Free Sample

Community Experience Distilled

Python 3 Object-oriented Programming
Second Edition

Unleash the power of Python 3 objects

Dusty Phillips
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'Expecting the Unexpected'
- A synopsis of the book’s content
About the Author

Dusty Phillips is a Canadian software developer and author currently living in Seattle, Washington. He has been active in the open source community for a decade and a half and programming in Python for nearly all of it. He cofounded the popular Puget Sound Programming Python meetup group; drop by and say hi if you’re in the area.

Python 3 Object Oriented Programming, Packt Publishing, was the first of his books. He has also written Creating Apps In Kivy, O’Reilly, the mobile Python library, and self-published Hacking Happy, a journey to mental wellness for the technically inclined. He was hospitalized for suicidal tendencies shortly after the first edition of this book was published and has been an outspoken proponent for positive mental health ever since.
Preface

This book introduces the terminology of the object-oriented paradigm. It focuses on object-oriented design with step-by-step examples. It guides us from simple inheritance, one of the most useful tools in the object-oriented programmer's toolbox through exception handling to design patterns, an object-oriented way of looking at object-oriented concepts.

Along the way, we'll learn to integrate the object-oriented and not-so-object-oriented aspects of the Python programming language. We will learn the complexities of string and file manipulation, emphasizing (as Python 3 does) the difference between binary and textual data.

We'll then cover the joys of unit testing, using not one, but two unit testing frameworks. Finally, we'll explore, through Python's various concurrency paradigms, how to make objects work well together at the same time.

What this book covers

This book is loosely divided into four major parts. In the first four chapters, we will dive into the formal principles of object-oriented programming and how Python leverages them. In chapters 5 through 8, we will cover some of Python's idiosyncratic applications of these principles by learning how they are applied to a variety of Python's built-in functions. Chapters 9 through 11 cover design patterns, and the final two chapters discuss two bonus topics related to Python programming that may be of interest.

Chapter 1, Object-oriented Design, covers important object-oriented concepts. It deals mainly with terminology such as abstraction, classes, encapsulation, and inheritance. We also briefly look at UML to model our classes and objects.
Chapter 2, *Objects in Python*, discusses classes and objects and how they are used in Python. We will learn about attributes and behaviors on Python objects, and also the organization of classes into packages and modules. Lastly, we will see how to protect our data.

Chapter 3, *When Objects Are Alike*, gives us a more in-depth look into inheritance. It covers multiple inheritance and shows us how to extend built-ins. This chapter also covers how polymorphism and duck typing work in Python.

Chapter 4, *Expecting the Unexpected*, looks into exceptions and exception handling. We will learn how to create our own exceptions and how to use exceptions for program flow control.

Chapter 5, *When to Use Object-oriented Programming*, deals with creating and using objects. We will see how to wrap data using properties and restrict data access. This chapter also discusses the DRY principle and how not to repeat code.

Chapter 6, *Python Data Structures*, covers the object-oriented features of Python's built-in classes. We'll cover tuples, dictionaries, lists, and sets, as well as a few more advanced collections. We'll also see how to extend these standard objects.

Chapter 7, *Python Object-oriented Shortcuts*, as the name suggests, deals with time-savers in Python. We will look at many useful built-in functions such as method overloading using default arguments. We'll also see that functions themselves are objects and how this is useful.

Chapter 8, *Strings and Serialization*, looks at strings, files, and formatting. We'll discuss the difference between strings, bytes, and bytearrays, as well as various ways to serialize textual, object, and binary data to several canonical representations.

Chapter 9, *The Iterator Pattern*, introduces us to the concept of design patterns and covers Python's iconic implementation of the iterator pattern. We'll learn about list, set, and dictionary comprehensions. We'll also demystify generators and coroutines.

Chapter 10, *Python Design Patterns I*, covers several design patterns, including the decorator, observer, strategy, state, singleton, and template patterns. Each pattern is discussed with suitable examples and programs implemented in Python.

Chapter 11, *Python Design Patterns II*, wraps up our discussion of design patterns with coverage of the adapter, facade, flyweight, command, abstract, and composite patterns. More examples of how idiomatic Python code differs from canonical implementations are provided.
Chapter 12, *Testing Object-oriented Programs*, opens with why testing is so important in Python applications. It emphasizes test-driven development and introduces two different testing suites: unittest and py.test. Finally, it discusses mocking test objects and code coverage.

Chapter 13, *Concurrency*, is a whirlwind tour of Python's support (and lack thereof) of concurrency patterns. It discusses threads, multiprocessing, futures, and the new AsyncIO library.

Each chapter includes relevant examples and a case study that collects the chapter's contents into a working (if not complete) program.
Expecting the Unexpected

Programs are very fragile. It would be ideal if code always returned a valid result, but sometimes a valid result can't be calculated. For example, it's not possible to divide by zero, or to access the eighth item in a five-item list.

In the old days, the only way around this was to rigorously check the inputs for every function to make sure they made sense. Typically, functions had special return values to indicate an error condition; for example, they could return a negative number to indicate that a positive value couldn't be calculated. Different numbers might mean different errors occurred. Any code that called this function would have to explicitly check for an error condition and act accordingly. A lot of code didn't bother to do this, and programs simply crashed. However, in the object-oriented world, this is not the case.

In this chapter, we will study exceptions, special error objects that only need to be handled when it makes sense to handle them. In particular, we will cover:

- How to cause an exception to occur
- How to recover when an exception has occurred
- How to handle different exception types in different ways
- Cleaning up when an exception has occurred
- Creating new types of exception
- Using the exception syntax for flow control
Raising exceptions

In principle, an exception is just an object. There are many different exception classes available, and we can easily define more of our own. The one thing they all have in common is that they inherit from a built-in class called `BaseException`. These exception objects become special when they are handled inside the program's flow of control. When an exception occurs, everything that was supposed to happen doesn't happen, unless it was supposed to happen when an exception occurred. Make sense? Don't worry, it will!

The easiest way to cause an exception to occur is to do something silly! Chances are you've done this already and seen the exception output. For example, any time Python encounters a line in your program that it can't understand, it bails with `SyntaxError`, which is a type of exception. Here's a common one:

```python
>>> print "hello world"
  File "<stdin>", line 1
    print "hello world"
    ^
SyntaxError: invalid syntax
```

This `print` statement was a valid command in Python 2 and previous versions, but in Python 3, because `print` is now a function, we have to enclose the arguments in parenthesis. So, if we type the preceding command into a Python 3 interpreter, we get the `SyntaxError`.

In addition to `SyntaxError`, some other common exceptions, which we can handle, are shown in the following example:

```python
>>> x = 5 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

```python
>>> lst = [1,2,3]
>>> print(lst[3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

```python
>>> lst + 2
Traceback (most recent call last):
```

```
Sometimes these exceptions are indicators of something wrong in our program (in which case we would go to the indicated line number and fix it), but they also occur in legitimate situations. A ZeroDivisionError doesn't always mean we received an invalid input. It could also mean we have received a different input. The user may have entered a zero by mistake, or on purpose, or it may represent a legitimate value, such as an empty bank account or the age of a newborn child.

You may have noticed all the preceding built-in exceptions end with the name Error. In Python, the words error and exception are used almost interchangeably. Errors are sometimes considered more dire than exceptions, but they are dealt with in exactly the same way. Indeed, all the error classes in the preceding example have Exception (which extends BaseException) as their superclass.

Raising an exception

We'll get to handling exceptions in a minute, but first, let's discover what we should do if we're writing a program that needs to inform the user or a calling function that the inputs are somehow invalid. Wouldn't it be great if we could use the same mechanism that Python uses? Well, we can! Here's a simple class that adds items to a list only if they are even numbered integers:
class EvenOnly(list):
    def append(self, integer):
        if not isinstance(integer, int):
            raise TypeError("Only integers can be added")
        if integer % 2:
            raise ValueError("Only even numbers can be added")
        super().append(integer)

This class extends the list built-in, as we discussed in Chapter 2, Objects in Python, and overrides the append method to check two conditions that ensure the item is an even integer. We first check if the input is an instance of the int type, and then use the modulus operator to ensure it is divisible by two. If either of the two conditions is not met, the raise keyword causes an exception to occur. The raise keyword is simply followed by the object being raised as an exception. In the preceding example, two objects are newly constructed from the built-in classes TypeError and ValueError. The raised object could just as easily be an instance of a new exception class we create ourselves (we'll see how shortly), an exception that was defined elsewhere, or even an exception object that has been previously raised and handled. If we test this class in the Python interpreter, we can see that it is outputting useful error information when exceptions occur, just as before:

```python
>>> e = EvenOnly()
>>> e.append("a string")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "even_integers.py", line 7, in add
      raise TypeError("Only integers can be added")
TypeError: Only integers can be added

>>> e.append(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "even_integers.py", line 9, in add
      raise ValueError("Only even numbers can be added")
ValueError: Only even numbers can be added
>>> e.append(2)
```
While this class is effective for demonstrating exceptions in action, it isn't very good at its job. It is still possible to get other values into the list using index notation or slice notation. This can all be avoided by overriding other appropriate methods, some of which are double-underscore methods.

The effects of an exception

When an exception is raised, it appears to stop program execution immediately. Any lines that were supposed to run after the exception is raised are not executed, and unless the exception is dealt with, the program will exit with an error message. Take a look at this simple function:

```python
def no_return():
    print("I am about to raise an exception")
    raise Exception("This is always raised")
    print("This line will never execute")
    return "I won't be returned"
```

If we execute this function, we see that the first `print` call is executed and then the exception is raised. The second `print` statement is never executed, and the `return` statement never executes either:

```bash
>>> no_return()
I am about to raise an exception
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "exception_quits.py", line 3, in no_return
    raise Exception("This is always raised")
Exception: This is always raised
```

Furthermore, if we have a function that calls another function that raises an exception, nothing will be executed in the first function after the point where the second function was called. Raising an exception stops all execution right up through the function call stack until it is either handled or forces the interpreter to exit. To demonstrate, let's add a second function that calls the earlier one:

```python
def call_exception():
    print("call_exception starts here...")
    no_return()
    print("an exception was raised...")
    print("...so these lines don't run")
```
When we call this function, we see that the first print statement executes, as well as the first line in the no_return function. But once the exception is raised, nothing else executes:

```python
>>> call_exception()
call_exception starts here...
I am about to raise an exception
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "method_calls_excepting.py", line 9, in call_exception
      no_return()
  File "method_calls_excepting.py", line 3, in no_return
      raise Exception("This is always raised")
Exception: This is always raised
```

We'll soon see that when the interpreter is not actually taking a shortcut and exiting immediately, we can react to and deal with the exception inside either method. Indeed, exceptions can be handled at any level after they are initially raised.

Look at the exception's output (called a traceback) from bottom to top, and notice how both methods are listed. Inside no_return, the exception is initially raised. Then, just above that, we see that inside call_exception, that pesky no_return function was called and the exception bubbled up to the calling method. From there, it went up one more level to the main interpreter, which, not knowing what else to do with it, gave up and printed a traceback.

### Handling exceptions

Now let's look at the tail side of the exception coin. If we encounter an exception situation, how should our code react to or recover from it? We handle exceptions by wrapping any code that might throw one (whether it is exception code itself, or a call to any function or method that may have an exception raised inside it) inside a try...except clause. The most basic syntax looks like this:

```python
try:
    no_return()
except:
    print("I caught an exception")
print("executed after the exception")
```
If we run this simple script using our existing `no_return` function, which as we know very well, always throws an exception, we get this output:

```
I am about to raise an exception
I caught an exception
executed after the exception
```

The `no_return` function happily informs us that it is about to raise an exception, but we fooled it and caught the exception. Once caught, we were able to clean up after ourselves (in this case, by outputting that we were handling the situation), and continue on our way, with no interference from that offensive function. The remainder of the code in the `no_return` function still went unexecuted, but the code that called the function was able to recover and continue.

Note the indentation around `try` and `except`. The `try` clause wraps any code that might throw an exception. The `except` clause is then back on the same indentation level as the `try` line. Any code to handle the exception is indented after the `except` clause. Then normal code resumes at the original indentation level.

The problem with the preceding code is that it will catch any type of exception. What if we were writing some code that could raise both a `TypeError` and a `ZeroDivisionError`? We might want to catch the `ZeroDivisionError`, but let the `TypeError` propagate to the console. Can you guess the syntax?

Here's a rather silly function that does just that:

```python
def funny_division(divider):
    try:
        return 100 / divider
    except ZeroDivisionError:
        return "Zero is not a good idea!"

print(funny_division(0))
print(funny_division(50.0))
print(funny_division("hello"))
```

The function is tested with `print` statements that show it behaving as expected:

```
Zero is not a good idea!
2.0
```

```python
Traceback (most recent call last):
  File "catch_specific_exception.py", line 9, in <module>
    print(funny_division("hello"))
  File "catch_specific_exception.py", line 3, in funny_division
    return 100 / anumber
```

`TypeError: unsupported operand type(s) for /: 'int' and 'str'.`
The first line of output shows that if we enter 0, we get properly mocked. If we call with a valid number (note that it's not an integer, but it's still a valid divisor), it operates correctly. Yet if we enter a string (you were wondering how to get a TypeError, weren't you?), it fails with an exception. If we had used an empty except clause that didn't specify a ZeroDivisionError, it would have accused us of dividing by zero when we sent it a string, which is not a proper behavior at all.

We can even catch two or more different exceptions and handle them with the same code. Here's an example that raises three different types of exception. It handles TypeError and ZeroDivisionError with the same exception handler, but it may also raise a ValueError if you supply the number 13:

```python
def funny_division2(anumber):
    try:
        if anumber == 13:
            raise ValueError("13 is an unlucky number")
        return 100 / anumber
    except (ZeroDivisionError, TypeError):
        return "Enter a number other than zero"

for val in (0, "hello", 50.0, 13):
    print("Testing {}:", format(val), end=" ")
    print(funny_division2(val))
```

The for loop at the bottom loops over several test inputs and prints the results. If you're wondering about that end argument in the print statement, it just turns the default trailing newline into a space so that it's joined with the output from the next line. Here's a run of the program:

```
Testing 0: Enter a number other than zero
Testing hello: Enter a number other than zero
Testing 50.0: 2.0
Testing 13: Traceback (most recent call last):
  File "catch_multiple_exceptions.py", line 11, in <module>
    print(funny_division2(val))
  File "catch_multiple_exceptions.py", line 4, in funny_division2
    raise ValueError("13 is an unlucky number")
ValueError: 13 is an unlucky number
```
The number 0 and the string are both caught by the `except` clause, and a suitable error message is printed. The exception from the number 13 is not caught because it is a `ValueError`, which was not included in the types of exceptions being handled. This is all well and good, but what if we want to catch different exceptions and do different things with them? Or maybe we want to do something with an exception and then allow it to continue to bubble up to the parent function, as if it had never been caught? We don't need any new syntax to deal with these cases. It's possible to stack `except` clauses, and only the first match will be executed. For the second question, the `raise` keyword, with no arguments, will reraise the last exception if we're already inside an exception handler. Observe in the following code:

```python
def funny_division3(anumber):
    try:
        if anumber == 13:
            raise ValueError("13 is an unlucky number")
        return 100 / anumber
    except ZeroDivisionError:
        return "Enter a number other than zero"
    except TypeError:
        return "Enter a numerical value"
    except ValueError:
        print("No, No, not 13!")
        raise
```

The last line reraises the `ValueError`, so after outputting "No, No, not 13!", it will raise the exception again; we'll still get the original stack trace on the console.

If we stack exception clauses like we did in the preceding example, only the first matching clause will be run, even if more than one of them fits. How can more than one clause match? Remember that exceptions are objects, and can therefore be subclassed. As we'll see in the next section, most exceptions extend the `Exception` class (which is itself derived from `BaseException`). If we catch `Exception` before we catch `TypeError`, then only the `Exception` handler will be executed, because `TypeError` is an `Exception` by inheritance.

This can come in handy in cases where we want to handle some exceptions specifically, and then handle all remaining exceptions as a more general case. We can simply catch `Exception` after catching all the specific exceptions and handle the general case there.
Sometimes, when we catch an exception, we need a reference to the `Exception` object itself. This most often happens when we define our own exceptions with custom arguments, but can also be relevant with standard exceptions. Most exception classes accept a set of arguments in their constructor, and we might want to access those attributes in the exception handler. If we define our own exception class, we can even call custom methods on it when we catch it. The syntax for capturing an exception as a variable uses the `as` keyword:

```python
try:
    raise ValueError("This is an argument")
except ValueError as e:
    print("The exception arguments were", e.args)
```

If we run this simple snippet, it prints out the string argument that we passed into `ValueError` upon initialization.

We've seen several variations on the syntax for handling exceptions, but we still don't know how to execute code regardless of whether or not an exception has occurred. We also can't specify code that should be executed only if an exception does not occur. Two more keywords, `finally` and `else`, can provide the missing pieces. Neither one takes any extra arguments. The following example randomly picks an exception to throw and raises it. Then some not-so-complicated exception handling code is run that illustrates the newly introduced syntax:

```python
import random
some_exceptions = [ValueError, TypeError, IndexError, None]

try:
    choice = random.choice(some_exceptions)
    print("raising {}".format(choice))
    if choice:
        raise choice("An error")
except ValueError:
    print("Caught a ValueError")
except TypeError:
    print("Caught a TypeError")
except Exception as e:
    print("Caught some other error: %s" % (e.__class__.__name__))
else:
    print("This code called if there is no exception")
finally:
    print("This cleanup code is always called")
```
If we run this example—which illustrates almost every conceivable exception handling scenario—a few times, we’ll get different output each time, depending on which exception `random` chooses. Here are some example runs:

```
$ python finally_and_else.py
raising None
This code called if there is no exception
This cleanup code is always called

$ python finally_and_else.py
raising <class 'TypeError'>
Caught a TypeError
This cleanup code is always called

$ python finally_and_else.py
raising <class 'IndexError'>
Caught some other error: IndexError
This cleanup code is always called

$ python finally_and_else.py
raising <class 'ValueError'>
Caught a ValueError
This cleanup code is always called
```

Note how the `print` statement in the `finally` clause is executed no matter what happens. This is extremely useful when we need to perform certain tasks after our code has finished running (even if an exception has occurred). Some common examples include:

- Cleaning up an open database connection
- Closing an open file
- Sending a closing handshake over the network

The `finally` clause is also very important when we execute a `return` statement from inside a `try` clause. The `finally` handle will still be executed before the value is returned.
Also, pay attention to the output when no exception is raised: both the `else` and the `finally` clauses are executed. The `else` clause may seem redundant, as the code that should be executed only when no exception is raised could just be placed after the entire `try...except` block. The difference is that the `else` block will still be executed if an exception is caught and handled. We'll see more on this when we discuss using exceptions as flow control later.

Any of the `except`, `else`, and `finally` clauses can be omitted after a `try` block (although `else` by itself is invalid). If you include more than one, the `except` clauses must come first, then the `else` clause, with the `finally` clause at the end. The order of the `except` clauses normally goes from most specific to most generic.

### The exception hierarchy

We've already seen several of the most common built-in exceptions, and you'll probably encounter the rest over the course of your regular Python development. As we noticed earlier, most exceptions are subclasses of the `Exception` class. But this is not true of all exceptions. `Exception` itself actually inherits from a class called `BaseException`. In fact, all exceptions must extend the `BaseException` class or one of its subclasses.

There are two key exceptions, `SystemExit` and `KeyboardInterrupt`, that derive directly from `BaseException` instead of `Exception`. The `SystemExit` exception is raised whenever the program exits naturally, typically because we called the `sys.exit` function somewhere in our code (for example, when the user selected an exit menu item, clicked the "close" button on a window, or entered a command to shut down a server). The exception is designed to allow us to clean up code before the program ultimately exits, so we generally don't need to handle it explicitly (because cleanup code happens inside a `finally` clause).

If we do handle it, we would normally reraise the exception, since catching it would stop the program from exiting. There are, of course, situations where we might want to stop the program exiting, for example, if there are unsaved changes and we want to prompt the user if they really want to exit. Usually, if we handle `SystemExit` at all, it's because we want to do something special with it, or are anticipating it directly. We especially don't want it to be accidentally caught in generic clauses that catch all normal exceptions. This is why it derives directly from `BaseException`.

The `KeyboardInterrupt` exception is common in command-line programs. It is thrown when the user explicitly interrupts program execution with an OS-dependent key combination (normally, `Ctrl + C`). This is a standard way for the user to deliberately interrupt a running program, and like `SystemExit`, it should almost always respond by terminating the program. Also, like `SystemExit`, it should handle any cleanup tasks inside `finally` blocks.
Here is a class diagram that fully illustrates the exception hierarchy:

When we use the `except:` clause without specifying any type of exception, it will catch all subclasses of `BaseException`; which is to say, it will catch all exceptions, including the two special ones. Since we almost always want these to get special treatment, it is unwise to use the `except:` statement without arguments. If you want to catch all exceptions other than `SystemExit` and `KeyboardInterrupt`, explicitly catch `Exception`.

Furthermore, if you do want to catch all exceptions, I suggest using the syntax `except BaseException:` instead of a raw `except:`. This helps explicitly tell future readers of your code that you are intentionally handling the special case exceptions.

### Defining our own exceptions

Often, when we want to raise an exception, we find that none of the built-in exceptions are suitable. Luckily, it's trivial to define new exceptions of our own. The name of the class is usually designed to communicate what went wrong, and we can provide arbitrary arguments in the initializer to include additional information.

All we have to do is inherit from the `Exception` class. We don't even have to add any content to the class! We can, of course, extend `BaseException` directly, but then it will not be caught by generic `except Exception` clauses.

Here's a simple exception we might use in a banking application:

```python
class InvalidWithdrawal(Exception):
    pass

    raise InvalidWithdrawal("You don't have $50 in your account")
```

The last line illustrates how to raise the newly defined exception. We are able to pass an arbitrary number of arguments into the exception. Often a string message is used, but any object that might be useful in a later exception handler can be stored. The `Exception.__init__` method is designed to accept any arguments and store them as a tuple in an attribute named `args`. This makes exceptions easier to define without needing to override `__init__`. 
Of course, if we do want to customize the initializer, we are free to do so. Here's an exception whose initializer accepts the current balance and the amount the user wanted to withdraw. In addition, it adds a method to calculate how overdrawn the request was:

```python
class InvalidWithdrawal(Exception):
    def __init__(self, balance, amount):
        super().__init__("account doesn't have \${}").format(  
            amount)
        self.amount = amount
        self.balance = balance

    def overage(self):
        return self.amount - self.balance

raise InvalidWithdrawal(25, 50)
```

The `raise` statement at the end illustrates how to construct this exception. As you can see, we can do anything with an exception that we would do with other objects. We could catch an exception and pass it around as a working object, although it is more common to include a reference to the working object as an attribute on an exception and pass that around instead.

Here's how we would handle an `InvalidWithdrawal` exception if one was raised:

```python
try:
    raise InvalidWithdrawal(25, 50)
except InvalidWithdrawal as e:
    print("I'm sorry, but your withdrawal is "  
    "more than your balance by "  
    "\${}").format(e.overage()))
```

Here we see a valid use of the `as` keyword. By convention, most Python coders name the exception variable `e`, although, as usual, you are free to call it `ex`, `exception`, or `aunt_sally` if you prefer.

There are many reasons for defining our own exceptions. It is often useful to add information to the exception or log it in some way. But the utility of custom exceptions truly comes to light when creating a framework, library, or API that is intended for access by other programmers. In that case, be careful to ensure your code is raising exceptions that make sense to the client programmer. They should be easy to handle and clearly describe what went on. The client programmer should easily see how to fix the error (if it reflects a bug in their code) or handle the exception (if it's a situation they need to be made aware of).
Exceptions aren't exceptional. Novice programmers tend to think of exceptions as only useful for exceptional circumstances. However, the definition of exceptional circumstances can be vague and subject to interpretation. Consider the following two functions:

```python
def divide_with_exception(number, divisor):
    try:
        print("{} / {} = {}".format(
            number, divisor, number / divisor * 1.0))
    except ZeroDivisionError:
        print("You can't divide by zero")

def divide_with_if(number, divisor):
    if divisor == 0:
        print("You can't divide by zero")
    else:
        print("{} / {} = {}".format(
            number, divisor, number / divisor * 1.0))
```

These two functions behave identically. If `divisor` is zero, an error message is printed; otherwise, a message printing the result of division is displayed. We could avoid a `ZeroDivisionError` ever being thrown by testing for it with an `if` statement. Similarly, we can avoid an `IndexError` by explicitly checking whether or not the parameter is within the confines of the list, and a `KeyError` by checking if the key is in a dictionary.

But we shouldn't do this. For one thing, we might write an `if` statement that checks whether or not the index is lower than the parameters of the list, but forget to check negative values.

> Remember, Python lists support negative indexing; `-1` refers to the last element in the list.

Eventually, we would discover this and have to find all the places where we were checking code. But if we had simply caught the `IndexError` and handled it, our code would just work.
Python programmers tend to follow a model of *Ask forgiveness rather than permission*, which is to say, they execute code and then deal with anything that goes wrong. The alternative, to *look before you leap*, is generally frowned upon. There are a few reasons for this, but the main one is that it shouldn't be necessary to burn CPU cycles looking for an unusual situation that is not going to arise in the normal path through the code. Therefore, it is wise to use exceptions for exceptional circumstances, even if those circumstances are only a little bit exceptional. Taking this argument further, we can actually see that the exception syntax is also effective for flow control. Like an *if* statement, exceptions can be used for decision making, branching, and message passing.

Imagine an inventory application for a company that sells widgets and gadgets. When a customer makes a purchase, the item can either be available, in which case the item is removed from inventory and the number of items left is returned, or it might be out of stock. Now, being out of stock is a perfectly normal thing to happen in an inventory application. It is certainly not an exceptional circumstance. But what do we return if it's out of stock? A string saying out of stock? A negative number? In both cases, the calling method would have to check whether the return value is a positive integer or something else, to determine if it is out of stock. That seems a bit messy. Instead, we can raise *OutOfStockException* and use the *try* statement to direct program flow control. Make sense? In addition, we want to make sure we don't sell the same item to two different customers, or sell an item that isn't in stock yet. One way to facilitate this is to lock each type of item to ensure only one person can update it at a time. The user must lock the item, manipulate the item (purchase, add stock, count items left...), and then unlock the item. Here's an incomplete *Inventory* example with docstrings that describes what some of the methods should do:

```python
class Inventory:
    def lock(self, item_type):
        '''Select the type of item that is going to
           be manipulated. This method will lock the
           item so nobody else can manipulate the
           inventory until it's returned. This prevents
           selling the same item to two different
           customers.'''
        pass

    def unlock(self, item_type):
        '''Release the given type so that other
           customers can access it.'''
        pass

    def purchase(self, item_type):
```

— [112] —
'''If the item is not locked, raise an exception. If the item_type does not exist, raise an exception. If the item is currently out of stock, raise an exception. If the item is available, subtract one item and return the number of items left.''
pass

We could hand this object prototype to a developer and have them implement the methods to do exactly as they say while we work on the code that needs to make a purchase. We'll use Python's robust exception handling to consider different branches, depending on how the purchase was made:

```python
item_type = 'widget'
inv = Inventory()
inv.lock(item_type)
try:
    num_left = inv.purchase(item_type)
except InvalidItemType:
    print("Sorry, we don't sell {}\n".format(item_type))
except OutOfStock:
    print("Sorry, that item is out of stock.\n")
else:
    print("Purchase complete. There are ", "{} {}s left\n".format(num_left, item_type))
finally:
    inv.unlock(item_type)
```

Pay attention to how all the possible exception handling clauses are used to ensure the correct actions happen at the correct time. Even though OutOfStock is not a terribly exceptional circumstance, we are able to use an exception to handle it suitably. This same code could be written with an if...elif...else structure, but it wouldn't be as easy to read or maintain.

We can also use exceptions to pass messages between different methods. For example, if we wanted to inform the customer as to what date the item is expected to be in stock again, we could ensure our OutOfStock object requires a back_in_stock parameter when it is constructed. Then, when we handle the exception, we can check that value and provide additional information to the customer. The information attached to the object can be easily passed between two different parts of the program. The exception could even provide a method that instructs the inventory object to reorder or backorder an item.
Using exceptions for flow control can make for some handy program designs. The important thing to take from this discussion is that exceptions are not a bad thing that we should try to avoid. Having an exception occur does not mean that you should have prevented this exceptional circumstance from happening. Rather, it is just a powerful way to communicate information between two sections of code that may not be directly calling each other.

Case study
We've been looking at the use and handling of exceptions at a fairly low level of detail—syntax and definitions. This case study will help tie it all in with our previous chapters so we can see how exceptions are used in the larger context of objects, inheritance, and modules.

Today, we'll be designing a simple central authentication and authorization system. The entire system will be placed in one module, and other code will be able to query that module object for authentication and authorization purposes. We should admit, from the start, that we aren't security experts, and that the system we are designing may be full of security holes. Our purpose is to study exceptions, not to secure a system. It will be sufficient, however, for a basic login and permission system that other code can interact with. Later, if that other code needs to be made more secure, we can have a security or cryptography expert review or rewrite our module, preferably without changing the API.

Authentication is the process of ensuring a user is really the person they say they are. We'll follow the lead of common web systems today, which use a username and private password combination. Other methods of authentication include voice recognition, fingerprint or retinal scanners, and identification cards.

Authorization, on the other hand, is all about determining whether a given (authenticated) user is permitted to perform a specific action. We'll create a basic permission list system that stores a list of the specific people allowed to perform each action.

In addition, we'll add some administrative features to allow new users to be added to the system. For brevity, we'll leave out editing of passwords or changing of permissions once they've been added, but these (highly necessary) features can certainly be added in the future.
There's a simple analysis; now let's proceed with design. We're obviously going to need a `User` class that stores the username and an encrypted password. This class will also allow a user to log in by checking whether a supplied password is valid. We probably won't need a `Permission` class, as those can just be strings mapped to a list of users using a dictionary. We should have a central `Authenticator` class that handles user management and logging in or out. The last piece of the puzzle is an `Authorizor` class that deals with permissions and checking whether a user can perform an activity. We'll provide a single instance of each of these classes in the `auth` module so that other modules can use this central mechanism for all their authentication and authorization needs. Of course, if they want to instantiate private instances of these classes, for non-central authorization activities, they are free to do so.

We'll also be defining several exceptions as we go along. We'll start with a special `AuthException` base class that accepts a `username` and optional `user` object as parameters; most of our self-defined exceptions will inherit from this one.

Let's build the `User` class first; it seems simple enough. A new user can be initialized with a username and password. The password will be stored encrypted to reduce the chances of its being stolen. We'll also need a `check_password` method to test whether a supplied password is the correct one. Here is the class in full:

```python
import hashlib

class User:
    def __init__(self, username, password):
        '''Create a new user object. The password will be encrypted before storing.'''
        self.username = username
        self.password = self._encrypt_pw(password)
        self.is_logged_in = False

    def _encrypt_pw(self, password):
        '''Encrypt the password with the username and return the sha digest.'''
        hash_string = (self.username + password)
        hash_string = hash_string.encode("utf8")
        return hashlib.sha256(hash_string).hexdigest()

    def check_password(self, password):
        '''Return True if the password is valid for this user, false otherwise.'''
        encrypted = self._encrypt_pw(password)
        return encrypted == self.password
```
Since the code for encrypting a password is required in both __init__ and check_password, we pull it out to its own method. This way, it only needs to be changed in one place if someone realizes it is insecure and needs improvement. This class could easily be extended to include mandatory or optional personal details, such as names, contact information, and birth dates.

Before we write code to add users (which will happen in the as-yet undefined Authenticator class), we should examine some use cases. If all goes well, we can add a user with a username and password; the User object is created and inserted into a dictionary. But in what ways can all not go well? Well, clearly we don't want to add a user with a username that already exists in the dictionary. If we did so, we'd overwrite an existing user's data and the new user might have access to that user's privileges. So, we'll need a UsernameAlreadyExists exception. Also, for security's sake, we should probably raise an exception if the password is too short. Both of these exceptions will extend AuthException, which we mentioned earlier. So, before writing the Authenticator class, let's define these three exception classes:

```python
class AuthException(Exception):
    def __init__(self, username, user=None):
        super().__init__(username, user)
        self.username = username
        self.user = user

class UsernameAlreadyExists(AuthException):
    pass

class PasswordTooShort(AuthException):
    pass
```

The AuthException requires a username and has an optional user parameter. This second parameter should be an instance of the User class associated with that username. The two specific exceptions we're defining simply need to inform the calling class of an exceptional circumstance, so we don't need to add any extra methods to them.

Now let's start on the Authenticator class. It can simply be a mapping of usernames to user objects, so we'll start with a dictionary in the initialization function. The method for adding a user needs to check the two conditions (password length and previously existing users) before creating a new User instance and adding it to the dictionary:

```python
class Authenticator:
    def __init__(self):
        '''Construct an authenticator to manage
users logging in and out.''
self.users = {}

def add_user(self, username, password):
    if username in self.users:
        raise UsernameAlreadyExists(username)
    if len(password) < 6:
        raise PasswordTooShort(username)
    self.users[username] = User(username, password)

We could, of course, extend the password validation to raise exceptions for passwords that are too easy to crack in other ways, if we desired. Now let's prepare the login method. If we weren't thinking about exceptions just now, we might just want the method to return True or False, depending on whether the login was successful or not. But we are thinking about exceptions, and this could be a good place to use them for a not-so-exceptional circumstance. We could raise different exceptions, for example, if the username does not exist or the password does not match. This will allow anyone trying to log a user in to elegantly handle the situation using a try/except/else clause. So, first we add these new exceptions:

class InvalidUsername(AuthException):
    pass

class InvalidPassword(AuthException):
    pass

Then we can define a simple login method to our Authenticator class that raises these exceptions if necessary. If not, it flags the user as logged in and returns:

def login(self, username, password):
    try:
        user = self.users[username]
    except KeyError:
        raise InvalidUsername(username)

    if not user.check_password(password):
        raise InvalidPassword(username, user)

    user.is_logged_in = True
    return True

Notice how the KeyError is handled. This could have been handled using if username not in self.users: instead, but we chose to handle the exception directly. We end up eating up this first exception and raising a brand new one of our own that better suits the user-facing API.
We can also add a method to check whether a particular username is logged in. Deciding whether to use an exception here is trickier. Should we raise an exception if the username does not exist? Should we raise an exception if the user is not logged in?

To answer these questions, we need to think about how the method would be accessed. Most often, this method will be used to answer the yes/no question, “Should I allow them access to <something>?” The answer will either be, "Yes, the username is valid and they are logged in", or "No, the username is not valid or they are not logged in". Therefore, a Boolean return value is sufficient. There is no need to use exceptions here, just for the sake of using an exception.

```python
def is_logged_in(self, username):
    if username in self.users:
        return self.users[username].is_logged_in
    return False
```

Finally, we can add a default authenticator instance to our module so that the client code can access it easily using `auth.authenticate`:

```python
authenticator = Authenticator()
```

This line goes at the module level, outside any class definition, so the authenticator variable can be accessed as `auth.authenticate`. Now we can start on the `Authorizor` class, which maps permissions to users. The `Authorizor` class should not permit user access to a permission if they are not logged in, so they’ll need a reference to a specific authenticator. We’ll also need to set up the permission dictionary upon initialization:

```python
class Authorizor:
    def __init__(self, authenticator):
        self.authenticator = authenticator
        self.permissions = {}
```

Now we can write methods to add new permissions and to set up which users are associated with each permission:

```python
def add_permission(self, perm_name):
    '''Create a new permission that users can be added to'''
    try:
        perm_set = self.permissions[perm_name]
    except KeyError:
        self.permissions[perm_name] = set()
    else:
```
raise PermissionError("Permission Exists")

def permit_user(self, perm_name, username):
    '''Grant the given permission to the user'''
    try:
        perm_set = self.permissions[perm_name]
    except KeyError:
        raise PermissionError("Permission does not exist")
    else:
        if username not in self.authenticator.users:
            raise InvalidUsername(username)
        perm_set.add(username)

The first method allows us to create a new permission, unless it already exists, in
which case an exception is raised. The second allows us to add a username to a
permission, unless either the permission or the username doesn’t yet exist.

We use set instead of list for usernames, so that even if you grant a user
permission more than once, the nature of sets means the user is only in the set
once. We'll discuss sets further in a later chapter.

A PermissionError is raised in both methods. This new error doesn't require
a username, so we'll make it extend Exception directly, instead of our custom
AuthException:

class PermissionError(Exception):
    pass

Finally, we can add a method to check whether a user has a specific permission
or not. In order for them to be granted access, they have to be both logged into the
authenticator and in the set of people who have been granted access to that privilege.
If either of these conditions is unsatisfied, an exception is raised:

def check_permission(self, perm_name, username):
    if not self.authenticator.is_logged_in(username):
        raise NotLoggedInError(username)
    try:
        perm_set = self.permissions[perm_name]
    except KeyError:
        raise PermissionError("Permission does not exist")
    else:
        if username not in perm_set:
            raise NotPermittedError(username)
        else:
            return True
There are two new exceptions in here; they both take usernames, so we'll define them as subclasses of `AuthException`:

```python
class NotLoggedInError(AuthException):
    pass

class NotPermittedError(AuthException):
    pass
```

Finally, we can add a default `authorizor` to go with our default authenticator:

```python
authorizor = Authorizor(authenticator)
```

That completes a basic authentication/authorization system. We can test the system at the Python prompt, checking to see whether a user, `joe`, is permitted to do tasks in the paint department:

```python
>>> import auth
>>> auth.authenticator.add_user("joe", "joepassword")
>>> auth.authorizor.add_permission("paint")
>>> auth.authorizor.check_permission("paint", "joe")
Traceback (most recent call last):
  File "<stdin>" line 1, in <module>
  File "auth.py", line 109, in check_permission
    raise NotLoggedInError(username)
auth.NotLoggedInError: joe
>>> auth.authenticator.is_logged_in("joe")
False
>>> auth.authenticator.login("joe", "joepassword")
True
>>> auth.authorizor.check_permission("paint", "joe")
Traceback (most recent call last):
  File "<stdin>" line 1, in <module>
  File "auth.py", line 116, in check_permission
    raise NotPermittedError(username)
auth.NotPermittedError: joe
>>> auth.authorizor.check_permission("mix", "joe")
Traceback (most recent call last):
  File "auth.py", line 111, in check_permission
    perm_set = self.permissions[perm_name]
```

---
KeyError: 'mix'

During handling of the above exception, another exception occurred:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "auth.py", line 113, in check_permission
    raise PermissionError("Permission does not exist")
auth.PermissionError: Permission does not exist

>>> auth.authorizor.permit_user("mix", "joe")
Traceback (most recent call last):
  File "auth.py", line 99, in permit_user
    perm_set = self.permissions[perm_name]
KeyError: 'mix'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "auth.py", line 101, in permit_user
    raise PermissionError("Permission does not exist")
auth.PermissionError: Permission does not exist

>>> auth.authorizor.permit_user("paint", "joe")
>>> auth.authorizor.check_permission("paint", "joe")
True

While verbose, the preceding output shows all of our code and most of our exceptions in action, but to really understand the API we've defined, we should write some exception handling code that actually uses it. Here's a basic menu interface that allows certain users to change or test a program:

```python
import auth

# Set up a test user and permission
auth.authenticator.add_user("joe", "joe_password")
auth.authorizor.add_permission("test program")
auth.authorizor.add_permission("change program")
auth.authorizor.permit_user("test program", "joe")

class Editor:
```

def __init__(self):
    self.username = None
    self.menu_map = {
        "login": self.login,
        "test": self.test,
        "change": self.change,
        "quit": self.quit
    }

def login(self):
    logged_in = False
    while not logged_in:
        username = input("username: ")
        password = input("password: ")
        try:
            logged_in = auth.authenticator.login(
                username, password)
        except auth.InvalidUsername:
            print("Sorry, that username does not exist")
        except auth.InvalidPassword:
            print("Sorry, incorrect password")
        else:
            self.username = username

def is_permitted(self, permission):
    try:
        auth.authorizor.check_permission(
            permission, self.username)
    except auth.NotLoggedInError as e:
        print("{} is not logged in".format(e.username))
        return False
    except auth.NotPermittedError as e:
        print("{} cannot {}".format(
            e.username, permission))
        return False
    else:
        return True

def test(self):
    if self.is_permitted("test program"):  
        print("Testing program now...")

def change(self):
    if self.is_permitted("change program"):  
        print("Changing program now...")

def quit(self):

raise SystemExit()

def menu(self):
    try:
        answer = ""
        while True:
            print(""
Please enter a command:
\tlogin\tLogin
\ttest\tTest the program
\tchange\tChange the program
\tquit\tQuit
"")
            answer = input("enter a command: ").lower()
            try:
                func = self.menu_map[answer]
            except KeyError:
                print("{} is not a valid option".format(answer))
                else:
                    func()
            finally:
                print("Thank you for testing the auth module")
    
Editor().menu()

This rather long example is conceptually very simple. The is_permitted method is probably the most interesting; this is a mostly internal method that is called by both test and change to ensure the user is permitted access before continuing. Of course, those two methods are stubs, but we aren't writing an editor here; we're illustrating the use of exceptions and exception handlers by testing an authentication and authorization framework!

Exercises

If you've never dealt with exceptions before, the first thing you need to do is look at any old Python code you've written and notice if there are places you should have been handling exceptions. How would you handle them? Do you need to handle them at all? Sometimes, letting the exception propagate to the console is the best way to communicate to the user, especially if the user is also the script's coder. Sometimes, you can recover from the error and allow the program to continue. Sometimes, you can only reformat the error into something the user can understand and display it to them.
Some common places to look are file I/O (is it possible your code will try to read a file that doesn't exist?), mathematical expressions (is it possible that a value you are dividing by is zero?), list indices (is the list empty?), and dictionaries (does the key exist?). Ask yourself if you should ignore the problem, handle it by checking values first, or handle it with an exception. Pay special attention to areas where you might have used finally and else to ensure the correct code is executed under all conditions.

Now write some new code. Think of a program that requires authentication and authorization, and try writing some code that uses the auth module we built in the case study. Feel free to modify the module if it's not flexible enough. Try to handle all the exceptions in a sensible way. If you're having trouble coming up with something that requires authentication, try adding authorization to the notepad example from Chapter 2, Objects in Python, or add authorization to the auth module itself—it's not a terribly useful module if just anybody can start adding permissions! Maybe require an administrator username and password before allowing privileges to be added or changed.

Finally, try to think of places in your code where you can raise exceptions. It can be in code you've written or are working on; or you can write a new project as an exercise. You'll probably have the best luck for designing a small framework or API that is meant to be used by other people; exceptions are a terrific communication tool between your code and someone else's. Remember to design and document any self-raised exceptions as part of the API, or they won't know whether or how to handle them!

**Summary**

In this chapter, we went into the gritty details of raising, handling, defining, and manipulating exceptions. Exceptions are a powerful way to communicate unusual circumstances or error conditions without requiring a calling function to explicitly check return values. There are many built-in exceptions and raising them is trivially easy. There are several different syntaxes for handling different exception events.

In the next chapter, everything we've studied so far will come together as we discuss how object-oriented programming principles and structures should best be applied in Python applications.
Where to buy this book


Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.