Code Reuse by Decorating

```python
... @spam.setter
... def spam(self, spam):
...     print('setting spam to %s' % spam)
...     self._spam = spam
...
... @spam.deleter
... def spam(self):
...     print('deleting spam')
...     del self._spam

>>> spam = Spam()
>>> spam.eggs = 123
setting eggs to 123
>>> spam.eggs
getting eggs
123
>>> del spam.eggs
deleting eggs
```

Note that the property decorator works only if the class inherits object.

Similar to how we implemented the `classmethod` and `staticmethod` decorators, we need the Python descriptors again. This time, we require the full power of the descriptors, however—not just `__get__` but `__set__` and `__delete__` as well:

```python
class Property(object):
    def __init__(self, fget=None, fset=None, fdel=None,
                 doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        # If no specific documentation is available, copy it
        # from the getter
        if fget and not doc:
            doc = fget.__doc__
        self.__doc__ = doc
```
def __get__(self, instance, cls):
    if instance is None:
        # Redirect class (not instance) properties to
        # self
        return self
    elif self.fget:
        return self.fget(instance)
    else:
        raise AttributeError('unreadable attribute')

def __set__(self, instance, value):
    if self.fset:
        self.fset(instance, value)
    else:
        raise AttributeError('can’t set attribute')

def __delete__(self, instance):
    if self.fdel:
        self.fdel(instance)
    else:
        raise AttributeError('can’t delete attribute')

def getter(self, fget):
    return type(self)(fget, self.fset, self.fdel)

def setter(self, fset):
    return type(self)(self.fget, fset, self.fdel)

def deleter(self, fdel):
    return type(self)(self.fget, self.fset, fdel)

As you can see, most of the Property implementation is simply an implementation
of the descriptor methods. The getter, setter, and deleter functions are simply
shortcuts for making the usage of the decorator possible, which is why we have to
return self if no instance is available.

Naturally, there are more methods of achieving this effect. In the previous paragraph,
we saw the bare descriptor implementation, and in our previous example, we saw
the property decorator. A somewhat more generic solution for a class is to implement
__getattr__ or __getattribute__. Here’s a simple demonstration:

>>> class Spam(object):
...     def __init__(self):
...         self.registry = {}
...     def __getattr__(self, key):
...         print('Getting %r' % key)
...         return self.registry.get(key, 'Undefined')
...     def __setattr__(self, key, value):
...         if key == 'registry':
...             object.__setattr__(self, key, value)
...         else:
...             print('Setting %r to %r' % (key, value))
...             self.registry[key] = value
...     def __delattr__(self, key):
...         print('Deleting %r' % key)
...         del self.registry[key]

>>> spam = Spam()

>>> spam.a
Getting 'a'
'Undefined'

>>> spam.a = 1
Setting 'a' to 1

>>> spam.a
Getting 'a'
1

>>> del spam.a
Deleting 'a'

The __getattr__ method looks for the key in instance.__dict__ first and is called only if it does not exist. That's why we never see a __getattr__ for the registry attribute. The __getattribute__ method is called in all cases, which makes it a bit more dangerous to use. With the __getattribute__ method, you will need a specific exclusion for registry since it will be executed recursively if you try to access self.registry.
There is rarely a need to look at descriptors, but they are used by several internal Python processes, such as the `super()` method when inheriting classes.

**Decorating classes**

Python 2.6 introduced the class decorator syntax. As is the case with the function decorator syntax, this is not really a new technique either. Even without the syntax, a class can be decorated simply by executing `DecoratedClass = decorator(RegularClass)`. After the previous paragraphs, you should be familiar with writing decorators. Class decorators are no different from regular ones, except for the fact that they take a class instead of a function. As is the case with functions, this happens at declaration time and not at instantiating/calling time.

Because there are quite a few alternative ways to modify how classes work, such as standard inheritance, mixins, and metaclasses (more about that in Chapter 8, *Metaclasses – Making Classes (Not Instances) Smarter*), class decorators are never strictly needed. This does not reduce their usefulness, but it does offer an explanation of why you will most likely not see too many examples of class decorating in the wild.

**Singletons – classes with a single instance**

Singletons are classes that always allow only a single instance to exist. So, instead of getting an instance specifically for your call, you always get the same one. These can be very useful for things such as a database connection pool, where you don't want to keep opening connections all of the time but want to reuse the original ones:

```python
>>> import functools
>>> def singleton(cls):
...     instances = dict()
...     @functools.wraps(cls)
...     def _singleton(*args, **kwargs):
...         if cls not in instances:
...             instances[cls] = cls(*args, **kwargs)
...         return instances[cls]
...     return _singleton

>>> @singleton
... class Spam(object):
...     def __init__(self):
```
...     print('Executing init')

>>> a = Spam()
Executing init
>>> b = Spam()

>>> a is b
True

>>> a.x = 123
>>> b.x
123

As you can see in the a is b comparison, both objects have the same identity, so we can conclude that they are indeed the same object. As is the case with regular decorators, due to the functools.wraps functionality, we can still access the original class through Spam.__wrapped__ if needed.

The is operator compares objects by identity, which is implemented as the memory address in CPython. If a is b returns True, we can conclude that both a and b are the same instance.

Total ordering – sortable classes the easy way

At some point or the other, you have probably needed to sort data structures. While this is easily achievable using the key parameter to the sorted function, there is a more convenient way if you need to do this often—by implementing the __gt__, __ge__, __lt__, __le__, and __eq__ functions. That seems a bit verbose, doesn't it? If you want the best performance, it's still a good idea, but if you can take a tiny performance hit and some slightly more complicated stack traces, then total_ordering might be a nice alternative. The total_ordering class decorator can implement all required sort functions based on a class that possesses an __eq__ function and one of the comparison functions (__lt__, __le__, __gt__, or __ge__). This means you can seriously shorten your function definitions. Let's compare the regular one and the one using the total_ordering decorator:

>>> import functools

>>> class Value(object):


[126]
... def __init__(self, value):
...    self.value = value
...
... def __repr__(self):
...    return '<%s[%d]>' % (self.__class__, self.value)

>>> class Spam(Value):
...    def __gt__(self, other):
...        return self.value > other.value
...
...    def __ge__(self, other):
...        return self.value >= other.value
...
...    def __lt__(self, other):
...        return self.value < other.value
...
...    def __le__(self, other):
...        return self.value <= other.value
...
...    def __eq__(self, other):
...        return self.value == other.value

>>> @functools.total_ordering
... class Egg(Value):
...    def __lt__(self, other):
...        return self.value < other.value
...
...    def __eq__(self, other):
...        return self.value == other.value

>>> numbers = [4, 2, 3, 4]
>>> spams = [Spam(n) for n in numbers]
>>> eggs = [Egg(n) for n in numbers]

>>> spams

>>> eggs

>>> sorted(spams)

>>> sorted(eggs)

# Sorting using key is of course still possible and in this case
# perhaps just as easy:
>>> values = [Value(n) for n in numbers]
>>> values

>>> sorted(values, key=lambda v: v.value)

Now, you might be wondering, "Why isn't there a class decorator to make a class
sortable using a specified key property?" Well, that might indeed be a good idea for
the functools library but it isn't there yet. So let's see how we would implement
something like it:

>>> def sort_by_attribute(attr, keyfunc=getattr):
...     def _sort_by_attribute(cls):
...         def __gt__(self, other):
...             return getattr(self, attr) > getattr(other, attr)
...     def __ge__(self, other):
...         return getattr(self, attr) >= getattr(other, attr)
...     return _sort_by_attribute
... return sort_by_attribute

[128]
... def __lt__(self, other):
    return getattr(self, attr) < getattr(other, attr)
...
... def __le__(self, other):
    return getattr(self, attr) <= getattr(other, attr)
...
... def __eq__(self, other):
    return getattr(self, attr) <= getattr(other, attr)
...
... cls.__gt__ = __gt__
... cls.__ge__ = __ge__
... cls.__lt__ = __lt__
... cls.__le__ = __le__
... cls.__eq__ = __eq__
...
... return cls
... return _sort_by_attribute

>>> class Value(object):
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return '<%s[%d]>' % (self.__class__, self.value)

>>> @sort_by_attribute('value')
class Spam(Value):
    pass

>>> numbers = [4, 2, 3, 4]
>>> spams = [Spam(n) for n in numbers]
>>> sorted(spams)
[[<class '...Spam'>[2]], [<class '...Spam'>[3]],
[<class '...Spam'>[4]], [<class '...Spam'>[4]>]
Decorators – Enabling Code Reuse by Decorating

Certainly, this greatly simplifies the making of a sortable class. And if you would rather have your own key function instead of getattr, it's even easier. Simply replace the getattr(self, attr) call with key_function(self), do that for other as well, and change the argument for the decorator to your function. You can even use that as the base function and implement sort_by_attribute by simply passing a wrapped getattr function.

Useful decorators

In addition to the ones already mentioned in this chapter, Python comes bundled with a few other useful decorators. There are some that aren't in the standard library (yet?).

Single dispatch – polymorphism in Python

If you've used C++ or Java before, you're probably used to having ad hoc polymorphism available—different functions being called depending on the argument types. Python being a dynamically typed language, most people would not expect the possibility of a single dispatch pattern. Python, however, is a language that is not only dynamically typed but also strongly typed, which means we can rely on the type we receive.

A dynamically typed language does not require strict type definitions. On the other hand, a language such as C would require the following to declare an integer:

```python
int some_integer = 123;
```

Python simply accepts that your value has a type:

```python
some_integer = 123
```

As opposed to languages such as JavaScript and PHP, however, Python does very little implicit type conversion. In Python, the following will return an error, whereas JavaScript would execute it without any problems:

```python
'spam' + 5
```

In Python, the result is a TypeError. In Javascript, it's `'spam5'`. 
The idea of single dispatch is that depending on the type you pass along, the correct function is called. Since `str + int` results in an error in Python, this can be very convenient to automatically convert your arguments before passing them to your function. This can be useful to separate the actual workings of your function from the type conversions.

Since Python 3.4, there is a decorator that makes it easily possible to implement the single dispatch pattern in Python. For one of those cases that you need to handle a specific type different from the normal execution. Here is the basic example:

```python
>>> import functools

>>> @functools.singledispatch
def printer(value):
    print('other: %r' % value)

>>> @printer.register(str)
def str_printer(value):
    print(value)

>>> @printer.register(int)
def int_printer(value):
    printer('int: %d' % value)

>>> @printer.register(dict)
def dict_printer(value):
    printer('dict:)
    for k, v in sorted(value.items()):
        printer('  key: %r, value: %r' % (k, v))

>>> printer('spam')
spam

>>> printer([1, 2, 3])
other: [1, 2, 3]

>>> printer(123)
```
int: 123

>>> printer({'a': 1, 'b': 2})
dict:
    key: 'a', value: 1
    key: 'b', value: 2

See how, depending on the type, the other functions were called? This pattern can be very useful for reducing the complexity of a single function that takes several types of argument.

When naming the functions, make sure that you do not overwrite the original `singledispatch` function. If we had named `str_printer` as just `printer`, it would overwrite the initial `printer` function. This would make it impossible to access the original `printer` function and make all register operations after that fail as well.

Now, a slightly more useful example—differentiating between a filename and a file handler:

>>> import json
>>> import functools

```python
>>> @functools.singledispatch
def write_as_json(file, data):
    json.dump(data, file)
```

```python
>>> @write_as_json.register(str)
>>> @write_as_json.register(bytes)
def write_as_json_filename(file, data):
    with open(file, 'w') as fh:
        write_as_json(fh, data)
```

```python
>>> data = dict(a=1, b=2, c=3)
```
Chapter 5

>>> write_as_json('test1.json', data)
>>> write_as_json(b'test2.json', 'w')
>>> with open('test3.json', 'w') as fh:
...     write_as_json(fh, data)

So now we have a single `write_as_json` function; it calls the right code depending on the type. If it's an `str` or `bytes` object, it will automatically open the file and call the regular version of `write_as_json`, which accepts file objects.

Writing a decorator that does this is not that hard to do, of course, but it's still quite convenient to have it in the base library. It most certainly beats a couple of `isinstance` calls in your function. To see which function will be called, you can use the `write_as_json.dispatch` function with a specific type. When passing along an `str`, you will get the `write_as_json_filename` function. It should be noted that the name of the dispatched functions is completely arbitrary. They are accessible as regular functions, of course, but you can name them anything you like.

To check the registered types, you can access the registry, which is a dictionary, through `write_as_json.registry`:

```python
>>> write_as_json.registry.keys()
dict_keys([<class 'bytes'>, <class 'object'>, <class 'str'>])
```

Contextmanager, with statements made easy

Using the `contextmanager` class, we can make the creation of a context wrapper very easy. Context wrappers are used whenever you use a `with` statement. One example is the `open` function, which works as a context wrapper as well, allowing you to use the following code:

```python
with open(filename) as fh:
    pass
```

Let's just assume for now that the `open` function is not usable as a context manager and that we need to build our own function to do this. The standard method of creating a context manager is by creating a class that implements the `__enter__` and `__exit__` methods, but that's a bit verbose. We can have it shorter and simpler:

```python
>>> import contextlib

>>> @contextlib.contextmanager
... def open_context_manager(filename, mode='r'):
...     fh = open(filename, mode)
...     yield fh
...     fh.close()
```

```python
>>> import contextlib

>>> @contextlib.contextmanager
... def open_context_manager(filename, mode='r'):
...     fh = open(filename, mode)
...     yield fh
...     fh.close()
```
...     yield fh
...     fh.close()

>>> with open_context_manager('test.txt', 'w') as fh:
...     print('Our test is complete!', file=fh)

Simple, right? However, I should mention that for this specific case—the closing of objects—there is a dedicated function in `contextlib`, and it is even easier to use. Let's demonstrate it:

```python
>>> import contextlib

>>> with contextlib.closing(open('test.txt', 'a')) as fh:
...     print('Yet another test', file=fh)
```

For a `file` object, this is of course not needed since it already functions as a context manager. However, some objects such as requests made by `urllib` don't support automatic closing in that manner and benefit from this function.

But wait; there's more! In addition to being usable in a `with` statement, the results of a `contextmanager` are actually usable as decorators since Python 3.2. In older Python versions, it was simply a small wrapper, but since Python 3.2 it's based on the `ContextDecorator` class, which makes it a decorator. The previous decorator isn't really suitable for that task since it yields a result (more about that in Chapter 6, Generators and Coroutines – Infinity, One Step at a Time), but we can think of other functions:

```python
>>> @contextlib.contextmanager
... def debug(name):
...     print('Debugging %r:' % name)
...     yield
...     print('End of debugging %r' % name)

>>> @debug('spam')
... def spam():
...     print('This is the inside of our spam function')

>>> spam()
Debugging 'spam':
This is the inside of our spam function
End of debugging 'spam'
```
There are quite a few nice use cases for this, but at the very least, it's just a convenient way to wrap a function in a context without all the (nested) `with` statements.

**Validation, type checks, and conversions**

While checking for types is usually not the best way to go in Python, at times it can be useful if you know that you will need a specific type (or something that can be cast to that type). To facilitate this, Python 3.5 introduces a type hinting system so that you can do the following:

```python
def spam(eggs: int):
    pass
```

Since Python 3.5 is not that common yet, here's a decorator that achieves the same with more advanced type checking. To allow for this type of checking, some magic has to be used, specifically the usage of the `inspect` module. Personally, I am not a great fan of inspecting code to perform tricks like these, as they are easy to break. This piece of code actually breaks when a regular decorator (one that doesn't copy `argspec`) is used between the function and this decorator, but it's a nice example nonetheless:

```python
>>> import inspect
>>> import functools

>>> def to_int(name, minimum=None, maximum=None):
...     def _to_int(function):
...         # Use the method signature to map *args to named arguments
...         signature = inspect.signature(function)
...         @functools.wraps(function,['__signature__'])
...         def __to_int(*args, **kwargs):
...             # Bind all arguments to the names so we get a single mapping of all arguments
...             bound = signature.bind(*args, **kwargs)
...             return function(**bound)
...         return __to_int
...     return _to_int

>>> def spam(eggs: int):
...     pass
```

---

[135]
Decorators – Enabling Code Reuse by Decorating

```python
...  # Make sure the value is (convertible to) an integer
...  default = signature.parameters[name].default
...  value = int(bound.arguments.get(name, default))
...
...  # Make sure it's within the allowed range
...  if minimum is not None:
...      assert value >= minimum, ('%s should be at least %r, got: %r' %
...          (name, minimum, value))
...
...  if maximum is not None:
...      assert value <= maximum, ('%s should be at most %r, got: %r' %
...          (name, maximum, value))
...
...  return function(*args, **kwargs)
...  return __to_int
...
>>> @to_int('a', minimum=10)
... @to_int('b', maximum=10)
... @to_int('c')
... def spam(a, b, c=10):
...     print('a', a)
...     print('b', b)
...     print('c', c)

>>> spam(10, b=0)
a 10
b 0
c 10

>>> spam(a=20, b=10)
a 20
b 10
c 10

>>> spam(1, 2, 3)
```

---

[136]
Traceback (most recent call last):
...
AssertionError: a should be at least 10, got: 1

>>> spam()
Traceback (most recent call last):
...
TypeError: 'a' parameter lacking default value

>>> spam('spam', {})
Traceback (most recent call last):
...
ValueError: invalid literal for int() with base 10: 'spam'

Because of the inspect magic, I’m still not sure whether I would recommend using the decorator like this. Instead, I would opt for a simpler version that uses no inspect whatsoever and simply parses the arguments from kwargs:

```python
>>> import functools
>>> def to_int(name, minimum=None, maximum=None):
...     def _to_int(function):
...         @functools.wraps(function)
...         def __to_int(**kwargs):
...             value = int(kwargs.get(name))
...             # Make sure it's within the allowed range
...             if minimum is not None:
...                 assert value >= minimum, (
...                     '%s should be at least %r, got: %r' %
...                     (name, minimum, value))
...             if maximum is not None:
...                 assert value <= maximum, (
...                     '%s should be at most %r, got: %r' %
...                     (name, maximum, value))
...             ...
```

```py[137]```
Decorators – Enabling Code Reuse by Decorating

```python
...    return function(**kwargs)
...    return __to_int
...    return _to_int

>>> @to_int('a', minimum=10)
... @to_int('b', maximum=10)
... def spam(a, b):
...     print('a', a)
...     print('b', b)

>>> spam(a=20, b=10)
a 20
b 10

>>> spam(a=1, b=10)
AssertionError: a should be at least 10, got: 1
```

However, as demonstrated, supporting both `args` and `kwargs` is not impossible as long as you keep in mind that `__signature__` is not copied by default. Without `__signature__`, the inspect module won't know which parameters are allowed and which aren't.

The missing `__signature__` issue is currently being discussed and might be solved in a future Python version: http://bugs.python.org/issue23764.

### Useless warnings – how to ignore them

Generally when writing Python, warnings are very useful the first time when you're actually writing the code. When executing it, however, it is not useful to get that same message every time you run your script/application. So, let's create some code that allows easy hiding of the expected warnings, but not all of them so that we can easily catch new ones:

```python
import warnings
import functools

def ignore_warning(warning, count=None):
```
```python
def _ignore_warning(function):
    @functools.wraps(function)
    def __ignore_warning(*args, **kwargs):
        # Execute the code while recording all warnings
        with warnings.catch_warnings(record=True) as ws:
            # Catch all warnings of this type
            warnings.simplefilter('always', warning)
            # Execute the function
            result = function(*args, **kwargs)

            # Now that all code was executed and the warnings
            # collected, re-send all warnings that are beyond our
            # expected number of warnings
            if count is not None:
                for w in ws[count:]:
                    warnings.showwarning(
                        message=w.message,
                        category=w.category,
                        filename=w.filename,
                        lineno=w.lineno,
                        file=w.file,
                        line=w.line,
                    )

        return result
    return __ignore_warning
return _ignore_warning

@ignore_warning(DeprecationWarning, count=1)
def spam():
    warnings.warn('deprecation 1', DeprecationWarning)
    warnings.warn('deprecation 2', DeprecationWarning)

Using this method, we can catch the first (expected) warning and still see the second
(not expected) warning.
```
Summary
This chapter showed us some of the places where decorators can be used to make our code simpler and add some fairly complex behavior to very simple functions. Truthfully, most decorators are more complex than the regular function would have been by simply adding the functionality directly, but the added advantage of applying the same pattern to many functions and classes is generally well worth it.

Decorators have so many uses to make your functions and classes smarter and more convenient to use:

- Debugging
- Validation
- Argument convenience (pre-filling or converting arguments)
- Output convenience (converting the output to a specific type)

The most important takeaway of this chapter should be to never forget `functools.wraps` when wrapping a function. Debugging decorated functions can be rather difficult because of (unexpected) behavior modification, but losing attributes as well can make that problem much worse.

The next chapter will show us how and when to use generators and coroutines. This chapter has already shown us the usage of the `with` statement slightly, but generators and coroutines go much further with this. We will still be using decorators often though, so make sure you have a good understanding of how they work.
Where to buy this book

You can buy Mastering Python from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.