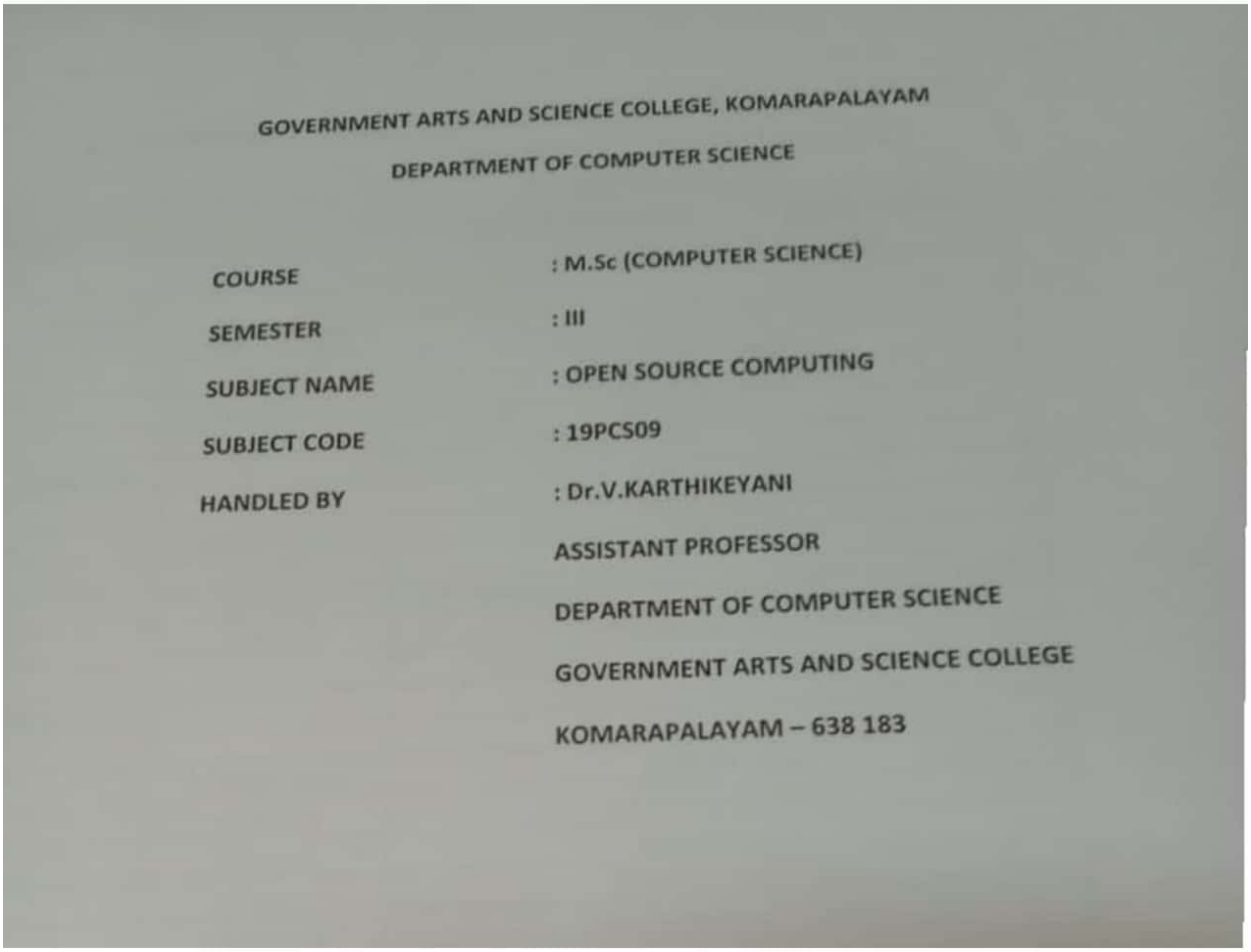

GOVERNMENT ARTS AND SCIENCE COLLEGE, KOMARAPALAYAM

DEPARTMENT OF COMPUTER SCIENCE

COURSE	: M.Sc (COMPUTER SCIENCE)
SEMESTER	: III
SUBJECT NAME	: OPEN SOURCE COMPUTING
SUBJECT CODE	: 19PCS09
HANDLED BY	: Dr.V.KARTHIKEYANI
	ASSISTANT PROFESSOR
	DEPARTMENT OF COMPUTER SCIENCE
	GOVERNMENT ARTS AND SCIENCE COLLEGE
	KOMARAPALAYAM – 638 183

Scanned with CamScanner

UNIT - II

Code Structures: if, elif, and else – Repeat with while – Iterate with for – Comprehensions – Functions – Generators – Decorators – Namespaces and Scope – Handle Errors with try and except – User Exceptions.

Modules, Packages, and Programs: Standalone Programs – Command-Line Arguments – Modules and the import Statement – The Python Standard Library. **Objects and Classes**: Define a Class with class – Inheritance – Override a Method – Add a Method – Get Help from Parent with super – In self Defense – Get and Set Attribute Values with Properties – Name Mangling for Privacy – Method Types – Duck Typing – Special Methods – Composition

Scanned with CamScanner

CHAPTER 4

Py Crust: Code Structures

In Chapters 1 through 3, you've seen many examples of data but have not done much with them. Most of the code examples used the interactive interpreter and were short. Now you'll see how to structure Python *code*, not just data.

Many computer languages use characters such as curly braces ({ and }) or keywords such as begin and end to mark off sections of code. In those languages, it's good practice to use consistent indentation to make your program more readable for yourself and others. There are even tools to make your code line up nicely.

When he was designing the language that became Python, Guido van Rossum decided that the indentation itself was enough to define a program's structure, and avoided typing all those parentheses and curly braces. Python is unusual in this use of *white space* to define program structure. It's one of the first aspects that newcomers notice, and it can seem odd to those who have experience with other languages. It turns out that after writing Python for a little while, it feels natural and you stop noticing it. You even get used to doing more while typing less.

Comment with

A *comment* is a piece of text in your program that is ignored by the Python interpreter. You might use comments to clarify nearby Python code, make notes to yourself to fix something someday, or for whatever purpose you like. You mark a comment by using the # character; everything from that point on to the end of the current line is part of the comment. You'll usually see a comment on a line by itself, as shown here:

```
>>> # 60 sec/min * 60 min/hr * 24 hr/day
>>> seconds_per_day = 86400
```

Or, on the same line as the code it's commenting:

```
>>> seconds_per_day = 86400 # 60 sec/min * 60 min/hr * 24 hr/day
```

Scanned with CamScanner
Scanned with CamScanner

The # character has many names: *hash*, *sharp*, *pound*, or the sinister-sounding *octothorpe*.[1] Whatever you call it,[2] its effect lasts only to the end of the line on which it appears.

Python does not have a multiline comment. You need to explicitly begin each comment line or section with a #.

```
>>> # I can say anything here, even if Python doesn't like it,
... # because I'm protected by the awesome
... # octothorpe.
...
>>>
```

However, if it's in a text string, the all-powerful octothorpe reverts back to its role as a plain old # character:

```
>>> print("No comment: quotes make the # harmless.")
No comment: quotes make the # harmless.
```

Continue Lines with \

Programs are more readable when lines are reasonably short. The recommended (not required) maximum line length is 80 characters. If you can't say everything you want to say in that length, you can use the *continuation character*: \ (backslash). Just put \ at the end of a line, and Python will suddenly act as though you're still on the same line.

For example, if I wanted to build a long string from smaller ones, I could do it in steps:

```
>>> alphabet = ''
>>> alphabet += 'abcdefg'
>>> alphabet += 'hijklmnop'
>>> alphabet += 'qrstuv'
>>> alphabet += 'wxyz'
```

Or, I could do it in one step, using the continuation character:

```
>>> alphabet = 'abcdefg' + \
...      'hijklmnop' + \
...      'qrstuv' + \
...      'wxyz'
```

1. Like that eight-legged green *thing* that's *right behind you*.
2. Please don't call it. It might come back.

Line continuation is also needed if a Python expression spans multiple lines:

```
>>> 1 + 2 +
  File "<stdin>", line
    1 + 2 +
          ^
SyntaxError: invalid syntax
>>> 1 + 2 + \
... 3
6
>>>
```

Compare with if, elif, and else

So far in this book, we've talked almost entirely about data structures. Now, we finally take our first step into the *code structures* that weave data into programs. (You got a little preview of these in the previous chapter's section on sets. I hope no lasting damage was done.) Our first example is this tiny Python program that checks the value of the boolean variable `disaster` and prints an appropriate comment:

```
>>> disaster = True
>>> if disaster:
...     print("Woe!")
... else:
...     print("Whee!")
...
Woe!
>>>
```

The `if` and `else` lines are Python *statements* that check whether a condition (here, the value of `disaster`) is `True`. Remember, `print()` is Python's built-in *function* to print things, normally to your screen.

If you've programmed in other languages, note that you don't need parentheses for the `if` test. Don't say something such as `if (disaster == True)`. You do need the colon (`:`) at the end. If, like me, you forget to type the colon at times, Python will display an error message.

Each `print()` line is indented under its test. I used four spaces to indent each subsection. Although you can use any indentation you like, Python expects you to be consistent with code within a section—the lines need to be indented the same amount, lined up on the left. The recommended style, called *PEP-8*, is to use four spaces. Don't use tabs, or mix tabs and spaces; it messes up the indent count.

We did a number of things here, which I'll explain more fully as the chapter progresses:

- Assigned the boolean value `True` to the variable named `disaster`
- Performed a *conditional comparison* by using `if` and `else`, executing different code depending on the value of `disaster`
- *Called* the `print()` *function* to print some text

You can have tests within tests, as many levels deep as needed:

```
>>> furry = True
>>> small = True
>>> if furry:
...     if small:
...         print("It's a cat.")
...     else:
...         print("It's a bear!")
... else:
...     if small:
...         print("It's a skink!")
...     else:
...         print("It's a human. Or a hairless bear.")
...
It's a cat.
```

In Python, indentation determines how the `if` and `else` sections are paired. Our first test was to check `furry`. Because `furry` is `True`, Python goes to the indented `if small` test. Because we had set `small` to `True`, `if small` is evaluated as `True`. This makes Python run the next line and print `It's a cat.`

If there are more than two possibilities to test, use `if`, `elif` (meaning *else if*), and `else`:

```
>>> color = "puce"
>>> if color == "red":
...     print("It's a tomato")
... elif color == "green":
...     print("It's a green pepper")
... elif color == "bee purple":
...     print("I don't know what it is, but only bees can see it")
... else:
...     print("I've never heard of the color", color)
...
I've never heard of the color puce
```

In the preceding example, we tested for equality with the `==` operator. Python's *comparison operators* are:

equality	`==`
inequality	`!=`
less than	`<`

less than or equal	`<=`
greater than	`>`
greater than or equal	`>=`
membership	`in ...`

These return the boolean values `True` or `False`. Let's see how these all work, but first, assign a value to `x`:

```
>>> x = [illegible]
```

Now, let's try some tests:

```
>>> x == [illegible]
False
>>> x == [illegible]
True
>>> [illegible] < x
True
>>> x < 10
True
```

Note that two equals signs (`==`) are used to *test equality*; remember, a single equals sign (`=`) is what you use to assign a value to a variable.

If you need to make multiple comparisons at the same time, you use the *boolean operators* `and`, `or`, and `not` to determine the final boolean result.

Boolean operators have lower *precedence* than the chunks of code that they're comparing. This means that the chunks are calculated first, then compared. In this example, because we set `x` to `7`, `5 < x` is calculated to be `True` and `x < 10` is also `True`, so we finally end up with `True and True`:

```
>>> [illegible] < x and x < [illegible]
True
```

As "Precedence" on page 23 points out, the easiest way to avoid confusion about precedence is to add parentheses:

```
>>> ([illegible] < x) and (x < [illegible])
True
```

Here are some other tests:

```
>>> [illegible] < x or x < 10
True
>>> [illegible] < x and x > [illegible]
False
>>> [illegible] < x and not x > [illegible]
True
```

If you're and-ing multiple comparisons with one variable, Python lets you do this:

```
>>> 5 < x < 10
True
```

It's the same as `5 < x and x < 10`. You can also write longer comparisons:

```
>>> 5 < x < 10 < 999
True
```

What Is True?

What if the element we're checking isn't a boolean? What does Python consider `True` and `False`?

A `false` value doesn't necessarily need to explicitly be `False`. For example, these are all considered `False`:

boolean	`False`
null	`None`
zero integer	`0`
zero float	`0.0`
empty string	`''`
empty list	`[]`
empty tuple	`()`
empty dict	`{}`
empty set	`set()`

Anything else is considered `True`. Python programs use this definition of "truthiness" (or in this case, "falsiness") to check for empty data structures as well as `False` conditions:

```
>>> some_list = []
>>> if some_list:
...     print("There's something in here")
... else:
...     print("Hey, it's empty!")
...
Hey, it's empty!
```

If what you're testing is an expression rather than a simple variable, Python evaluates the expression and returns a boolean result. So, if you type the following:

```
if color == "red":
```

Python evaluates `color == "red"`. In our example, we assigned the string `"puce"` to `color` earlier, so `color == "red"` is `False`, and Python moves on to the next test:

```
elif color == "green":
```

Repeat with while

Testing with `if`, `elif`, and `else` runs from top to bottom. Sometimes, we need to do something more than once. We need a *loop*, and the simplest looping mechanism in Python is `while`. Using the interactive interpreter, try this next example, which is a simple loop that prints the numbers from 1 to 5:

```
>>> count = 1
>>> while count <= 5:
...     print(count)
...     count += 1
...
1
2
3
4
5
>>>
```

We first assigned the value `1` to `count`. The `while` loop compared the value of `count` to 5 and continued if `count` was less than or equal to 5. Inside the loop, we printed the value of `count` and then *incremented* its value by one with the statement `count += 1`. Python goes back to the top of the loop, and again compares `count` with `5`. The value of `count` is now 2, so the contents of the `while` loop are again executed, and `count` is incremented to 3.

This continues until `count` is incremented from 5 to 6 at the bottom of the loop. On the next trip to the top, `count <= 5` is now `False`, and the `while` loop ends. Python moves on to the next lines.

Cancel with break

If you want to loop until something occurs, but you're not sure when that might happen, you can use an *infinite loop* with a `break` statement. This time we'll read a line of input from the keyboard via Python's `input()` function and then print it with the first letter capitalized. We break out of the loop when a line containing only the letter `q` is typed:

```
>>> while True:
...     stuff = input("String to capitalize [type q to quit]: ")
...     if stuff == "q":
...         break
...     print(stuff.capitalize())
...
String to capitalize [type q to quit]: test
Test
String to capitalize [type q to quit]: hey, it works
Hey, it works
String to capitalize [type q to quit]: q
>>>
```

Skip Ahead with continue

Sometimes you don't want to break out of a loop but just want to skip ahead to the next iteration for some reason. Here's a contrived example: let's read an integer, print its square if it's odd, and skip it if it's even. We even added a few comments. Again, we'll use q to stop the loop:

```
>>> while True:
...     value = input("Integer, please [q to quit]: ")
...     if value == 'q':      # quit
...         break
...     number = int(value)
...     if number % 2 == 0:   # an even number
...        continue
...     print(number, "squared is", number*number)
...
Integer, please [q to quit]: 1
1 squared is 1
Integer, please [q to quit]: 2
Integer, please [q to quit]: 3
3 squared is 9
Integer, please [q to quit]: 4
Integer, please [q to quit]: 5
5 squared is 25
Integer, please [q to quit]: q
>>>
```

Check break Use with else

If the while loop ended normally (no break call), control passes to an optional else. You use this when you've coded a while loop to check for something, and breaking as soon as it's found. The else would be run if the while loop completed but the object was not found:

```
>>> numbers = [1, 3, 5]
>>> position = 0
>>> while position < len(numbers):
...     number = numbers[position]
...     if number % 2 == 0:
...         print('Found even number', number)
...         break
...     position += 1
... else:  # break not called
...     print('No even number found')
...
No even number found
```

This use of `else` might seem nonintuitive. Consider it a *break checker*.

Iterate with for

Python makes frequent use of *iterators*, for good reason. They make it possible for you to traverse data structures without knowing how large they are or how they are implemented. You can even iterate over data that is created on the fly, allowing processing of data *streams* that would otherwise not fit in the computer's memory all at once.

It's legal Python to step through a sequence like this:

```
>>> rabbits = ['Flopsy', 'Mopsy', 'Cottontail', 'Peter']
>>> current = 0
>>> while current < len(rabbits):
...     print(rabbits[current])
...     current += 1
...
Flopsy
Mopsy
Cottontail
Peter
```

But there's a better, more Pythonic way:

```
>>> for rabbit in rabbits:
...     print(rabbit)
...
Flopsy
Mopsy
Cottontail
Peter
```

Lists such as `rabbits` are one of Python's *iterable* objects, along with strings, tuples, dictionaries, sets, and some other elements. Tuple or list iteration produces an item at a time. String iteration produces a character at a time, as shown here:

```
>>> word = 'cat'
>>> for letter in word:
...     print(letter)
...
c
a
t
```

Iterating over a dictionary (or its `keys()` function) returns the keys. In this example, the keys are the types of cards in the board game Clue (Cluedo outside of North America):

```
>>> accusation = {'room': 'ballroom', 'weapon': 'lead pipe',
                  'person': 'Col. Mustard'}
>>> for card in accusation:  # or, for card in accusation.keys():
...     print(card)
...
room
weapon
person
```

To iterate over the values rather than the keys, you use the dictionary's `values()` function:

```
>>> for value in accusation.values():
...     print(value)
...
ballroom
lead pipe
Col. Mustard
```

To return both the key and value in a tuple, you can use the `items()` function:

```
>>> for item in accusation.items():
...     print(item)
...
('room', 'ballroom')
('weapon', 'lead pipe')
('person', 'Col. Mustard')
```

Remember that you can assign to a tuple in one step. For each tuple returned by `items()`, assign the first value (the key) to card and the second (the value) to contents:

```
>>> for card, contents in accusation.items():
...     print('Card', card, 'has the contents', contents)
...
Card weapon has the contents lead pipe
Card person has the contents Col. Mustard
Card room has the contents ballroom
```

Cancel with break

A `break` in a `for` loop breaks out of the loop, as it does for a `while` loop.

Skip with continue

Inserting a `continue` in a `for` loop jumps to the next iteration of the loop, as it does for a `while` loop.

Check break Use with else

Similar to `while`, `for` has an optional `else` that checks if the `for` completed normally. If `break` was *not* called, the `else` statement is run.

This is useful when you want to verify that the previous for loop ran to completion, instead of being stopped early with a break. The for loop in the following example prints the name of the cheese and breaks if any cheese is found in the cheese shop:

```
>>> cheeses = []
>>> for cheese in cheeses:
...     print('This shop has some lovely', cheese)
...     break
... else:  # no break means no cheese
...     print('This is  not much of a cheese shop, is it?')
...
This is not much of a cheese shop, is it?
```

As with `while`, the use of `else` with `for` might seem nonintuitive. It makes more sense if you think of the `for` as looking for something, and `else` being called if you didn't find it. To get the same effect without `else`, use some variable to indicate whether you found what you wanted in the for loop, as demonstrated here:

```
>>> cheeses = []
>>> found_one = False
>>> for cheese in cheeses:
...     found_one = True
...     print('This shop has some lovely', cheese)
...     break
...
>>> if not found_one:
...     print('This is not much of a cheese shop, is it?')
...
This is not much of a cheese shop, is it?
```

Iterate Multiple Sequences with zip()

There's one more nice iteration trick: iterating over multiple sequences in parallel by using the `zip()` function:

```
>>> days = ['Monday', 'Tuesday', 'Wednesday']
>>> fruits = ['banana', 'orange', 'peach']
>>> drinks = ['coffee', 'tea', 'beer']
>>> desserts = ['tiramisu', 'ice cream', 'pie', 'pudding']
>>> for day, fruit, drink, dessert in zip(days, fruits, drinks, desserts):
...     print(day, ": drink", drink, "- eat", fruit, "- enjoy", dessert)
...
Monday : drink coffee - eat banana - enjoy tiramisu
Tuesday : drink tea - eat orange - enjoy ice cream
Wednesday : drink beer - eat peach - enjoy pie
```

`zip()` stops when the shortest sequence is done. One of the lists (`desserts`) was longer than the others, so no one gets any pudding unless we extend the other lists.

"Dictionaries" on page 53 shows you how the dict() function can create dictionaries from two-item sequences like tuples, lists, or strings. You can use zip() to walk through multiple sequences and make tuples from items at the same offsets. Let's make two tuples of corresponding English and French words:

```
>>> english = 'Monday', 'Tuesday', 'Wednesday'
>>> french = 'Lundi', 'Mardi', 'Mercredi'
```

Now, use zip() to pair these tuples. The value returned by zip() is itself not a tuple or list, but an iterable value that can be turned into one:

```
>>> list( zip(english, french) )
[('Monday', 'Lundi'), ('Tuesday', 'Mardi'), ('Wednesday', 'Mercredi')]
```

Feed the result of zip() directly to dict() and voilà: a tiny English-French dictionary!

```
>>> dict( zip(english, french) )
{'Monday': 'Lundi', 'Tuesday': 'Mardi', 'Wednesday': 'Mercredi'}
```

Generate Number Sequences with range()

The range() function returns a stream of numbers within a specified range. without first having to create and store a large data structure such as a list or tuple. This lets you create huge ranges without using all the memory in your computer and crashing your program.

You use range() similar to how to you use slices: range(*start*, *stop*, *step*). If you omit *start*, the range begins at 0. The only required value is *stop*; as with slices, the last value created will be just before *stop*. The default value of *step* is 1, but you can go backward with -1.

Like zip(), range() returns an *iterable* object, so you need to step through the values with for ... in, or convert the object to a sequence like a list. Let's make the range 0, 1, 2:

```
>>> for x in range(0,3):
...     print(x)
...
0
1
2
>>> list( range(0, 3) )
[0, 1, 2]
```

Here's how to make a range from 2 down to 0:

```
>>> for x in range(2, -1, -1):
...     print(x)
...
2
1
```

```
>>> list( range(2, -1, -1) )
[2, 1, 0]
```

The following snippet uses a step size of 2 to get the even numbers from 0 to 10:

```
>>> list( range(0, 11, 2) )
[0, 2, 4, 6, 8, 10]
```

Other Iterators

Chapter 8 shows iteration over files. In Chapter 6, you can see how to enable iteration over objects that you've defined yourself.

Comprehensions

A *comprehension* is a compact way of creating a Python data structure from one or more iterators. Comprehensions make it possible for you to combine loops and conditional tests with a less verbose syntax. Using a comprehension is sometimes taken as a sign that you know Python at more than a beginner's level. In other words, it's more Pythonic.

List Comprehensions

You could build a list of integers from 1 to 5, one item at a time, like this:

```
>>> number_list = []
>>> number_list.append(1)
>>> number_list.append(2)
>>> number_list.append(3)
>>> number_list.append(4)
>>> number_list.append(5)
>>> number_list
[1, 2, 3, 4, 5]
```

Or, you could also use an iterator and the range() function:

```
>>> number_list = []
>>> for number in range(1, 6):
...     number_list.append(number)
...
>>> number_list
[1, 2, 3, 4, 5]
```

Or, you could just turn the output of range() into a list directly:

```
>>> number_list = list(range(1, 6))
>>> number_list
[1, 2, 3, 4, 5]
```

All of these approaches are valid Python code and will produce the same result. However, a more Pythonic way to build a list is by using a *list comprehension*. The simplest form of list comprehension is:

```
[ expression for item in iterable ]
```

Here's how a list comprehension would build the integer list:

```
>>> number_list = [number for number in range(1,6)]
>>> number_list
[1, 2, 3, 4, 5]
```

In the first line, you need the first `number` variable to produce values for the list: that is, to put a result of the loop into `number_list`. The second `number` is part of the `for` loop. To show that the first `number` is an expression, try this variant:

```
>>> number_list = [number-1 for number in range(1,6)]
>>> number_list
[0, 1, 2, 3, 4]
```

The list comprehension moves the loop inside the square brackets. This comprehension example really wasn't simpler than the previous example, but there's more. A list comprehension can include a conditional expression, looking something like this:

```
[ expression for item in iterable if condition ]
```

Let's make a new comprehension that builds a list of only the odd numbers between 1 and 5 (remember that `number % 2` is `True` for odd numbers and `False` for even numbers):

```
>>> a_list = [number for number in range(1,6) if number % 2 == 1]
>>> a_list
[1, 3, 5]
```

Now, the comprehension is a little more compact than its traditional counterpart:

```
>>> a_list = []
>>> for number in range(1,6):
...     if number % 2 == 1:
...         a_list.append(number)
...
>>> a_list
[1, 3, 5]
```

Finally, just as there can be nested loops, there can be more than one set of `for ...` clauses in the corresponding comprehension. To show this, let's first try a plain, old nested loop and print the results:

```
>>> rows = range(1,4)
>>> cols = range(1,3)
>>> for row in rows:
...     for col in cols:
...         print(row, col)
```

Now, let's use a comprehension and assign it to the variable `cells`, making it a list of `(row, col)` tuples:

```
>>> rows = range(1,4)
>>> cols = range(1,3)
>>> cells = [(row, col) for row in rows for col in cols]
>>> for cell in cells:
...     print(cell)
...
(1, 1)
(1, 2)
(2, 1)
(2, 2)
(3, 1)
(3, 2)
```

By the way, you can also use tuple unpacking to yank the `row` and `col` values from each tuple as you iterate over the `cells` list:

```
>>> for row, col in cells:
...     print(row, col)
...
1 1
1 2
2 1
2 2
3 1
3 2
```

The `for row ...` and `for col ...` fragments in the list comprehension could also have had their own `if` tests.

Dictionary Comprehensions

Not to be outdone by mere lists, dictionaries also have comprehensions. The simplest form looks familiar:

`{` *`key_expression`* `:` *`value_expression`* `for` *`expression`* `in` *`iterable`* `}`

Similar to list comprehensions, dictionary comprehensions can also have `if` tests and multiple `for` clauses:

```
>>> word = 'letters'
>>> letter_counts = {letter: word.count(letter) for letter in word}
>>> letter_counts
{'l': 1, 'e': 2, 't': 2, 'r': 1, 's': 1}
```

We are running a loop over each of the seven letters in the string `'letters'` and counting how many times that letter appears. Two of our uses of `word.count(letter)` are a waste of time because we have to count all the e's twice and all the t's twice. But, when we count the e's the second time, we do no harm because we just replace the entry in the dictionary that was already there; the same goes for counting the t's. So, the following would have been a teeny bit more Pythonic:

```
>>> word = 'letters'
>>> letter_counts = {letter: word.count(letter) for letter in set(word)}
>>> letter_counts
{'t': 2, 'l': 1, 'e': 2, 'r': 1, 's': 1}
```

The dictionary's keys are in a different order than the previous example, because iterating `set(word)` returns letters in a different order than iterating the string `word`.

Set Comprehensions

No one wants to be left out, so even sets have comprehensions. The simplest version looks like the list and dictionary comprehensions that you've just seen:

`{ expression for expression in iterable }`

The longer versions (`if` tests, multiple `for` clauses) are also valid for sets:

```
>>> a_set = {number for number in range(1,6) if number % 3 == 1}
>>> a_set
{1, 4}
```

Generator Comprehensions

Tuples do not have comprehensions! You might have thought that changing the square brackets of a list comprehension to parentheses would create a tuple comprehension. And it would appear to work because there's no exception if you type this:

```
>>> number_thing = (number for number in range(1, 6))
```

The thing between the parentheses is a *generator comprehension*, and it returns a *generator object*:

```
>>> type(number_thing)
<class 'generator'>
```

I'll get into generators in more detail in "Generators" on page 98. A generator is one way to provide data to an iterator.

You can iterate over this generator object directly, as illustrated here:

```
>>> for number in number_thing:
...     print(number)
...
1
2
3
4
5
```

Or, you can wrap a `list()` call around a generator comprehension to make it work like a list comprehension:

```
>>> number_list = list(number_thing)
>>> number_list
[1, 2, 3, 4, 5]
```

A generator can be run only once. Lists, sets, strings, and dictionaries exist in memory, but a generator creates its values on the fly and hands them out one at a time through an iterator. It doesn't remember them, so you can't restart or back up a generator.

If you try to re-iterate this generator, you'll find that it's tapped out:

```
>>> try_again = list(number_thing)
>>> try_again
[]
```

You can create a generator from a generator comprehension, as we did here, or from a *generator function*. We'll talk about functions in general first, and then we'll get to the special case of generator functions.

Functions

So far, all our Python code examples have been little fragments. These are good for small tasks, but no one wants to retype fragments all the time. We need some way of organizing larger code into manageable pieces.

The first step to code reuse is the *function*: a named piece of code, separate from all others. A function can take any number and type of input *parameters* and return any number and type of output *results*.

You can do two things with a function:

- *Define* it
- *Call* it

To define a Python function, you type `def`, the function name, parentheses enclosing any input parameters to the function, and then finally, a colon (`:`). Function names have the same rules as variable names (they must start with a letter or `_` and contain only letters, numbers, or `_`).

Let's take things one step at a time, and first define and call a function that has no parameters. Here's the simplest Python function:

```
>>> def do_nothing():
...     pass
```

Even for a function with no parameters like this one, you still need the parentheses and the colon in its definition. The next line needs to be indented, just as you would indent code under an `if` statement. Python requires the `pass` statement to show that this function does nothing. It's the equivalent of *This page intentionally left blank* (even though it isn't anymore).

You call this function just by typing its name and parentheses. It works as advertised, doing nothing very well:

```
>>> do_nothing()
>>>
```

Now, let's define and call another function that has no parameters but prints a single word:

```
>>> def make_a_sound():
...     print('quack')
...
>>> make_a_sound()
quack
```

When you called the `make_a_sound()` function, Python ran the code inside its definition. In this case, it printed a single word and returned to the main program.

Let's try a function that has no parameters but *returns* a value:

```
>>> def agree():
...    return True
...
```

You can call this function and test its returned value by using `if`:

```
>>> if agree():
...     print('Splendid!')
... else:
...     print('That was unexpected.')
...
Splendid!
```

You've just made a big step. The combination of functions with tests such as `if` and loops such as `while` make it possible for you to do things that you could not do before.

At this point, it's time to put something between those parentheses. Let's define the function echo() with one parameter called anything. It uses the return statement to send the value of anything back to its caller twice, with a space between:

```
>>> def echo(anything):
...     return anything + ' ' + anything
...
>>>
```

Now let's call echo() with the string 'Rumplestiltskin':

```
>>> echo('Rumplestiltskin')
'Rumplestiltskin Rumplestiltskin'
```

The values you pass into the function when you call it are known as *arguments*. When you call a function with arguments, the values of those arguments are copied to their corresponding *parameters* inside the function. In the previous example, the function echo() was called with the argument string 'Rumplestiltskin'. This value was copied within echo() to the parameter anything, and then returned (in this case doubled, with a space) to the caller.

These function examples were pretty basic. Let's write a function that takes an input argument and actually does something with it. We'll adapt the earlier code fragment that comments on a color. Call it commentary and have it take an input string parameter called color. Make it return the string description to its caller, which can decide what to do with it:

```
>>> def commentary(color):
...     if color == 'red':
...         return "It's a tomato."
...     elif color == "green":
...         return "It's a green pepper."
...     elif color == 'bee purple':
...         return "I don't know what it is, but only bees can see it."
...     else:
...         return "I've never heard of the color " + color + "."
...
>>>
```

Call the function commentary() with the string argument 'blue'.

```
>>> comment = commentary('blue')
```

The function does the following:

- Assigns the value 'blue' to the function's internal color parameter
- Runs through the if-elif-else logic chain
- Returns a string

- Assigns the string to the variable `comment`

What do we get back?

```
>>> print(comment)
I've never heard of the color blue.
```

A function can take any number of input arguments (including zero) of any type. It can return any number of output results (also including zero) of any type. If a function doesn't call `return` explicitly, the caller gets the result `None`.

```
>>> print(do_nothing())
None
```

None Is Useful

`None` is a special Python value that holds a place when there is nothing to say. It is not the same as the boolean value `False`, although it looks false when evaluated as a boolean. Here's an example:

```
>>> thing = None
>>> if thing:
...     print("It's some thing")
... else:
...     print("It's no thing")
...
It's no thing
```

To distinguish `None` from a boolean `False` value, use Python's `is` operator:

```
>>> if thing is None:
...     print("It's nothing")
... else:
...     print("It's something")
...
It's nothing
```

This seems like a subtle distinction, but it's important in Python. You'll need `None` to distinguish a missing value from an empty value. Remember that zero-valued integers or floats, empty strings (`''`), lists (`[]`), tuples (`(,)`), dictionaries (`{}`), and sets(`set()`) are all `False`, but are not equal to `None`.

Let's write a quick function that prints whether its argument is `None`:

```
>>> def is_none(thing):
...     if thing is None:
...         print("It's None")
...     elif thing:
...         print("It's True")
...     else:
...         print("It's False")
...
```

Now, let's run some tests:

```
>>> is_none(None)
It's None
>>> is_none(True)
It's True
>>> is_none(False)
It's False
>>> is_none(0)
It's False
>>> is_none(0.0)
It's False
>>> is_none(())
It's False
>>> is_none([])
It's False
>>> is_none({})
It's False
>>> is_none(set())
It's False
```

Positional Arguments

Python handles function arguments in a manner that's unusually flexible, when compared to many languages. The most familiar types of arguments are *positional arguments*, whose values are copied to their corresponding parameters in order.

This function builds a dictionary from its positional input arguments and returns it:

```
>>> def menu(wine, entree, dessert):
...     return {'wine': wine, 'entree': entree, 'dessert': dessert}
...
>>> menu('chardonnay', 'chicken', 'cake')
{'dessert': 'cake', 'wine': 'chardonnay', 'entree': 'chicken'}
```

Although very common, a downside of positional arguments is that you need to remember the meaning of each position. If we forgot and called `menu()` with wine as the last argument instead of the first, the meal would be very different:

```
>>> menu('beef', 'bagel', 'bordeaux')
{'dessert': 'bordeaux', 'wine': 'beef', 'entree': 'bagel'}
```

Keyword Arguments

To avoid positional argument confusion, you can specify arguments by the names of their corresponding parameters, even in a different order from their definition in the function:

```
>>> menu(entree='beef', dessert='bagel', wine='bordeaux')
{'dessert': 'bagel', 'wine': 'bordeaux', 'entree': 'beef'}
```

Scanned with CamScanner
Scanned with CamScanner

You can mix positional and keyword arguments. Let's specify the wine first, but use keyword arguments for the entree and dessert:

```
>>> menu('frontenac', dessert='flan', entree='fish')
{'entree': 'fish', 'dessert': 'flan', 'wine': 'frontenac'}
```

If you call a function with both positional and keyword arguments, the positional arguments need to come first.

Specify Default Parameter Values

You can specify default values for parameters. The default is used if the caller does not provide a corresponding argument. This bland-sounding feature can actually be quite useful. Using the previous example:

```
>>> def menu(wine, entree, dessert='pudding'):
...     return {'wine': wine, 'entree': entree, 'dessert': dessert}
```

This time, try calling `menu()` without the `dessert` argument:

```
>>> menu('chardonnay', 'chicken')
{'dessert': 'pudding', 'wine': 'chardonnay', 'entree': 'chicken'}
```

If you do provide an argument, it's used instead of the default:

```
>>> menu('dunkelfelder', 'duck', 'doughnut')
{'dessert': 'doughnut', 'wine': 'dunkelfelder', 'entree': 'duck'}
```

Default argument values are calculated when the function is *defined*, not when it is run. A common error with new (and sometimes not-so-new) Python programmers is to use a mutable data type such as a list or dictionary as a default argument.

In the following test, the `buggy()` function is expected to run each time with a fresh empty `result` list, add the `arg` argument to it, and then print a single-item list. However, there's a bug: it's empty only the first time it's called. The second time, `result` still has one item from the previous call:

```
>>> def buggy(arg, result=[]):
...     result.append(arg)
...     print(result)
...
>>> buggy('a')
['a']
>>> buggy('b')   # expect ['b']
['a', 'b']
```

It would have worked if it had been written like this:

```
>>> def works(arg):
...     result = []
...     result.append(arg)
...     return result
...
>>> works('a')
['a']
>>> works('b')
['b']
```

The fix is to pass in something else to indicate the first call:

```
>>> def nonbuggy(arg, result=None):
...     if result is None:
...         result = []
...     result.append(arg)
...     print(result)
...
>>> nonbuggy('a')
['a']
>>> nonbuggy('b')
['b']
```

Gather Positional Arguments with *

If you've programmed in C or C++, you might assume that an asterisk (*) in a Python program has something to do with a pointer. Nope, Python doesn't have pointers.

When used inside the function with a parameter, an asterisk groups a variable number of positional arguments into a tuple of parameter values. In the following example, args is the parameter tuple that resulted from the arguments that were passed to the function print_args():

```
>>> def print_args(*args):
...     print('Positional argument tuple:', args)
...
```

If you call it with no arguments, you get nothing in *args:

```
>>> print_args()
Positional argument tuple: ()
```

Whatever you give it will be printed as the args tuple:

```
>>> print_args(3, 2, 1, 'wait!', 'uh...')
Positional argument tuple: (3, 2, 1, 'wait!', 'uh...')
```

This is useful for writing functions such as print() that accept a variable number of arguments. If your function has required positional arguments as well, *args goes at the end and grabs all the rest:

```
>>> def print_more(required1, required2, *args):
...     print('Need this one:', required1)
...     print('Need this one too:', required2)
...     print('All the rest:', args)
...
>>> print_more('cap', 'gloves', 'scarf', 'monocle', 'mustache wax')
Need this one: cap
Need this one too: gloves
All the rest: ('scarf', 'monocle', 'mustache wax')
```

When using *, you don't need to call the tuple parameter args, but it's a common idiom in Python.

Gather Keyword Arguments with **

You can use two asterisks (**) to group keyword arguments into a dictionary, where the argument names are the keys, and their values are the corresponding dictionary values. The following example defines the function print_kwargs() to print its keyword arguments:

```
>>> def print_kwargs(**kwargs):
...     print('Keyword arguments:', kwargs)
...
```

Now, try calling it with some keyword arguments:

```
>>> print_kwargs(wine='merlot', entree='mutton', dessert='macaroon')
Keyword arguments: {'dessert': 'macaroon', 'wine': 'merlot', 'entree': 'mutton'}
```

Inside the function, kwargs is a dictionary.

If you mix positional parameters with *args and **kwargs, they need to occur in that order. As with args, you don't need to call this keyword parameter kwargs, but it's common usage.

Docstrings

Readability counts, says the Zen of Python. You can attach documentation to a function definition by including a string at the beginning of the function body. This is the function's *docstring*:

```
>>> def echo(anything):
...     'echo returns its input argument'
...     return anything
```

You can make a docstring quite long and even add rich formatting, if you want, as is demonstrated in the following:

```
def print_if_true(thing, check):
    '''
    Prints the first argument if a second argument is true.
    The operation is:
        1. Check whether the *second* argument is true.
```

```
...     2. If it is, print the *first* argument.
    if check:
        print(thing)
```

To print a function's docstring, call the Python `help()` function. Pass the function's name to get a listing of arguments along with the nicely formatted docstring:

```
>>> help(echo)
Help on function echo in module __main__:

echo(anything)
    echo returns its input argument
```

If you want to see just the raw docstring, without the formatting:

```
>>> print(echo.__doc__)
echo returns its input argument
```

That odd-looking `__doc__` is the internal name of the docstring as a variable within the function. "Uses of _ and __ in Names" on page 103 explains the reason behind all those underscores.

Functions Are First-Class Citizens

I've mentioned the Python mantra, *everything is an object*. This includes numbers, strings, tuples, lists, dictionaries—and functions, as well. Functions are first-class citizens in Python. You can assign them to variables, use them as arguments to other functions, and return them from functions. This gives you the capability to do some things in Python that are difficult-to-impossible to carry out in many other languages.

To test this, let's define a simple function called `answer()` that doesn't have any arguments; it just prints the number 42:

```
>>> def answer():
...     print(42)
```

If you run this function, you know what you'll get:

```
>>> answer()
42
```

Now, let's define another function named `run_something`. It has one argument called `func`, a function to run. Once inside, it just calls the function.

```
>>> def run_something(func):
...     func()
```

If we pass `answer` to `run_something()`, we're using a function as data, just as with anything else:

```
>>> run_something(answer)
42
```

Notice that you passed answer, not answer(). In Python, those parentheses mean *call this function*. With no parentheses, Python just treats the function like any other object. That's because, like everything else in Python, it *is* an object:

```
>>> type(run_something)
<class 'function'>
```

Let's try running a function with arguments. Define a function add_args() that prints the sum of its two numeric arguments, arg1 and arg2:

```
>>> def add_args(arg1, arg2):
...     print(arg1 + arg2)
```

And what is add_args()?

```
>>> type(add_args)
<class 'function'>
```

At this point, let's define a function called run_something_with_args() that takes three arguments:

- func—The function to run
- arg1—The first argument for func
- arg2—The second argument for func

```
>>> def run_something_with_args(func, arg1, arg2):
...     func(arg1, arg2)
```

When you call run_something_with_args(), the function passed by the caller is assigned to the func parameter, whereas arg1 and arg2 get the values that follow in the argument list. Then, running func(arg1, arg2) executes that function with those arguments because the parentheses told Python to do so.

Let's test it by passing the function name add_args and the arguments 5 and 9 to run_something_with_args():

```
>>> run_something_with_args(add_args, 5, 9)
14
```

Within the function run_something_with_args(), the function name argument add_args was assigned to the parameter func, 5 to arg1, and 9 to arg2. This ended up running:

```
add_args(5, 9)
```

You can combine this with the *args and **kwargs techniques.

Let's define a test function that takes any number of positional arguments, calculates their sum by using the sum() function, and then returns that sum:

```
>>> def sum_args(*args):
...     return sum(args)
```

I haven't mentioned sum() before. It's a built-in Python function that calculates the sum of the values in its iterable numeric (int or float) argument.

We'll define the new function run_with_positional_args(), which takes a function and any number of positional arguments to pass to it:

```
>>> def run_with_positional_args(func, *args):
...     return func(*args)
```

Now, go ahead and call it:

```
>>> run_with_positional_args(sum_args, 1, 2, 3, 4)
10
```

You can use functions as elements of lists, tuples, sets, and dictionaries. Functions are immutable, so you can also use them as dictionary keys.

Inner Functions

You can define a function within another function:

```
>>> def outer(a, b):
...     def inner(c, d):
...         return c + d
...     return inner(a, b)
...
>>>
>>> outer(4, 7)
11
```

An inner function can be useful when performing some complex task more than once within another function, to avoid loops or code duplication. For a string example, this inner function adds some text to its argument:

```
>>> def knights(saying):
...     def inner(quote):
...         return "We are the knights who say: '%s'" % quote
...     return inner(saying)
...
>>> knights('Ni!')
"We are the knights who say: 'Ni!'"
```

Closures

An inner function can act as a *closure*. This is a function that is dynamically generated by another function and can both change and remember the values of variables that were created outside the function.

The following example builds on the previous knights() example. Let's call the new one knights2(), because we have no imagination, and turn the inner() function into a closure called inner2(). Here are the differences:

- inner2() uses the outer saying parameter directly instead of getting it as an argument.
- knights2() returns the inner2 function name instead of calling it.

```
>>> def knights2(saying):
...     def inner2():
...         return "We are the knights who say: '%s'" % saying
...     return inner2
...
```

The inner2() function knows the value of saying that was passed in and remembers it. The line return inner2 returns this specialized copy of the inner2 function (but doesn't call it). That's a closure: a dynamically created function that remembers where it came from.

Let's call knights2() twice, with different arguments:

```
>>> a = knights2('Duck')
>>> b = knights2('Hasenpfeffer')
```

Okay, so what are a and b?

```
>>> type(a)
<class 'function'>
>>> type(b)
<class 'function'>
```

They're functions, but they're also closures:

```
>>> a
<function knights2.<locals>.inner2 at 0x10193e158>
>>> b
<function knights2.<locals>.inner2 at 0x10193e1e0>
```

If we call them, they remember the saying that was used when they were created by knights2:

```
>>> a()
"We are the knights who say: 'Duck'"
>>> b()
"We are the knights who say: 'Hasenpfeffer'"
```

Anonymous Functions: the lambda() Function

In Python, a *lambda function* is an anonymous function expressed as a single statement. You can use it instead of a normal tiny function.

To illustrate it, let's first make an example that uses normal functions. To begin, we'll define the function `edit_story()`. Its arguments are the following:

- `words`—a list of words
- `func`—a function to apply to each word in `words`

```
>>> def edit_story(words, func):
...     for word in words:
...         print(func(word))
```

Now, we need a list of words and a function to apply to each word. For the words, here's a list of (hypothetical) sounds made by my cat if he (hypothetically) missed one of the stairs:

Eg colours [Blue, red, green]

```
>>> stairs = ['thud', 'meow', 'thud', 'hiss']
```

And for the function, this will capitalize each word and append an exclamation point, perfect for feline tabloid newspaper headlines:

```
>>> def enliven(word):   # give that prose more punch
...     return word.capitalize() + '!'
```

Mixing our ingredients:

```
>>> edit_story(stairs, enliven)
Thud!
Meow!
Thud!
Hiss!
```

Finally, we get to the lambda. The `enliven()` function was so brief that we could replace it with a lambda:

```
>>>
>>> edit_story(stairs, lambda word: word.capitalize() + '!')
Thud!
Meow!
Thud!
Hiss!
>>>
```

The lambda takes one argument, which we call `word` here. Everything between the colon and the terminating parenthesis is the definition of the function.

Often, using real functions such as `enliven()` is much clearer than using lambdas. Lambdas are mostly useful for cases in which you would otherwise need to define many tiny functions and remember what you called them all. In particular, you can use lambdas in graphical user interfaces to define *callback functions*; see Appendix A for examples.

Generators

A *generator* is a Python sequence creation object. With it, you can iterate through potentially huge sequences without creating and storing the entire sequence in memory at once. Generators are often the source of data for iterators. If you recall, we already used one of them, range(), in earlier code examples to generate a series of integers. In Python 2, range() returns a list, which limits it to fit in memory. Python 2 also has the generator xrange(), which became the normal range() in Python 3. This example adds all the integers from 1 to 100:

```
>>> sum(range(1, 101))
5050
```

Every time you iterate through a generator, it keeps track of where it was the last time it was called and returns the next value. This is different from a normal function, which has no memory of previous calls and always starts at its first line with the same state.

If you want to create a potentially large sequence, and the code is too large for a generator comprehension, write a *generator function*. It's a normal function, but it returns its value with a yield statement rather than return. Let's write our own version of range():

```
>>> def my_range(first=0, last=10, step=1):
...     number = first
...     while number < last:
...         yield number
...         number += step
...
```

It's a normal function:

```
>>> my_range
<function my_range at 0x10193e268>
```

And it returns a generator object:

```
>>> ranger = my_range(1, 5)
>>> ranger
<generator object my_range at 0x101a0a168>
```

We can iterate over this generator object:

```
>>> for x in ranger:
...     print(x)
...
1
2
3
4
```

Decorators

2018 2 5m.

Sometimes, you want to modify an existing function without changing its source code. A common example is adding a debugging statement to see what arguments were passed in.

A *decorator* is a function that takes one function as input and returns another function. We'll dig into our bag of Python tricks and use the following:

- `*args` and `**kwargs`
- Inner functions
- Functions as arguments

The function `document_it()` defines a decorator that will do the following:

- Print the function's name and the values of its arguments
- Run the function with the arguments
- Print the result
- Return the modified function for use

Here's what the code looks like:

```
>>> def document_it(func):
...     def new_function(*args, **kwargs):
...         print('Running function:', func.__name__)
...         print('Positional arguments:', args)
...         print('Keyword arguments:', kwargs)
...         result = func(*args, **kwargs)
...         print('Result:', result)
...         return result
...     return new_function
```

Whatever `func` you pass to `document_it()`, you get a new function that includes the extra statements that `document_it()` adds. A decorator doesn't actually have to run any code from `func`, but `document_it()` calls `func` part way through so that you get the results of `func` as well as all the extras.

So, how do you use this? You can apply the decorator manually:

```
>>> def add_ints(a, b):
...     return a + b
...
>>> add_ints([illegible], [illegible])
3
>>> cooler_add_ints = document_it(add_ints)  # manual decorator assignment
>>> cooler_add_ints(3, [illegible])
Running function: add_ints
```

Scanned with CamScanner
Scanned with CamScanner

```
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
8
```

As an alternative to the manual decorator assignment above, just add `@decorator_name` before the function that you want to decorate:

```
>>> @document_it
... def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
Start function add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
8
```

You can have more than one decorator for a function. Let's write another decorator called `square_it()` that squares the result:

```
>>> def square_it(func):
...     def new_function(*args, **kwargs):
...         result = func(*args, **kwargs)
...         return result * result
...     return new_function
...
```

The decorator that's used closest to the function (just above the `def`) runs first and then the one above it. Either order gives the same end result, but you can see how the intermediate steps change:

```
>>> @document_it
... @square_it
... def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
Running function: new_function
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 64
64
```

Let's try reversing the decorator order:

```
>>> @square_it
... @document_it
... def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
Running function: add_ints
```

```
Positional arguments: ([illegible], [illegible])
Keyword arguments: {}
Result: 8
64
```

Namespaces and Scope

A name can refer to different things, depending on where it's used. Python programs have various *namespaces*—sections within which a particular name is unique and unrelated to the same name in other namespaces.

Each function defines its own namespace. If you define a variable called x in a main program and another variable called x in a function, they refer to different things. But the walls can be breached: if you need to, you can access names in other namespaces in various ways.

The main part of a program defines the *global* namespace; thus, the variables in that namespace are *global variables*.

You can get the value of a global variable from within a function:

```
>>> animal = 'fruitbat'
>>> def print_global():
...     print('inside print_global:', animal)
...
>>> print('at the top level:', animal)
at the top level: fruitbat
>>> print_global()
inside print_global: fruitbat
```

But, if you try to get the value of the global variable *and* change it within the function, you get an error:

```
>>> def change_and_print_global():
...     print('inside change_and_print_global:', animal)
...     animal = 'wombat'
...     print('after the change:', animal)
...
>>> change_and_print_global()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in change_and_report_it
UnboundLocalError: local variable 'animal' referenced before assignment
```

If you just change it, it changes a different variable also named animal, but this variable is inside the function:

```
>>> def change_local():
...     animal = 'wombat'
...     print('inside change_local:', animal, id(animal))
...
```

```
>>> change_local()
inside change_local: wombat 4330406160
>>> animal
'fruitbat'
>>> id(animal)
4330390832
```

What happened here? The first line assigned the string `'fruitbat'` to a global variable named `animal`. The `change_local()` function also has a variable named `animal`, but that's in its local namespace.

We used the Python function `id()` here to print the unique value for each object and prove that the variable `animal` inside `change_local()` is not the same as `animal` at the main level of the program.

To access the global variable rather than the local one within a function, you need to be explicit and use the `global` keyword (you knew this was coming: *explicit is better than implicit*):

```
>>> animal = 'fruitbat'
>>> def change_and_print_global():
...     global animal
...     animal = 'wombat'
...     print('inside change_and_print_global:', animal)
...
>>> animal
'fruitbat'
>>> change_and_print_global()
inside change_and_print_global: wombat
>>> animal
'wombat'
```

If you don't say `global` within a function, Python uses the local namespace and the variable is local. It goes away after the function completes.

Python provides two functions to access the contents of your namespaces:

- `locals()` returns a dictionary of the contents of the local namespace.
- `globals()` returns a dictionary of the contents of the global namespace.

And, here they are in use:

```
>>> animal = 'fruitbat'
>>> def change_local():
...     animal = 'wombat'  # local variable
...     print('locals:', locals())
...
>>> animal
'fruitbat'
>>> change_local()
locals: {'animal': 'wombat'}
```

```
>>> print('globals:', globals()) # reformatted a little for presentation
globals: {'animal': 'fruitbat',
'__doc__': None,
'change_local': <function change_it at [illegible]>,
'__package__': None,
'__name__': '__main__',
'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__builtins__': <module 'builtins'>}
>>> animal
'fruitbat'
```

The local namespace within `change_local()` contained only the local variable `animal`. The global namespace contained the separate global variable `animal` and a number of other things.

Uses of _ and __ in Names

Names that begin and end with two underscores (`__`) are reserved for use within Python, so you should not use them with your own variables. This naming pattern was chosen because it seemed unlikely to be selected by application developers for their own variables.

For instance, the name of a function is in the system variable *`function`*`.__name__`, and its documentation string is *`function`*`.__doc__`:

```
>>> def amazing():
...     '''This is the amazing function.
...     Want to see it again?'''
...     print('This function is named:', amazing.__name__)
...     print('And its docstring is:', amazing.__doc__)
...
>>> amazing()
This function is named: amazing
And its docstring is: This is the amazing function.
    Want to see it again?
```

As you saw in the earlier `globals` printout, the main program is assigned the special name `__main__`.

Handle Errors with try and except

Do, or do not. There is no try.

— Yoda

In some languages, errors are indicated by special function return values. Python uses *exceptions*: code that is executed when an associated error occurs.

You've seen some of these already, such as accessing a list or tuple with an out-of-range position, or a dictionary with a nonexistent key. When you run code that might fail under some circumstances, you also need appropriate *exception handlers* to intercept any potential errors.

It's good practice to add exception handling anywhere an exception might occur to let the user know what is happening. You might not be able to fix the problem, but at least you can note the circumstances and shut your program down gracefully. If an exception occurs in some function and is not caught there, it *bubbles up* until it is caught by a matching handler in some calling function. If you don't provide your own exception handler, Python prints an error message and some information about where the error occurred and then terminates the program, as demonstrated in the following snippet.

```
>>> short_list = [1, 2, 3]
>>> position = 5
>>> short_list[position]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Rather than leaving it to chance, use `try` to wrap your code, and `except` to provide the error handling:

```
>>> short_list = [1, 2, 3]
>>> position = 5
>>> try:
...     short_list[position]
... except:
...     print('Need a position between 0 and', len(short_list)-1, ' but got',
...            position)
...
Need a position between 0 and 2 but got 5
```

The code inside the `try` block is run. If there is an error, an exception is raised and the code inside the `except` block runs. If there are no errors, the `except` block is skipped.

Specifying a plain `except` with no arguments, as we did here, is a catchall for any exception type. If more than one type of exception could occur, it's best to provide a separate exception handler for each. No one forces you to do this; you can use a bare `except` to catch all exceptions, but your treatment of them would probably be generic (something akin to printing *Some error occurred*). You can use any number of specific exception handlers.

Sometimes, you want exception details beyond the type. You get the full exception object in the variable *name* if you use the form:

```
except exceptiontype as name
```

The example that follows looks for an `IndexError` first, because that's the exception type raised when you provide an illegal position to a sequence. It saves an `IndexError` exception in the variable `err`, and any other exception in the variable `other`. The example prints everything stored in `other` to show what you get in that object.

```
>>> short_list = [1, 2, 3]
>>> while True:
...     value = input('Position [q to quit]? ')
...     if value == 'q':
...         break
...     try:
...         position = int(value)
...         print(short_list[position])
...     except IndexError as err:
...         print('Bad index:', position)
...     except Exception as other:
...         print('Something else broke:', other)
...
Position [q to quit]? 1
2
Position [q to quit]? 0
1
Position [q to quit]? 2
3
Position [q to quit]? 3
Bad index: 3
Position [q to quit]? -1
3
Position [q to quit]? two
Something else broke: invalid literal for int() with base 10: 'two'
Position [q to quit]? q
```

Inputting position 3 raised an `IndexError` as expected. Entering two annoyed the `int()` function, which we handled in our second, catchall `except` code.

Make Your Own Exceptions

The previous section discussed handling exceptions, but all of the exceptions (such as `IndexError`) were predefined in Python or its standard library. You can use any of these for your own purposes. You can also define your own exception types to handle special situations that might arise in your own programs.

This requires defining a new object type with a *class*—something we don't get into until Chapter 6. So, if you're unfamiliar with classes, you might want to return to this section later.

An exception is a class. It is a child of the class `Exception`. Let's make an exception called `UppercaseException` and raise it when we encounter an uppercase word in a string.

```
>>> class UppercaseException(Exception):
...     pass
...
```

```
>>> words = ['eeenie', 'meenie', 'miny', 'MO']
>>> for word in words:
...     if word.isupper():
...         raise UppercaseException(word)
...
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
__main__.UppercaseException: MO
```

We didn't even define any behavior for UppercaseException (notice we just used pass), letting its parent class Exception figure out what to print when the exception was raised.

You can access the exception object itself and print it:

```
>>> try:
...     raise OopsException('panic')
... except OopsException as exc:
...     print(exc)
...
panic
```

Things to Do

4.1 Assign the value 7 to the variable guess_me. Then, write the conditional tests (if, else, and elif) to print the string 'too low' if guess_me is less than 7, 'too high' if greater than 7, and 'just right' if equal to 7.

4.2 Assign the value 7 to the variable guess_me and the value 1 to the variable start. Write a while loop that compares start with guess_me. Print too low if start is less than guess_me. If start equals guess_me, print 'found it!' and exit the loop. If start is greater than guess_me, print 'oops' and exit the loop. Increment start at the end of the loop.

4.3 Use a for loop to print the values of the list [3, 2, 1, 0].

4.4 Use a list comprehension to make a list of the even numbers in range(10).

4.5 Use a dictionary comprehension to create the dictionary squares. Use range(10) to return the keys, and use the square of each key as its value.

4.6 Use a set comprehension to create the set odd from the odd numbers in range(10).

4.7 Use a generator comprehension to return the string 'Got ' and a number for the numbers in range(10). Iterate through this by using a for loop.

4.8 Define a function called good that returns the list ['Harry', 'Ron', 'Hermione'].

4.9 Define a generator function called get_odds that returns the odd numbers from range(10). Use a for loop to find and print the third value returned.

4.10 Define a decorator called `test` that prints `'start'` when a function is called and `'end'` when it finishes.

4.11 Define an exception called `OopsException`. Raise this exception to see what happens. Then write the code to catch this exception and print `'Caught an oops'`.

4.12 Use `zip()` to make a dictionary called `movies` that pairs these lists: `titles = ['Creature of Habit', 'Crewel Fate']` and `plots = ['A nun turns into a monster', 'A haunted yarn shop']`.

[illegible]

II [illegible] unit

CHAPTER 5

Py Boxes: Modules, Packages, and Programs

During your bottom-up climb, you've progressed from built-in data types to constructing ever-larger data and code structures. In this chapter, you'll finally get down to brass tacks and learn how to write realistic, large programs in Python.

Standalone Programs

Thus far, you've been writing and running code fragments such as the following within Python's interactive interpreter:

```
>>> print("This interactive snippet works.")
This interactive snippet works.
```

Now let's make your first standalone program. On your computer, create a file called *test1.py* containing this single line of Python code:

```
print("This standalone program works!")
```

Notice that there's no >>> prompt, just a single line of Python code. Ensure that there is no indentation in the line before `print`.

If you're running Python in a text terminal or terminal window, type the name of your Python program followed by the program filename:

```
$ python test1.py
This standalone program works!
```

You can save all of the interactive snippets that you've seen in this book so far to files and run them directly. If you're cutting and pasting, ensure that you delete the initial >>> and ... (include the final space).

Command-Line Arguments

On your computer, create a file called *test2.py* that contains these two lines:

```
import sys
print('Program arguments:', sys.argv)
```

Now, use your version of Python to run this program. Here's how it might look in a Linux or Mac OS X terminal window using a standard shell program:

```
$ python test2.py
Program arguments: ['test2.py']
$ python test2.py tra la la
Program arguments: ['test2.py', 'tra', 'la', 'la']
```

Modules and the import Statement

We're going to step up another level, creating and using Python code in more than one file. A *module* is just a file of Python code.

The text of this book is organized in a hierarchy: words, sentences, paragraphs, and chapters. Otherwise, it would be unreadable after a page or two. Code has a roughly similar bottom-up organization: data types are like words, statements are like sentences, functions are like paragraphs, and modules are like chapters. To continue the analogy, in this book, when I say that something will be explained in Chapter 8, in programming, that's like referring to code in another module.

We refer to code of other modules by using the `import` statement. This makes the code and variables in the imported module available to your program.

Import a Module

The simplest use of the `import` statement is `import` *module*, where *module* is the name of another Python file, without the *.py* extension. Let's simulate a weather station and print a weather report. One main program prints the report, and a separate module with a single function returns the weather description used by the report.

Here's the main program (call it *weatherman.py*):

```
import report

description = report.get_description()
print("Today's weather:", description)
```

And here is the module (*report.py*):

```
def get_description():  # see the docstring below?
    """Return random weather, just like the pros"""
    from random import choice
    possibilities = ['rain', 'snow', 'sleet', 'fog', 'sun', 'who knows']
    return choice(possibilities)
```

If you have these two files in the same directory and instruct Python to run *weatherman.py* as the main program, it will access the report module and run its get_description() function. We wrote this version of get_description() to return a random result from a list of strings, so that's what the main program will get back and print:

```
$ python weatherman.py
Today's weather: who knows
$ python weatherman.py
Today's weather: sun
$ python weatherman.py
Today's weather: sleet
```

We used imports in two different places:

- The main program *weatherman.py* imported the module report.
- In the module file *report.py*, the get_description() function imported the choice function from Python's standard random module.

We also used imports in two different ways:

- The main program called import report and then ran report.get_description().
- The get_description() function in *report.py* called from random import choice and then ran choice(possibilities).

In the first case, we imported the entire report module but needed to use report. as a prefix to get_description(). After this import statement, everything in *report.py* is available to the main program, as long as we tack report. before its name. By *qualifying* the contents of a module with the module's name, we avoid any nasty naming conflicts. There could be a get_description() function in some other module, and we would not call it by mistake.

In the second case, we're within a function and know that nothing else named choice is here, so we imported the choice() function from the random module directly. We could have written the function like the following snippet, which returns random results:

```
def get_description():
    import random
    possibilities = ['rain', 'snow', 'sleet', 'fog', 'sun', 'who knows']
    return random.choice(possibilities)
```

Like many aspects of programming, pick the style that seems the most clear to you. The module-qualified name (random.choice) is safer but requires a little more typing.

These get_description() examples showed variations of *what* to import, but but not *where* to do the importing—they all called import from inside the function. We could have imported random from outside the function:

```
>>> import random
>>> def get_description():
...     possibilities = ['rain', 'snow', 'sleet', 'fog', 'sun', 'who knows']
...     return random.choice(possibilities)
...
>>> get_description()
'who knows'
>>> get_description()
'rain'
```

You should consider importing from outside the function if the imported code might be used in more than one place, and from inside if you know its use will be limited. Some people prefer to put all their imports at the top of the file, just to make all the dependencies of their code explicit. Either way works.

Import a Module with Another Name

In our main *weatherman.py* program, we called import report. But what if you have another module with the same name or want to use a name that is more mnemonic or shorter? In such a situation, you can import using an *alias*. Let's use the alias wr:

```
import report as wr
description = wr.get_description()
print("Today's weather:", description)
```

Import Only What You Want from a Module

With Python, you can import one or more parts of a module. Each part can keep its original name or you can give it an alias. First, let's import get_description() from the report module with its original name:

```
from report import get_description
description = get_description()
print("Today's weather:", description)
```

Now, import it as do_it:

```
from report import get_description as do_it
description = do_it()
print("Today's weather:", description)
```

Module Search Path

Where does Python look for files to import? It uses a list of directory names and ZIP archive files stored in the standard sys module as the variable path. You can access and modify this list. Here's the value of sys.path for Python 3.3 on my Mac:

```
>>> import [illegible]
>>> for place in sys.path:
...     print(place)
...

/Library/Frameworks/Python.framework/Versions/3.3/lib/python33.zip
/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3
/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/plat-darwin
/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/lib-dynload
/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/site-packages
```

That initial blank output line is the empty string `''`, which stands for the current directory. If `''` is first in `sys.path`, Python looks in the current directory first when you try to import something: `import report` looks for `report.py`.

The first match will be used. This means that if you define a module named `random` and it's in the search path before the standard library, you won't be able to access the standard library's `random` now.

Packages

We went from single lines of code, to multiline functions, to standalone programs, to multiple modules in the same directory. To allow Python applications to scale even more, you can organize modules into file hierarchies called *packages*.

Maybe we want different types of text forecasts: one for the next day and one for the next week. One way to structure this is to make a directory named `sources`, and create two modules within it: *daily.py* and *weekly.py*. Each has a function called `forecast`. The daily version returns a string, and the weekly version returns a list of seven strings.

Here's the main program and the two modules. (The `enumerate()` function takes apart a list and feeds each item of the list to the `for` loop, adding a number to each item as a little bonus.)

Main program: *boxes/weather.py*

```
from [illegible] import daily, weekly

print("Daily forecast:", daily.forecast())
print("Weekly forecast:")
for number, outlook in enumerate(weekly.forecast(), [illegible]):
    print(number, outlook)
```

Module 1: *boxes/sources/daily.py*.

```
def forecast():
    'fake daily forecast'
    return 'like yesterday'
```

Module 2: *boxes/sources/weekly.py*.

```
def forecast():
    """Fake weekly forecast"""
    return ['snow', 'more snow', 'sleet',
        'freezing rain', 'rain', 'fog', 'hail']
```

You'll need one more thing in the sources directory: a file named *__init__.py*. This can be empty, but Python needs it to treat the directory containing it as a package.

Run the main *weather.py* program to see what happens:

```
$ python weather.py
Daily forecast: like yesterday
Weekly forecast:
1 snow
2 more snow
3 sleet
4 freezing rain
5 rain
6 fog
7 hail
```

The Python Standard Library

One of Python's prominent claims is that it has "batteries included"—a large standard library of modules that perform many useful tasks, and are kept separate to avoid bloating the core language. When you're about to write some Python code, it's often worthwhile to first check whether there's a standard module that already does what you want. It's surprising how often you encounter little gems in the standard library. Python also provides authoritative documentation for the modules, along with a tutorial. Doug Hellmann's website *Python Module of the Week* and his book *The Python Standard Library by Example* (Addison-Wesley Professional) are also very useful guides.

Upcoming chapters in this book feature many of the standard modules that are specific to the Web, systems, databases, and so on. In this section, I'll talk about some standard modules that have generic uses.

Handle Missing Keys with setdefault() and defaultdict()

You've seen that trying to access a dictionary with a nonexistent key raises an exception. Using the dictionary `get()` function to return a default value avoids an exception.

The setdefault() function is like get(), but also assigns an item to the dictionary if the key is missing:

```
>>> periodic_table = {'Hydrogen': [illegible], 'Helium': [illegible]}
>>> print(periodic_table)
{'Helium': [illegible], 'Hydrogen': [illegible]}
```

If the key was *not* already in the dictionary, the new value is used:

```
>>> carbon = periodic_table.setdefault('Carbon', [illegible])
>>> carbon
12
>>> periodic_table
{'Helium': [illegible], 'Carbon': [illegible], 'Hydrogen': [illegible]}
```

If we try to assign a different default value to an *existing* key, the original value is returned and nothing is changed:

```
>>> helium = periodic_table.setdefault('Helium', 9[illegible]7)
>>> helium
2
>>> periodic_table
{'Helium': [illegible], 'Carbon': [illegible], 'Hydrogen': [illegible]}
```

defaultdict() is similar, but specifies the default value for any new key up front, when the dictionary is created. Its argument is a function. In this example, we pass the function int, which will be called as int() and return the integer 0:

```
>>> from [illegible] import defaultdict
>>> periodic_table = defaultdict(int)
```

Now, any missing value will be an integer (int), with the value 0:

```
>>> periodic_table['Hydrogen'] = [illegible]
>>> periodic_table['Lead']
[illegible]
>>> periodic_table
defaultdict(<class 'int'>, {'Lead': 0, 'Hydrogen': 1})
```

The argument to defaultdict() is a function that returns the value to be assigned to a missing key. In the following example, no_idea() is executed to return a value when needed:

```
>>> from [illegible] import defaultdict
>>>
>>> def no_idea():
...     return 'Huh?'
...
>>> bestiary = defaultdict(no_idea)
>>> bestiary['A'] = 'Abominable Snowman'
>>> bestiary['B'] = 'Basilisk'
>>> bestiary['A']
'Abominable Snowman'
>>> bestiary['B']
```

```
'Basilisk'
>>> bestiary['C']
'Huh?'
```

You can use the functions `int()`, `list()`, or `dict()` to return default empty values for those types: `int()` returns `0`, `list()` returns an empty list (`[]`), and `dict()` returns an empty dictionary (`{}`). If you omit the argument, the initial value of a new key will be set to `None`.

By the way, you can use `lambda` to define your default-making function right inside the call:

```
>>> bestiary = defaultdict(lambda: 'Huh?')
>>> bestiary['E']
'Huh?'
```

Using `int` is one way to make your own counter:

```
>>> from collections import defaultdict
>>> food_counter = defaultdict(int)
>>> for food in ['spam', 'spam', 'eggs', 'spam']:
...     food_counter[food] += 1
...
>>> for food, count in food_counter.items():
...     print(food, count)
...
eggs 1
spam 3
```

In the preceding example, if `food_counter` had been a normal dictionary instead of a `defaultdict`, Python would have raised an exception every time we tried to increment the dictionary element `food_counter[food]` because it would not have been initialized. We would have needed to do some extra work, as shown here:

```
>>> dict_counter = {}
>>> for food in ['spam', 'spam', 'eggs', 'spam']:
...     if not food in dict_counter:
...         dict_counter[food] = 0
...     dict_counter[food] += 1
...
>>> for food, count in dict_counter.items():
...     print(food, count)
...
spam 3
eggs 1
```

Count Items with Counter()

Speaking of counters, the standard library has one that does the work of the previous example and more:

```
>>> from [illegible] import Counter
>>> breakfast = ['spam', 'spam', 'eggs', 'spam']
>>> breakfast_counter = Counter(breakfast)
>>> breakfast_counter
Counter({'spam': 3, 'eggs': 1})
```

The most_common() function returns all elements in descending order, or just the top count elements if given a count:

```
>>> breakfast_counter.most_common()
[('spam', 3), ('eggs', 1)]
>>> breakfast_counter.most_common(1)
[('spam', 3)]
```

You can combine counters. First, let's see again what's in breakfast_counter:

```
>>> breakfast_counter
>>> Counter({'spam': 3, 'eggs ': 1})
```

This time, we'll make a new list called lunch, and a counter called lunch_counter:

```
>>> lunch = ['eggs', 'eggs',  'bacon']
>>> lunch_counter = Counter(lunch)
>>> lunch_counter
Counter({'eggs': 2, 'bacon': 1})
```

The first way we combine the two counters is by addition, using +:

```
>>> breakfast_counter + lunch__counter
Counter({'spam': 3, 'eggs': [illegible], 'bacon': 1})
```

As you might expect, you subtract one counter from another by using -. What's for breakfast but not for lunch?

```
>>> breakfast_counter - lunch__counter
Counter({'spam': 3})
```

Okay, now what can we have for lunch that we can't have for breakfast?

```
>>> lunch_counter - breakfast__counter
Counter({'bacon': 1, 'eggs': 1})
```

Similar to sets in Chapter 4, you can get common items by using the intersection operator &:

```
>>> breakfast_counter & lunch__counter
Counter({'eggs': 1})
```

The intersection picked the common element ('eggs') with the lower count. This makes sense: breakfast only offered one egg, so that's the common count.

Finally, you can get all items by using the union operator |:

```
>>> breakfast_counter | lunch__counter
Counter({'spam': [illegible], 'eggs': 2, 'bacon': [illegible]})
```

The item 'eggs' was again common to both. Unlike addition, union didn't add their counts, but picked the one with the larger count.

Order by Key with OrderedDict()

Many of the code examples in the early chapters of this book demonstrate that the order of keys in a dictionary is not predictable: you might add keys a, b, and c in that order, but keys() might return c, a, b. Here's a repurposed example from Chapter 1:

```
>>> quotes = {
...     'Moe': 'A wise guy, huh?',
...     'Larry': 'Ow!',
...     'Curly': 'Nyuk nyuk!',
...     }
>>> for stooge in quotes:
...  print(stooge)
...
Larry
Curly
Moe
```

An OrderedDict() remembers the order of key addition and returns them in the same order from an iterator. Try creating an OrderedDict from a sequence of (*key*, *value*) tuples:

```
>>> from collections import OrderedDict
>>> quotes = OrderedDict([
...     ('Moe', 'A wise guy, huh?'),
...     ('Larry', 'Ow!'),
...     ('Curly', 'Nyuk nyuk!'),
...     ])
>>>
>>> for stooge in quotes:
...     print(stooge)
...
Moe
Larry
Curly
```

Stack + Queue == deque

A deque (pronounced *deck*) is a double-ended queue, which has features of both a stack and a queue. It's useful when you want to add and delete items from either end of a sequence. Here, we'll work from both ends of a word to the middle to see if it's a palindrome. The function popleft() removes the leftmost item from the deque and returns it; pop() removes the rightmost item and returns it. Together, they work from the ends toward the middle. As long as the end characters match, it keeps popping until it reaches the middle:

```
>>> def palindrome(word):
...     from collections import deque
...     dq = deque(word)
...     while len(dq) > 1:
...        if dq.popleft() != dq.pop():
...            return False
...     return True
...
...
>>> palindrome('a')
True
>>> palindrome('racecar')
True
>>> palindrome('')
True
>>> palindrome('radar')
True
>>> palindrome('halibut')
False
```

I used this as a simple illustration of deques. If you really wanted a quick palindrome checker, it would be a lot simpler to just compare a string with its reverse. Python doesn't have a reverse() function for strings, but it does have a way to reverse a string with a slice, as illustrated here:

```
>>> def another_palindrome(word):
...     return word == word[::-1]
...
>>> another_palindrome('radar')
True
>>> another_palindrome('halibut')
False
```

Iterate over Code Structures with itertools

itertools contains special-purpose iterator functions. Each returns one item at a time when called within a for … in loop, and remembers its state between calls.

chain() runs through its arguments as though they were a single iterable:

```
>>> import itertools
>>> for item in itertools.chain([1, 2], ['a', 'b']):
...     print(item)
...
1
2
a
b
```

cycle() is an infinite iterator, cycling through its arguments:

```
>>> import itertools
>>> for item in itertools.cycle([1, 2]):
...     print(item)
...
1
2
1
2
.
.
.
```

...and so on.

accumulate() calculates accumulated values. By default, it calculates the sum:

```
>>> import itertools
>>> for item in itertools.accumulate([1, 2, 3, 4]):
...     print(item)
...
1
3
6
10
```

You can provide a function as the second argument to accumulate(), and it will be used instead of addition. The function should take two arguments and return a single result. This example calculates an accumulated product:

```
>>> import itertools
>>> def multiply(a, b):
...     return a * b
...
>>> for item in itertools.accumulate([1, 2, 3, 4], multiply):
...     print(item)
...
1
2
6
24
```

The itertools module has many more functions, notably some for combinations and permutations that can be time savers when the need arises.

Print Nicely with pprint()

All of our examples have used print() (or just the variable name, in the interactive interpreter) to print things. Sometimes, the results are hard to read. We need a *pretty printer* such as pprint():

```
>>> from pprint import pprint
>>> quotes = OrderedDict([
...     ('Moe', 'A wise guy, huh?'),
...     ('Larry', 'Ow!'),
...     ('Curly', 'Nyuk nyuk!'),
...     ])
>>>
```

Plain old `print()` just dumps things out there:

```
>>> print(quotes)
OrderedDict([('Moe', 'A wise guy, huh?'), ('Larry', 'Ow!'), ('Curly', 'Nyuk nyuk!')])
```

However, `pprint()` tries to align elements for better readability:

```
>>> pprint(quotes)
{'Moe': 'A wise guy, huh?',
 'Larry': 'Ow!',
 'Curly': 'Nyuk nyuk!'}
```

More Batteries: Get Other Python Code

Sometimes, the standard library doesn't have what you need, or doesn't do it in quite the right way. There's an entire world of open-source, third-party Python software. Good resources include:

- PyPi (also known as the Cheese Shop, after an old Monty Python skit)
- github
- readthedocs

You can find many smaller code examples at activestate.

Almost all of the Python code in this book uses the standard Python installation on your computer, which includes all the built-ins and the standard library. External packages are featured in some places: I mentioned `requests` in Chapter 1, and have more details in "Beyond the Standard Library: Requests" on page 222. Appendix D shows how to install third-party Python software, along with many other nuts-and-bolts development details.

Things to Do

5.1. Create a file called *zoo.py*. In it, define a function called `hours()` that prints the string `'Open 9-5 daily'`. Then, use the interactive interpreter to import the `zoo` module and call its `hours()` function.

5.2. In the interactive interpreter, import the `zoo` module as `menagerie` and call its `hours()` function.

CHAPTER 6
Oh Oh: Objects and Classes

No object is mysterious. The mystery is your eye.

— Elizabeth Bowen

Take an object. Do something to it. Do something else to it.

— Jasper Johns

Up to this point, you've seen data structures such as strings and dictionaries, and code structures such as functions and modules. In this chapter, you'll deal with custom data structures: *objects*.

What Are Objects?

As I mention in Chapter 2, everything in Python, from numbers to modules, is an object. However, Python hides most of the object machinery by means of special syntax. You can type `num = 7` to create a object of type integer with the value 7, and assign an object reference to the name `num`. The only time you need to look inside objects is when you want to make your own or modify the behavior of existing objects. You'll see how to do both in this chapter.

An object contains both data (variables, called *attributes*) and code (functions, called *methods*). It represents a unique instance of some concrete thing. For example, the integer object with the value 7 is an object that facilitates methods such as addition and multiplication, as is demonstrated in "Numbers" on page 19. 8 is a different object. This means there's an Integer class in Python, to which both 7 and 8 belong. The strings `'cat'` and `'duck'` are also objects in Python, and have string methods that you've seen, such as `capitalize()` and `replace()`.

When you create new objects no one has ever created before, you must create a class that indicates what they contain.

Think of objects as nouns and their methods as verbs. An object represents an individual thing, and its methods define how it interacts with other things.

Unlike modules, you can have multiple objects at the same time, each one with different values for its attributes. They're like super data structures, with code thrown in.

Define a Class with class

In Chapter 1, I compare an object to a plastic box. A *class* is like the mold that makes that box. For instance, a `String` is the built-in Python class that makes string objects such as `'cat'` and `'duck'`. Python has many other built-in classes to create the other standard data types, including lists, dictionaries, and so on. To create your own custom object in Python, you first need to define a class by using the `class` keyword. Let's walk through a simple example.

Suppose that you want to define objects to represent information about people. Each object will represent one person. You'll first want to define a class called `Person` as the mold. In the examples that follow, we'll try more than one version of this class as we build up from the simplest class to ones that actually do something useful.

Our first try is the simplest possible class, an empty one:

```
>>> class Person():
...     pass
```

Just as with functions, we needed to say `pass` to indicate that this class was empty. This definition is the bare minimum to create an object. You create an object from a class by calling the class name as though it were a function:

```
>>> someone = Person()
```

In this case, `Person()` creates an individual object from the `Person` class and assigns it the name `someone`. But, our `Person` class was empty, so the `someone` object that we create from it just sits there and can't do anything else. You would never actually define such a class, and I'm only showing it here to build up to the next example.

Let's try again, this time including the special Python object initialization method `__init__`:

```
>>> class Person():
...     def __init__(self):
...         pass
```

This is what you'll see in real Python class definitions. I admit that the `__init__()` and `self` look strange. `__init__()` is the special Python name for a method that initializes an individual object from its class definition.[1] The `self` argument specifies that it refers to the individual object itself.

When you define `__init__()` in a class definition, its first parameter should be `self`. Although `self` is not a reserved word in Python, it's common usage. No one reading your code later (including you!) will need to guess what you meant if you use `self`.

But even that second `Person` definition didn't create an object that really did anything. The third try is the charm that really shows how to create a simple object in Python. This time, we'll add the parameter `name` to the initialization method:

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

Now, we can create an object from the `Person` class by passing a string for the `name` parameter:

```
>>> hunter = Person('Elmer Fudd')
```

Here's what this line of code does:

- Looks up the definition of the `Person` class
- *Instantiates* (creates) a new object in memory
- Calls the object's `__init__` method, passing this newly-created object as `self` and the other argument (`'Elmer Fudd'`) as `name`
- Stores the value of `name` in the object
- Returns the new object
- Attaches the name `hunter` to the object

This new object is like any other object in Python. You can use it as an element of a list, tuple, dictionary, or set. You can pass it to a function as an argument, or return it as a result.

What about the `name` value that we passed in? It was saved with the object as an attribute. You can read and write it directly:

```
>>> print('The mighty hunter: ', hunter.name)
The mighty hunter: Elmer Fudd
```

1. You'll see many examples of double underscores in Python names; to save syllables, some people pronounce them as *dunder*.

Remember, *inside* the `Person` class definition, you access the `name` attribute as `self.name`. When you create an actual object such as `hunter`, you refer to it as `hunter.name`.

It is *not* necessary to have an `__init__` method in every class definition; it's used to do anything that's needed to distinguish this object from others created from the same class.

Inheritance

When you're trying to solve some coding problem, often you'll find an existing class that creates objects that do almost what you need. What can you do? You could modify this old class, but you'll make it more complicated, and you might break something that used to work.

Of course, you could write a new class, cutting and pasting from the old one and merging your new code. But this means that you have more code to maintain, and the parts of the old and new classes that used to work the same might drift apart because they're now in separate places.

The solution is *inheritance*: creating a new class from an existing class but with some additions or changes. It's an excellent way to reuse code. When you use inheritance, the new class can automatically use all the code from the old class but without copying any of it.

You define only what you need to add or change in the new class, and this overrides the behavior of the old class. The original class is called a *parent*, *superclass*, or *base class*; the new class is called a *child*, *subclass*, or *derived class*. These terms are interchangeable in object oriented programming.

So, let's inherit something. We'll define an empty class called `Car`. Next, define a subclass of `Car` called `Yugo`. You define a subclass by using the same `class` keyword but with the parent class name inside the parentheses (`class Yugo(Car)` below):

```
>>> class Car():
...     pass
...
>>> class Yugo(Car):
...     pass
...
```

Next, create an object from each class:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

A child class is a specialization of a parent class; in object-oriented lingo, `Yugo` *is-a* `Car`. The object named `give_me_a_yugo` is an instance of class `Yugo`, but it also inherits whatever a Car can do. In this case, Car and Yugo are as useful as deckhands on a submarine, so let's try new class definitions that actually do something:

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     pass
...
```

Finally, make one object from each class and call the exclaim method:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
>>> give_me_a_car.exclaim()
I'm a Car!
>>> give_me_a_yugo.exclaim()
I'm a Car!
```

Without doing anything special, Yugo inherited the exclaim() method from Car. In fact, Yugo says that it *is* a Car, which might lead to an identity crisis. Let's see what we can do about that.

Override a Method

SM 2018q.

As you just saw, a new class initially inherits everything from its parent class. Moving forward, you'll see how to replace or *override* a parent method. Yugo should probably be different from Car in some way; otherwise, what's the point of defining a new class? Let's change how the exclaim() method works for a Yugo:

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     def exclaim(self):
...         print("I'm a Yugo! Much like a Car, but more Yugo-ish.")
...
```

Now, make two objects from these classes:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

What do they say?

```
>>> give_me_a_car.exclaim()
I'm a Car!
>>> give_me_a_yugo.exclaim()
I'm a Yugo! Much like a Car, but more Yugo-ish.
```

In these examples, we overrode the exclaim() method. We can override any methods, including __init__(). Here's another example that uses our earlier Person class. Let's make subclasses that represent doctors (MDPerson) and lawyers (JDPerson):

```
>>> class Person():
...     def __init__(self, name):
...         self.name = name
...
>>> class MDPerson(Person):
...     def __init__(self, name):
...         self.name = "Doctor " + name
...
>>> class JDPerson(Person):
...     def __init__(self, name):
...         self.name = name + ", Esquire"
...
```

In these cases, the initialization method __init__() takes the same arguments as the parent Person class but stores the value of name differently inside the object instance:

```
>>> person = Person('Fudd')
>>> doctor = MDPerson('Fudd')
>>> lawyer = JDPerson('Fudd')
>>> print(person.name)
Fudd
>>> print(doctor.name)
Doctor Fudd
>>> print(lawyer.name)
Fudd, Esquire
```

Add a Method

The child class can also *add* a method that was not present in its parent class. Going back to classes Car and Yugo, we'll define the new method need_a_push() for class Yugo only:

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     def exclaim(self):
...         print("I'm a Yugo! Much like a Car, but more Yugo-ish.")
...     def need_a_push(self):
...         print("A little help here?")
...
```

Next, make a Car and a Yugo:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

A Yugo object can react to a need_a_push() method call:

```
>>> give_me_a_yugo.need_a_push()
A little help here?
```

But a generic Car object cannot:

```
>>> give_me_a_car.need_a_push()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Car' object has no attribute 'need_a_push'
```

At this point, a Yugo can do something that a Car cannot, and the distinct personality of a Yugo can emerge.

Get Help from Your Parent with super

We saw how the child class could add or override a method from the parent. What if it wanted to call that parent method? "I'm glad you asked," says super(). We'll define a new class called EmailPerson that represents a Person with an email address. First, our familiar Person definition:

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

Notice that the __init__() call in the following subclass has an additional email parameter:

```
>>> class EmailPerson(Person):
...     def __init__(self, name, email):
...         super().__init__(name)
...         self.email = email
```

When you define an __init__() method for your class, you're replacing the __init__() method of its parent class, and the latter is not called automatically anymore. As a result, we need to call it explicitly. Here's what's happening:

- The super() gets the definition of the parent class, Person.
- The __init__() method calls the Person.__init__() method. It takes care of passing the self argument to the superclass, so you just need to give it any optional arguments. In our case, the only other argument Person() accepts is name.
- The self.email = email line is the new code that makes this EmailPerson different from a Person.

Moving on, let's make one of these creatures:

```
>>> bob = EmailPerson('Bob Frapples', 'bob@frapples.com')
```

We should be able to access both the name and email attributes:

```
>>> bob.name
'Bob Frapples'
>>> bob.email
'bob@frapples.com'
```

Why didn't we just define our new class as follows?

```
>>> class EmailPerson(Person):
...     def __init__(self, name, email):
...         self.name = name
...         self.email = email
```

We could have done that, but it would have defeated our use of inheritance. We used super() to make Person do its work, the same as a plain Person object would. There's another benefit: if the definition of Person changes in the future, using super() will ensure that the attributes and methods that EmailPerson inherits from Person will reflect the change.

Use super() when the child is doing something its own way but still needs something from the parent (as in real life).

In self Defense

One criticism of Python (besides the use of whitespace) is the need to include self as the first argument to instance methods (the kind of method you've seen in the previous examples). Python uses the self argument to find the right object's attributes and methods. For an example, I'll show how you would call an object's method, and what Python actually does behind the scenes.

Remember class Car from earlier examples? Let's call its exclaim() method again:

```
>>> car = Car()
>>> car.exclaim()
I'm a Car!
```

Here's what Python actually does, under the hood:

- Look up the class (Car) of the object car.
- Pass the object car to the exclaim() method of the Car class as the self parameter.

Just for fun, you can even run it this way yourself and it will work the same as the normal (car.exclaim()) syntax:

```
>>> Car.exclaim(car)
I'm a Car!
```

However, there's never a reason to use that lengthier style.

Get and Set Attribute Values with Properties

Some object-oriented languages support private object attributes that can't be accessed directly from the outside; programmers often need to write *getter* and *setter* methods to read and write the values of such private attributes.

Python doesn't need getters or setters, because all attributes and methods are public, and you're expected to behave yourself. If direct access to attributes makes you nervous, you can certainly write getters and setters. But be Pythonic—use *properties*.

In this example, we'll define a Duck class with a single attribute called hidden_name. (In the next section, I'll show you a better way to name attributes that you want to keep private.) We don't want people to access this directly, so we'll define two methods: a getter (get_name()) and a setter (set_name()). I've added a print() statement to each method to show when it's being called. Finally, we define these methods as properties of the name attribute:

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.hidden_name = input_name
...     def get_name(self):
...         print('inside the getter')
...         return self.hidden_name
...     def set_name(self, input_name):
...         print('inside the setter')
...         self.hidden_name = input_name
...     name = property(get_name, set_name)
```

The new methods act as normal getters and setters until that last line; it defines the two methods as properties of the attribute called name. The first argument to property() is the getter method, and the second is the setter. Now, when you refer to the name of any Duck object, it actually calls the get_name() method to return it:

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
```

You can still call get_name() directly, too, like a normal getter method:

```
>>> fowl.get_name()
inside the getter
'Howard'
```

When you assign a value to the name attribute, the set_name() method will be called:

```
>>> fowl.name = 'Daffy'
inside the setter
>>> fowl.name
inside the getter
'Daffy'
```

You can still call the `set_name()` method directly:

```
>>> fowl.set_name('Daffy')
inside the setter
>>> fowl.name
inside the getter
'Daffy'
```

Another way to define properties is with *decorators*. In this next example, we'll define two different methods, each called `name()` but preceded by different decorators:

- `@property`, which goes before the getter method
- `@name.setter`, which goes before the setter method

Here's how they actually look in the code:

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.hidden_name = input_name
...     @property
...     def name(self):
...         print('inside the getter')
...         return self.hidden_name
...     @name.setter
...     def name(self, input_name):
...         print('inside the setter')
...         self.hidden_name = input_name
```

You can still access `name` as though it were an attribute, but there are no visible `get_name()` or `set_name()` methods:

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
>>> fowl.name = 'Donald'
inside the setter
>>> fowl.name
inside the getter
'Donald'
```

If anyone guessed that we called our attribute `hidden_name`, they could still read and write it directly as `fowl.hidden_name`. In the next section, you'll see how Python provides a special way to name private attributes.

In both of the previous examples, we used the `name` property to refer to a single attribute (ours was called `hidden_name`) stored within the object. A property can refer to a *com-*

puted value, as well. Let's define a Circle class that has a radius attribute and a computed diameter property:

```
>>> class Circle():
...     def __init__(self, radius):
...         self.radius = radius
...
...     def diameter(self):
...         return 2 * self.radius
...
```

We create a Circle object with an initial value for its radius:

```
>>> c = Circle(5)
>>> c.radius
5
```

We can refer to diameter as if it were an attribute such as radius:

```
>>> c.diameter
10
```

Here's the fun part: we can change the radius attribute at any time, and the diameter property will be computed from the current value of radius:

```
>>> c.radius = 7
>>> c.diameter
[illegible]
```

If you don't specify a setter property for an attribute, you can't set it from the outside. This is handy for read-only attributes:

```
>>> c.diameter = 20
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

There's one more big advantage of using a property over direct attribute access: if you ever change the definition of the attribute, you only need to fix the code within the class definition, not in all the callers.

Name Mangling for Privacy

In the Duck class example in the previous section, we called our (not completely) hidden attribute hidden_name. Python has a naming convention for attributes that should not be visible outside of their class definition: begin by using with two underscores (__).

Let's rename hidden_name to __name, as demonstrated here:

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.__name = input_name
...     def name(self):
...         print('inside the getter')
...         return self.__name
...
...     def name(self, input_name):
...         print('inside the setter')
...         self.__name = input_name
...
...
```

Take a moment to see if everything still works:

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
>>> fowl.name = 'Donald'
inside the setter
>>> fowl.name
inside the getter
'Donald'
```

Looks good. And, you can't access the `__name` attribute:

```
>>> fowl.__name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Duck' object has no attribute '__name'
```

This naming convention doesn't make it private, but Python does *mangle* the name to make it unlikely for external code to stumble upon it. If you're curious and promise not to tell everyone, here's what it becomes:

```
>>> fowl._Duck__name
'Donald'
```

Notice that it didn't print `inside the getter`. Although this isn't perfect protection, name mangling discourages accidental or intentional direct access to the attribute.

Method Types

Some data (*attributes*) and functions (*methods*) are part of the class itself, and some are part of the objects that are created from that class.

When you see an initial `self` argument in methods within a class definition, it's an *instance method*. These are the types of methods that you would normally write when creating your own classes. The first parameter of an instance method is `self`, and Python passes the object to the method when you call it.

In contrast, a *class method* affects the class as a whole. Any change you make to the class affects all of its objects. Within a class definition, a preceding `@classmethod` decorator indicates that that following function is a class method. Also, the first parameter to the method is the class itself. The Python tradition is to call the parameter `cls`, because `class` is a reserved word and can't be used here. Let's define a class method for `A` that counts how many object instances have been made from it:

```
>>> class A():
...     count = [illegible]
...     def __init__(self):
...         A.count += 1
...     def exclaim(self):
...         print("I'm an A !")
...
...     def kids(cls):
...         print("A has", cls.count, "little objects.")
...
>>>
>>> easy_a = A()
>>> breezy_a = A()
>>> wheezy_a = A()
>>> A.kids()
A has [illegible] little objects.
```

Notice that we referred to `A.count` (the class attribute) rather than `self.count` (which would be an object instance attribute). In the `kids()` method, we used `cls.count`, but we could just as well have used `A.count`.

A third type of method in a class definition affects neither the class nor its objects; it's just in there for convenience instead of floating around on its own. It's a *static method*, preceded by a `@staticmethod` decorator, with no initial `self` or `class` parameter. Here's an example that serves as a commercial for the class `CoyoteWeapon`:

```
>>> class CoyoteWeapon():
...
...     def commercial():
...         print('This CoyoteWeapon has been brought to you by Acme')
...
>>>
>>> CoyoteWeapon.commercial()
This CoyoteWeapon has been brought to you by Acme
```

Notice that we didn't need to create an object from class `CoyoteWeapon` to access this method. Very class-y.

Duck Typing

Python has a loose implementation of *polymorphism*; this means that it applies the same operation to different objects, regardless of their class.

Let's use the same `__init__()` initializer for all three `Quote` classes now, but add two new functions:

- `who()` just returns the value of the saved `person` string
- `says()` returns the saved `words` string with the specific punctuation

And here they are in action:

```
>>> class Quote():
...     def __init__(self, person, words):
...         self.person = person
...         self.words = words
...     def who(self):
...         return self.person
...     def says(self):
...         return self.words + '.'
...
>>> class QuestionQuote(Quote):
...     def says(self):
...         return self.words + '?'
...
>>> class ExclamationQuote(Quote):
...     def says(self):
...         return self.words + '!'
...
>>>
```

We didn't change how `QuestionQuote` or `ExclamationQuote` were initialized, so we didn't override their `__init__()` methods. Python then automatically calls the `__init__()` method of the parent class `Quote` to store the instance variables `person` and `words`. That's why we can access `self.words` in objects created from the subclasses `QuestionQuote` and `ExclamationQuote`.

Next up, let's make some objects:

```
>>> hunter = Quote('Elmer Fudd', "I'm hunting wabbits")
>>> print(hunter.who(), 'says:', hunter.says())
Elmer Fudd says: I'm hunting wabbits.

>>> hunted1 = QuestionQuote('Bugs Bunny', "What's up, doc")
>>> print(hunted1.who(), 'says:', hunted1.says())
Bugs Bunny says: What's up, doc?

>>> hunted2 = ExclamationQuote('Daffy Duck', "It's rabbit season")
>>> print(hunted2.who(), 'says:', hunted2.says())
Daffy Duck says: It's rabbit season!
```

Three different versions of the `says()` method provide different behavior for the three classes. This is traditional **polymorphism** in object-oriented languages. Python goes a little further and lets you run the `who()` and `says()` methods of *any* objects that have them. Let's define a class called BabblingBrook that has no relation to our previous woodsy hunter and huntees (descendants of the Quote class):

```
>>> class BabblingBrook():
...     def who(self):
...         return 'Brook'
...     def says(self):
...         return 'Babble'
...
>>> brook = BabblingBrook()
```

Now, run the who() and says() methods of various objects, one (brook) completely unrelated to the others:

```
>>> def who_says(obj):
...     print(obj.who(), 'says', obj.says())
...
>>> who_says(hunter)
Elmer Fudd says I'm hunting wabbits.
>>> who_says(hunted1)
Bugs Bunny says What's up, doc?
>>> who_says(hunted2)
Daffy Duck says It's rabbit season!
>>> who_says(brook)
Brook says Babble
```

This behavior is sometimes called *duck typing*, after the old saying:

If it walks like a duck and quacks like a duck, it's a duck.

— A Wise Person

Special Methods

You can now create and use basic objects, but now let's go a bit deeper and do more.

When you type something such as a = 3 + 8, how do the integer objects with values 3 and 8 know how to implement +? Also, how does a know how to use = to get the result? You can get at these operators by using Python's *special methods* (you might also see them called *magic methods*). You don't need Gandalf to perform any magic, and they're not even complicated.

The names of these methods begin and end with double underscores (__). You've already seen one: __init__ initializes a newly created object from its class definition and any arguments that were passed in.

Suppose that you have a simple Word class, and you want an equals() method that compares two words but ignores case. That is, a Word containing the value 'ha' would be considered equal to one containing 'HA'.

The example that follows is a first attempt, with a normal method we're calling equals(). self.text is the text string that this Word object contains, and the equals() method compares it with the text string of word2 (another Word object):

```
>>> class Word():
...     def __init__(self, text):
...         self.text = text
...
...     def equals(self, word2):
...         return self.text.lower() == word2.text.lower()
...
```

Then, make three Word objects from three different text strings:

```
>>> first = Word('ha')
>>> second = Word('HA')
>>> third = Word('eh')
```

When strings 'ha' and 'HA' are compared to lowercase, they should be equal:

```
>>> first.equals(second)
True
```

But the string 'eh' will not match 'ha':

```
>>> first.equals(third)
False
```

We defined the method equals() to do this lowercase conversion and comparison. It would be nice to just say if first == second, just like Python's built-in types. So, let's do that. We change the equals() method to the special name __eq__() (you'll see why in a moment):

```
>>> class Word():
...     def __init__(self, text):
...         self.text = text
...     def __eq__(self, word2):
...         return self.text.lower() == word2.text.lower()
...
```

Let's see if it works:

```
>>> first = Word('ha')
>>> second = Word('HA')
>>> third = Word('eh')
>>> first == second
True
>>> first == third
False
```

Magic! All we needed was the Python's special method name for testing equality, __eq__(). Tables 6-1 and 6-2 list the names of the most useful magic methods.

Table 6-1. Magic methods for comparison

`__eq__( self, other )`	`self == other`
`__ne__( self, other )`	`self != other`
`__lt__( self, other )`	`self < other`
`__gt__( self, other )`	`self > other`
`__le__( self, other )`	`self <= other`
`__ge__( self, other )`	`self >= other`

Table 6-2. Magic methods for math

`__add__( self, other )`	`self + other`
`__sub__( self, other )`	`self - other`
`__mul__( self, other )`	`self * other`
`__floordiv__( self, other )`	`self // other`
`__truediv__( self, other )`	`self / other`
`__mod__( self, other )`	`self % other`
`__pow__( self, other )`	`self ** other`

You aren't restricted to use the math operators such as + (magic method `__add__()`) and - (magic method `__sub__()`) with numbers. For instance, Python string objects use + for concatenation and * for duplication. There are many more, documented online at Special method names. The most common among them are presented in Table 6-3.

Table 6-3. Other, miscellaneous magic methods

`__str__( self )`	`str( self )`
`__repr__( self )`	`repr( self )`
`__len__( self )`	`len( self )`

Besides `__init__()`, you might find yourself using `__str__()` the most in your own methods. It's how you print your object. It's used by `print()`, `str()`, and the string formatters that you can read about in Chapter 7. The interactive interpreter uses the `__repr__()` function to echo variables to output. If you fail to define either `__str__()` or `__repr__()`, you get Python's default string version of your object:

```
>>> first = Word('ha')
>>> first
<__main__.Word object at [illegible]>
>>> print(first)
<__main__.Word object at [illegible]>
```

Let's add both `__str__()` and `__repr__()` methods to the `Word` class to make it prettier:

```
>>> class Word():
...     def __init__(self, text):
...         self.text = text
...     def __eq__(self, word2):
...         return self.text.lower() == word2.text.lower()
...     def __str__(self):
...         return self.text
...     def __repr__(self):
...         return 'Word("'  self.text  '")'
...
>>> first = Word('ha')
>>> first          # uses __repr__
Word("ha")
>>> print(first)   # uses __str__
ha
```

To explore even more special methods, check out the Python documentation.

Composition

Inheritance is a good technique to use when you want a child class to act like its parent class most of the time (when child *is-a* parent). It's tempting to build elaborate inheritance hierarchies, but sometimes *composition* or *aggregation* (when x *has-a* y) make more sense. A duck *is-a* bird, but *has-a* tail. A tail is not a kind of duck, but part of a duck. In this next example, let's make bill and tail objects and provide them to a new duck object:

```
>>> class Bill():
...     def __init__(self, description):
...         self.description = description
...
>>> class Tail():
...     def __init__(self, length):
...         self.length = length
...
>>> class Duck():
...     def __init__(self, bill, tail):
...         self.bill = bill
...         self.tail = tail
...     def about(self):
...         print('This duck has a', bill.description, 'bill and a',
...             tail.length, 'tail')
...
>>> tail = Tail('long')
>>> bill = Bill('wide orange')
>>> duck = Duck(bill, tail)
>>> duck.about()
This duck has a wide orange bill and a long tail
```

— III Unit —