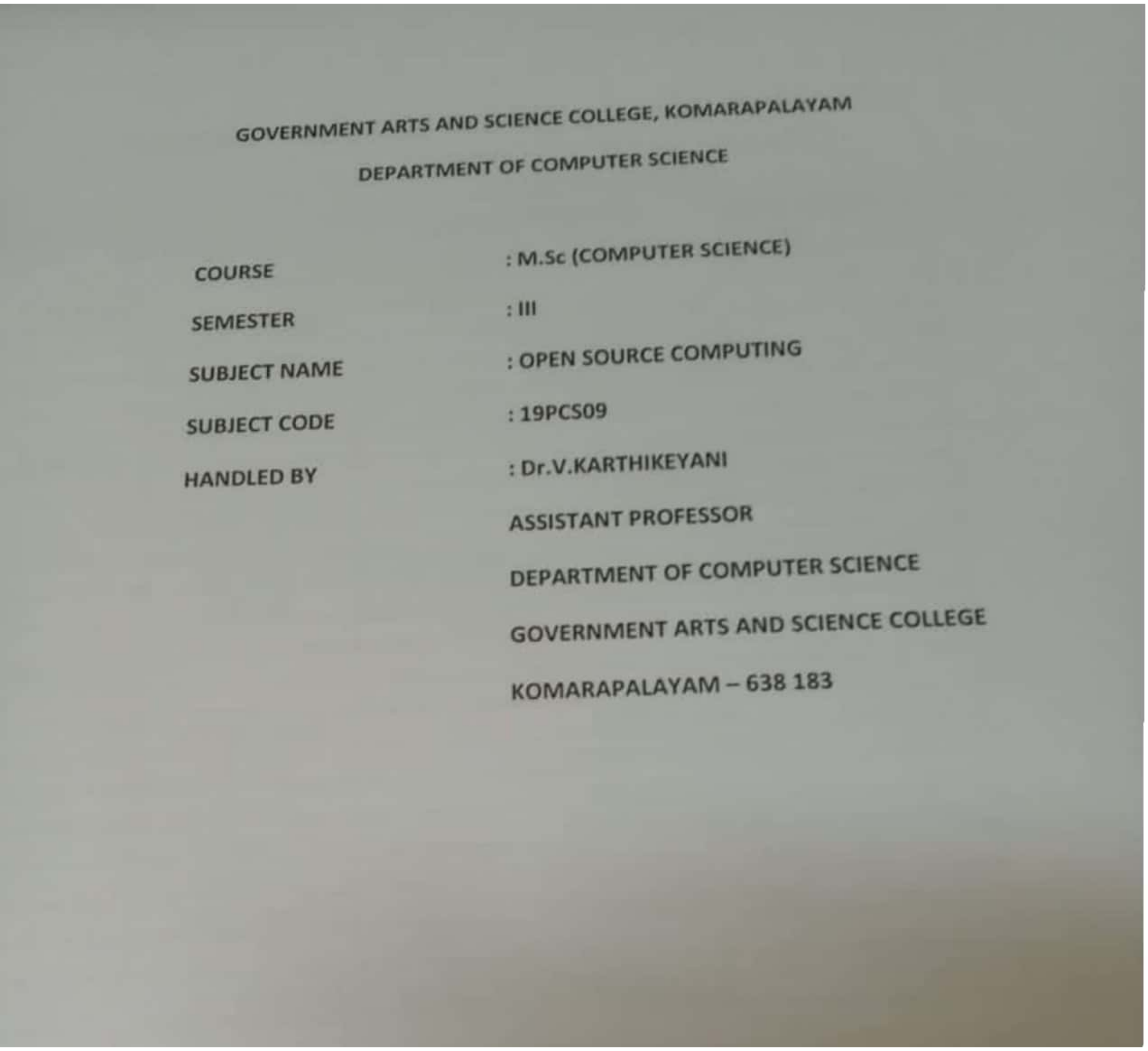

GOVERNMENT ARTS AND SCIENCE COLLEGE, KOMARAPALAYAM

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

III SEMESTER

Core Course- XIII-17PCS09 OPEN SOURCE COMPUTING

Credits: 4

Course Objective:

- To understand the basic Concepts of Python

UNIT - I

Python: Introduction – Numbers – Strings – Variables – Lists – Tuples – Dictionaries – Sets – Comparison.

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

UNIT-III

Data Types: Text Strings – Binary Data. **Storing and Retrieving Data**: File Input/Output – Structured Text Files – Structured Binary Files - Relational Databases – NoSQL Data Stores.

UNIT-IV

Web: Web Clients – Web Servers – Web Services and Automation – **Systems:** Files – Directories – Programs and Processes – Calendars and Clocks

UNIT-V

Concurrency: Queues – Processes – Threads – Green Threads and gevent – twisted – Redis. **Networks:** Patterns – The Publish-Subscribe Model – TCP/IP – Sockets – ZeroMQ –Internet Services – Web Services and APIs – Remote Processing – Big Fat Data and MapReduce – Working in the Clouds.

TEXT BOOK

1. Bill Lubanovic, "Introducing Python", O'Reilly, First Edition-Second Release, 2014.

REFERENCE BOOKS

1. Mark Lutz, "Learning Python", O'Reilly, Fifth Edition, 2013.

David M. Beazley,"Python Essential Reference", Developer's Library, Fourth Edition, 2009.

III SEMESTER

Core Course- XIII-17PCS09 OPEN SOURCE COMPUTING

Credits: 4

Course Objective:

- To understand the basic Concepts of Python

UNIT - I

Python: Introduction – Numbers – Strings – Variables – Lists – Tuples – Dictionaries – Sets – Comparison.

I unit

Py Ingredients: Numbers, Strings, and Variables

used for:
web development
Software "
System Scripting.

Python → 1991
By:
Guido van Rossum

In this chapter we'll begin by looking at Python's simplest built-in data types:

→ works on different platform (windows, linux, mac)

- *booleans* (which have the value True or False)
- *integers* (whole numbers such as 42 and 100000000)
- *floats* (numbers with decimal points such as 3.14159, or sometimes exponents like 1.0e8, which means *one times ten to the eighth power*, or 100000000.0)
- *strings* (sequences of text characters)

In a way, they're like atoms. We'll use them individually in this chapter. Chapter 3 shows how to combine them into larger "molecules."

Each type has specific rules for its usage and is handled differently by the computer. We'll also introduce *variables* (names that refer to actual data; more on these in a moment).

The code examples in this chapter are all valid Python, but they're snippets. We'll be using the Python interactive interpreter, typing these snippets and seeing the results immediately. Try running them yourself with the version of Python on your computer. You'll recognize these examples by the >>> prompt. In Chapter 4, we start writing Python programs that can run on their own.

Variables, Names, and Objects

In Python, *everything*—booleans, integers, floats, strings, even large data structures, functions, and programs—is implemented as an *object*. This gives the language a consistency (and useful features) that some other languages lack.

An object is like a clear plastic box that contains a piece of data (Figure 2-1). The object has a *type*, such as boolean or integer, that determines what can be done with the data. A real-life box marked "Pottery" would tell you certain things (it's probably heavy, and don't drop it on the floor). Similarly, in Python, if an object has the type `int`, you know that you could add it to another `int`.

Figure 2-1. An object is like a box

The type also determines if the data *value* contained by the box can be changed (*mutable*) or is constant (*immutable*). Think of an immutable object as a closed box with a clear window: you can see the value but you can't change it. By the same analogy, a mutable object is like an open box: not only can you see the value inside, you can also change it; however, you can't change its type.

Python is *strongly typed*, which means that the type of an object does not change, even if its value is mutable (Figure 2-2).

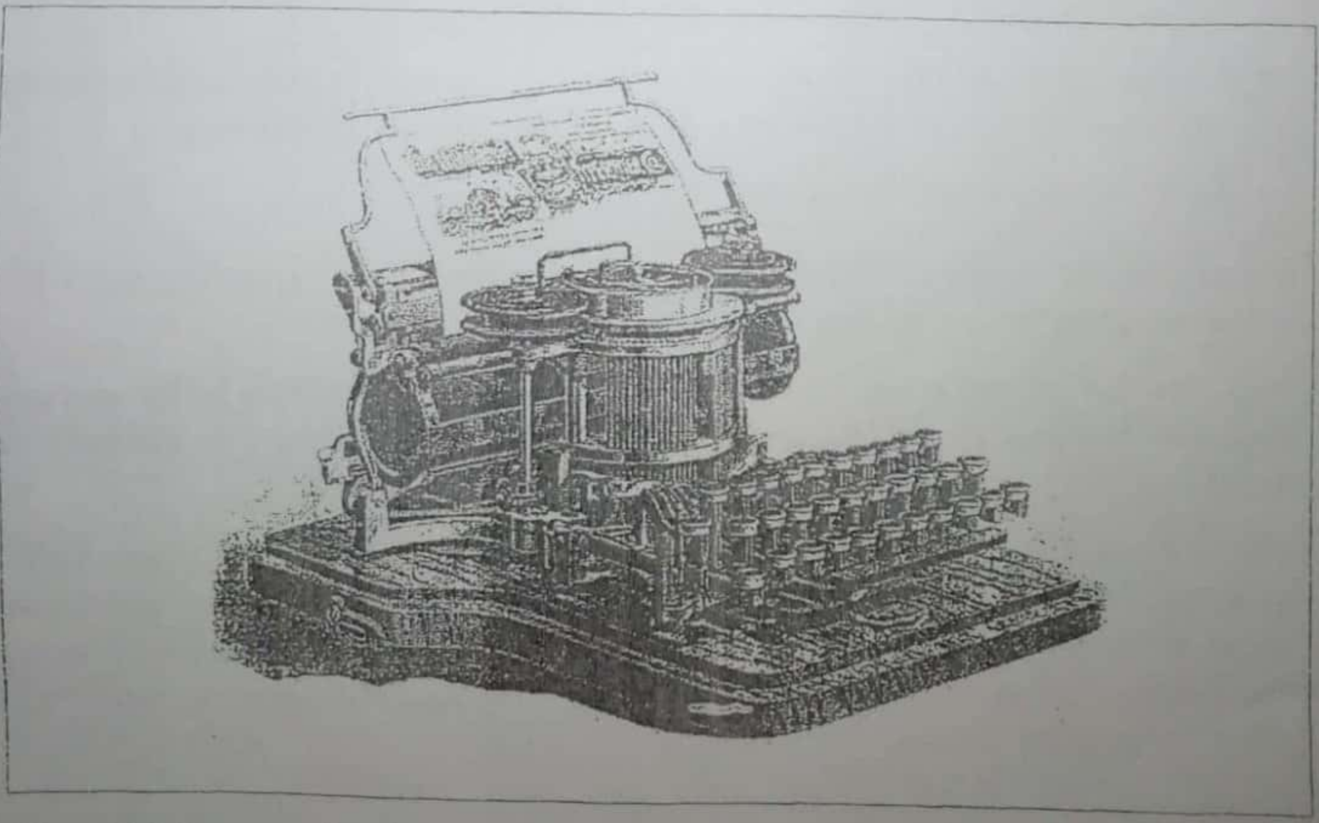

Figure 2-2. Strong typing does not mean push the keys harder

Programming languages allow you to define *variables*. These are names that refer to values in the computer's memory that you can define for use with your program. In Python, you use = to *assign* a value to a variable.

We all learned in grade school math that = means *equal to*. So why do many computer languages, including Python, use = for assignment? One reason is that standard keyboards lack logical alternatives such as a left arrow key, and = didn't seem too confusing. Also, in computer programs you use assignment much more than you test for equality.

The following is a two-line Python program that assigns the integer value 7 to the variable named a, and then prints the value currently associated with a:

```
>>> a = 7
>>> print(a)
7
```

Now, it's time to make a crucial point about Python variables: *variables are just names*. Assignment **does not copy** a value; it just **attaches** a name to the object that contains the data. The name is a *reference* to a thing rather than the thing itself. Think of a name as a sticky note (see Figure 2-3).

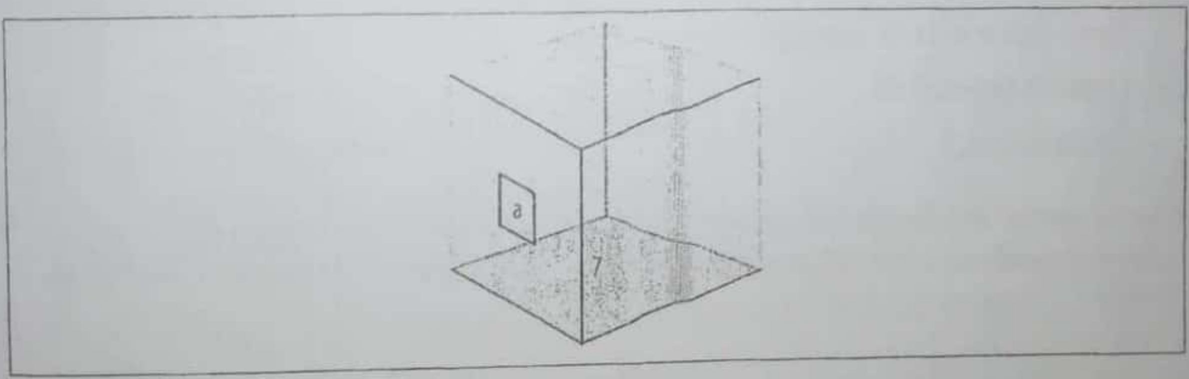

Figure 2-3. Names stick to objects

Try this with the interactive interpreter:

1. As before, assign the value 7 to the name a. This creates an object box containing the integer value 7.
2. Print the value of a.
3. Assign a to b, making b also stick to the object box containing 7.
4. Print the value of b.

```
>>> a =
>>> print(a)
```

```
7
>>> b = a
>>> print(b)
7
```

In Python, if you want to know the type of anything (a variable or a literal value), use `type( thing )`. Let's try it with different literal values (`58`, `99.9`, `abc`) and different variables (`a`, `b`):

```
>>> type(a)
<class 'int'>
>>> type(b)
<class 'int'>
>>> type(58)
<class 'int'>
>>> type(99.9)
<class 'float'>
>>> type('abc')
<class 'str'>
```

A *class* is the definition of an object; Chapter 6 covers classes in greater detail. In Python, "class" and "type" mean pretty much the same thing.

Variable names can only contain these characters:

- Lowercase letters (`a` through `z`)
- Uppercase letters (`A` through `Z`)
- Digits (`0` through `9`)
- Underscore (`_`)

Names cannot begin with a digit. Also, Python treats names that begin with an underscore in special ways (which you can read about in Chapter 4). These are valid names:

- `a`
- `a1`
- `a b c___95`
- `_abc`
- `_1a`

These names, however, are not valid:

- `1`
- `1a`
- `1_`

Finally, don't use any of these for variable names, because they are Python's *reserved words*:

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

These words, and some punctuation, are used to define Python's syntax. You'll see all of them as you progress through this book.

Numbers

Python has built-in support for *integers* (whole numbers such as 5 and 1,000,000,000) and *floating point* numbers (such as 3.1416, 14.99, and 1.87e4). You can calculate combinations of numbers with the simple math *operators* in this table:

Operator	Description	Example	Result
+	addition	5 + 8	13
-	subtraction	90 - 10	80
*	multiplication	4 * 7	28
/	floating point division	7 / 2	3.5
//	integer (truncating) division	7 // 2	3
%	modulus (remainder)	7 % 3	1
**	exponentiation	3 ** 4	81

$3^4 = 9 \times 9 = 81.$

$$2 \overline{)7} \quad \boxed{3.5} \quad 6 \quad 10 \quad 10 \quad 0$$

$$2 \overline{)7} \quad 2 \quad 6 \quad \boxed{1}$$

For the next few pages, I'll show simple examples of Python acting as a glorified calculator.

Integers

Any sequence of digits in Python is assumed to be a literal *integer*:

```
>>> 5
5
```

You can use a plain zero (0):

```
>>> 0
0
```

But don't put it in front of other digits:

```
>>> 05
  File "<stdin>", line 1
    05
     ^
SyntaxError: invalid token
```

This is the first time you've seen a Python *exception*—a program error. In this case, it's a warning that `05` is an "invalid token." I'll explain what this means in "Bases" on page 24. You'll see many more examples of exceptions in this book because they're Python's main error handling mechanism.

A sequence of digits specifies a positive integer. If you put a + sign before the digits, the number stays the same:

```
>>> 123
123
>>> +123
123
```

To specify a negative integer, insert a – before the digits:

```
>>> -123
-123
```

You can do normal arithmetic with Python, much as you would with a calculator, by using the operators listed in the table on the previous page. Addition and subtraction work as you'd expect:

```
>>> 5 + 9
14
>>> 100 - 7
93
>>> 4 - 10
-6
```

You can include as many numbers and operators as you'd like:

```
>>> 5 + 9 + 3
17
>>> 4 + 3 - 2 - 1 + 6
10
```

A style note: you're not required to have a space between each number and operator:

```
>>> 5+9   +      3
17
```

It just looks better and is easier to read.

Multiplication is also straightforward:

```
>>> 6 * 7
42
>>> 7 * 6
42
>>> 6 * 7 * 2 * 3
252
```

$$\begin{array}{r} 42 \times 2 \\ \hline 84 \times 3 \\ \hline 252 \end{array}$$

Division is a little more interesting, because it comes in two flavors:

- / carries out *floating-point* (decimal) division
- // performs *integer* (truncating) division

Even if you're dividing an integer by an integer, using a / will give you a floating-point result:

```
>>> 9 / 5
1.8
```

Truncating integer division gives you an integer answer, throwing away any remainder:

```
>>> 9 // 5
1
```

Dividing by zero with either kind of division causes a Python exception:

```
>>> 5 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 7 // 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by z
```

All of the preceding examples used literal integers. You can mix literal integers and variables that have been assigned integer values:

```
>>> a = 95
>>> a
95
>>> a - 3
92
```

Earlier, when we said a - 3, we didn't assign the result to a, so the value of a did not change:

```
>>> a
95
```

If you wanted to change a, you would do this:

```
>>> a = a - 3
>>> a
92
```

Scanned with CamScanner
Scanned with CamScanner

This usually confuses beginning programmers, again because of our ingrained grade school math training, we see that = sign and think of equality. In Python, the expression on the right side of the = is calculated first, *then* assigned to the variable on the left side.

If it helps, think of it this way:

- Subtract 3 from a
- Assign the result of that subtraction to a temporary variable
- Assign the value of the temporary variable to a:

```
>>> a = 95
>>> temp = a - 3
>>> a = temp
```

So, when you say:

```
>>> a = a - 3
```

Python is calculating the subtraction on the righthand side, remembering the result, and then assigning it to a on the left side of the = sign. It's faster and neater than using a temporary variable.

You can combine the arithmetic operators with assignment by putting the operator before the =. Here, a -= 3 is like saying a = a - 3:

```
>>> a = 95
>>> a -= 3
>>> a
92
```

This is like a = a + 8:

```
>>> a += 8
>>> a
100
```

And this is like a = a * 2:

```
>>> a *= 2
>>> a
200
```

Here's a floating-point division example, such as a = a / 3:

```
>>> a /= 3
>>> a
66.66666666666667
```

Let's assign 13 to a, and then try the shorthand for a = a // 4 (truncating integer division):

```
>>> a = 13
>>> a //= 4
>>> a
```

The % character has multiple uses in Python. When it's between two numbers, it produces the remainder when the first number is divided by the second:

```
>>> 9 % 5
4
```

Here's how to get both the (truncated) quotient and remainder at once:

```
>>> divmod(9,5)
(1, 4)
```

Otherwise, you could have calculated them separately:

```
>>> 9 // 5
1
>>> 9 % 5
4
```

You just saw some new things here: a *function* named `divmod` is given the integers 9 and 5 and returns a two-item result called a *tuple*. Tuples will take a bow in Chapter 3; functions will make their debut in Chapter 4.

Precedence

What would you get if you typed the following?

```
>>> 2+3*4
```

If you do the addition first, `2 + 3` is 5, and `5 * 4` is `20`. But if you do the multiplication first, `3 * 4` is 12, and `2 + 12` is 14. In Python, as in most languages, multiplication has higher *precedence* than addition, so the second version is what you'd see:

```
>>> 2 + 3 * 4
14
```

How do you know the precedence rules? There's a big table in Appendix F that lists them all, but I've found that in practice I never look up these rules. It's much easier to just add parentheses to group your code as you intend the calculation to be carried out:

```
>>> 2 + (3 * 4)
14
```

This way, anyone reading the code doesn't need to guess its intent or look up precedence rules.

Bases

Integers are assumed to be decimal (base 10) unless you use a prefix to specify another *base*. You might never need to use these other bases, but you'll probably see them in Python code somewhere, sometime.

We generally have 10 fingers or toes (one of my cats has a few more, but rarely uses them for counting). So, we count 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Next, we run out of digits and carry the one to the "ten's place" and put a 0 in the one's place: 10 means "1 ten and 0 ones". We don't have a single digit that represents "ten." Then, it's 11, 12, up to 19, carry the one to make 20 (2 tens and 0 ones), and so on.

A base is how many digits you can use until you need to "carry the one." In base 2 (`binary`), the only digits are 0 and 1. 0 is the same as a plain old decimal 0, and 1 is the same as a decimal 1. However, in base 2, if you add a 1 to a 1, you get 10 (1 decimal two plus 0 decimal ones).

In Python, you can express literal integers in three bases besides decimal:

- `0b` or `0B` for *binary* (base 2).
- `0o` or `0O` for *octal* (base 8).
- `0x` or `0X` for *hex* (base 16).

The interpreter prints these for you as decimal integers. Let's try each of these bases. First, a plain old decimal `10`, which means *1 ten and 0 ones*.

```
>>> 10
10
```

Now, a binary (base two), which means *1 (decimal) two and 0 ones*:

```
>>> 0b10
2
```

Octal (base 8) for *1 (decimal) eight and 0 ones*:

```
>>> 0o10
8
```

Hexadecimal (base 16) for *1 (decimal) 16 and 0 ones*:

```
>>> 0x10
16
```

In case you're wondering what "digits" base 16 uses, they are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, and f. `0xa` is a decimal `10`, and `0xf` is a decimal `15`. Add 1 to `0xf` and you get `0x10` (decimal 16).

Why use a different base from 10? It's useful in *bit-level* operations, which are described in Chapter 7, along with more details about converting numbers from one base to another.

Type Conversions

To change other Python data types to an integer, use the `int()` function. This will keep the whole number and discard any fractional part.

Python's simplest data type is the *boolean*, which has only the values `True` and `False`. When converted to integers, they represent the values `1` and `0`:

```
>>> int(True)
1
>>> int(False)
0
```

Converting a floating-point number to an integer just lops off everything after the decimal point:

```
>>> int(98.6)
98
>>> int([illegible])
[illegible]
```

Finally, here's an example of converting a text string (you'll see more about strings in a few pages) that contains only digits, possibly with + or - signs:

```
>>> int('99')
99
>>> int('-23')
-23
>>> int('+12')
12
```

Converting an integer to an integer doesn't change anything but doesn't hurt either:

```
>>> int(12345)
12345
```

If you try to convert something that doesn't look like a number, you'll get an *exception*:

```
>>> int('99 bottles of beer on the wall')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '99 bottles of beer on the wall'
>>> int('')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: ''
```

The preceding text string started with valid digit characters (`99`), but it kept on going with others that the `int()` function just wouldn't stand for.

We'll get to exceptions in Chapter 4. For now, just know that it's how Python alerts you that an error occurred (rather than crashing the program, as some languages might do). Instead of assuming that things always go right, I'll show many examples of exceptions throughout this book, so you can see what Python does when they go wrong.

`int()` will make integers from floats or strings of digits, but won't handle strings containing decimal points or exponents:

```
>>> int('98.6')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '98.6'
>>> int('1.0e4')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1.0e4'
```

If you mix numeric types, Python will sometimes try to automatically convert them for you:

```
>>> 4 + 7.0
11.0
```

The boolean value `False` is treated as `0` or `0.0` when mixed with integers or floats, and `True` is treated as `1` or `1.0`:

```
>>> True + 2
3
>>> False + 5.0
5.0
```

How Big Is an int?

In Python 2, the size of an `int` was limited to 32 bits. This was enough room to store any integer from –2,147,483,648 to 2,147,483,647.

A `long` had even more room: 64 bits, allowing values from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. In Python 3, `long` is long gone, and an `int` can be *any* size—even greater than 64 bits. Thus, you can say things like the following (`10**100` is called a *googol*, and was the original name of Google before they decided on the easier spelling):

```
>>>
>>> googol = 10**100
>>> googol
10000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000
>>> googol * googol
10000000000000000000000000000000000000000000000000000000000000000000000000
```

```
[illegible]
[illegible]
```

In many languages, trying this would cause something called *integer overflow*, where the number would need more space than the computer allowed for it, causing various bad effects. Python handles humungous integers with no problem. Score one for Python.

Floats

Integers are whole numbers, but *floating-point* numbers (called *floats* in Python) have decimal points. Floats are handled similarly to integers: you can use the operators (+, -, *, /, //, **, and %) and divmod() function.

To convert other types to floats, you use the float() function. As before, booleans act like tiny integers:

```
>>> float(True)
1.0
>>> float(False)
0.0
```

Converting an integer to a float just makes it the proud possessor of a decimal point:

```
>>> float(98)
98.0
>>> float('99')
99.0
```

And, you can convert a string containing characters that would be a valid float (digits, signs, decimal point, or an e followed by an exponent) to a real float:

```
>>> float('98.6')
98.6
>>> float('-1.5')
-1.5
>>> float('1.0e4')
10000.0
```

Math Functions

Python has the usual math functions such as square roots, cosines, and so on. We'll save them for Appendix C, in which we also discuss Python uses in science.

Strings

Nonprogrammers often think that programmers must be good at math because they work with numbers. Actually, most programmers work with *strings* of text much more than numbers. Logical (and creative!) thinking is often more important than math skills.

Because of its support for the Unicode standard, Python 3 can contain characters from any written language in the world, plus a lot of symbols. Its handling of that standard was a big reason for its split from Python 2. It's also a good reason to use version 3. I'll get into Unicode in various places, because it can be daunting at times. In the string examples that follow, I'll mostly use ASCII examples.

Strings are our first example of a Python *sequence*. In this case, they're a sequence of characters.

Unlike other languages, strings in Python are *immutable*. You can't change a string in-place, but you can copy parts of strings to another string to get the same effect. You'll see how to do this shortly.

Create with Quotes

You make a Python string by enclosing characters in either single quotes or double quotes, as demonstrated in the following:

```
>>> 'Snap'
'Snap'
>>> "Crackle"
'Crackle'
```

The interactive interpreter echoes strings with a single quote, but all are treated exactly the same by Python.

Why have two kinds of quote characters? The main purpose is so that you can create strings containing quote characters. You can have single quotes inside double-quoted strings, or double quotes inside single-quoted strings:

```
>>> "'Nay,' said the naysayer."
"'Nay,' said the naysayer."
>>> 'The rare double quote in captivity: ".'
'The rare double quote in captivity: ".'
>>> 'A "two by four" is actually 1 1/2" × 3 1/2".'
'A "two by four is" actually 1 1/2" × 3 1/2".'
>>> "'There's the man that shot my paw!' cried the limping hound."
"'There's the man that shot my paw!' cried the limping hound."
```

You can also use three single quotes (''') or three double quotes ("""):

```
>>> '''Boom!'''
'Boom'
>>> """Eek!"""
'Eek!'
```

Triple quotes aren't very useful for short strings like these. Their most common use is to create *multiline strings*, like this classic poem from Edward Lear:

```
>>> poem =  '''There was a Young Lady of Norway,
... Who casually sat in a doorway;
... When the door squeezed her flat,
... She exclaimed, "What of that?"
... This courageous Young Lady of Norway.'''
>>>
```

(This was entered in the interactive interpreter, which prompted us with `>>>` for the first line and `...` until we entered the final triple quotes and went to the next line.)

If you tried to create that poem with single quotes, Python would make a fuss when you went to the second line:

```
>>> poem = 'There was a young lady of Norway,
  File "<stdin>", line 1
    poem = 'There was a young lady of Norway,
                                            ^
SyntaxError: EOL while scanning string literal
>>>
```

If you have multiple lines within triple quotes, the line ending characters will be preserved in the string. If you have leading or trailing spaces, they'll also be kept:

```
>>> poem2 = '''I do not like thee, Doctor Fell.
...     The reason why, I cannot tell.
...     But this I know, and know full well:
...     I do not like thee, Doctor Fell.
... '''
>>> print(poem2)
I do not like thee, Doctor Fell.
    The reason why, I cannot tell.
    But this I know, and know full well:
    I do not like thee, Doctor Fell.

>>>
```

By the way, there's a difference between the output of `print()` and the automatic echoing done by the interactive interpreter:

```
>>> poem2
'I do not like thee, Doctor Fell.\n    The reason why, I cannot tell.\n    But
this I know, and know full well:\n    I do not like thee, Doctor Fell.\n'
```

`print()` strips quotes from strings and prints their contents. It's meant for human output. It helpfully adds a space between each of the things it prints, and a newline at the end:

```
>>> print(99, 'bottles', 'would be enough.')
99 bottles would be enough.
```

If you don't want the space or newline, you'll see how to avoid them shortly.

The interpreter prints the string with single quotes and *escape characters* such as `\n`, which are explained in "Escape with \" on page 30.

Finally, there is the *empty string*, which has no characters at all but is perfectly valid. You can create an empty string with any of the aforementioned quotes:

```
>>> ''
''
>>> ""
''
>>> ''''''
''
>>> """"""
''
>>>
```

Why would you need an empty string? Sometimes you might want to build a string from other strings, and you need to start with a blank slate.

```
>>> bottles = 99
>>> base = ''
>>> base += 'current inventory: '
>>> base += str(bottles)
>>> base
'current inventory: 99'
```

Convert Data Types by Using str()

You can convert other Python data types to strings by using the `str()` function:

```
>>> str(98.6)
'98.6'
>>> str(1.0e4)
'10000.0'
>>> str(True)
'True'
```

Python uses the `str()` function internally when you call `print()` with objects that are not strings and when doing *string interpolation*, which you'll see in Chapter 7.

Escape with \

Python lets you *escape* the meaning of some characters within strings to achieve effects that would otherwise be hard to express. By preceding a character with a backslash (\), you give it a special meaning. The most common escape sequence is `\n`, which means to begin a new line. With this you can create multiline strings from a one-line string.

```
>>> palindrome = 'A man,\nA plan,\nA canal:\nPanama.'
>>> print(palindrome)
A man,
A plan,
A canal:
Panama.
```

You will see the escape sequence `\t` (tab) used to align text:

```
>>> print('\tabc')
    abc
>>> print('a\tbc')
a   bc
>>> print('ab\tc')
ab      c
>>> print('abc\t')
abc
```

(The final string has a terminating tab which, of course, you can't see.)

You might also need `\'` or `\"` to specify a literal single or double quote inside a string that's quoted by the same character:

```
>>> testimony = "\"I did nothing!\" he said. \"Not that either! Or the other
 thing.\""
>>> print(testimony)
"I did nothing!" he said. "Not that either! Or the other thing."
>>> fact = "The world's largest rubber duck was 54'2\" by 65'7\" by 105'"
>>> print(fact)
The world's largest rubber duck was 54'2" by 65'7" by 105'
```

And if you need a literal backslash, just type two of them:

```
>>> speech = 'Today we honor our friend, the backslash: \\.'
>>> print(speech)
Today we honor our friend, the backslash: \.
```

Combine with +

You can combine literal strings or string variables in Python by using the `+` operator, as demonstrated here:

```
>>> 'Release the kraken! ' + 'At once!'
'Release the kraken! At once!'
```

You can also combine *literal strings* (not string variables) just by having one after the other:

```
>>> "My word! " "A gentleman caller!"
'My word! A gentleman caller!'
```

Python does not add spaces for you when concatenating strings, so in the preceding example, we needed to include spaces explicitly. It does add a space between each argument to a `print()` statement, and a newline at the end:

```
>>> a = 'Duck.'
>>> b = a
>>> c = 'Grey Duck!'
>>> a + b + c
'Duck.Duck.Grey Duck!'
```

```
>>> print(a, b, c)
Duck. Duck. Grey Duck!
```

Duplicate with *

You use the * operator to duplicate a string. Try typing these lines into your interactive interpreter and see what they print:

```
>>> start = 'Na ' * 4 + '\n'
>>> middle = 'Hey ' * 3 + '\n'
>>> end = 'Goodbye.'
>>> print(start + start + middle + end)
```

Extract a Character with []

To get a single character from a string, specify its *offset* inside square brackets after the string's name. The first (leftmost) offset is 0, the next is 1, and so on. The last (rightmost) offset can be specified with –1 so you don't have to count; going to the left are –2, –3, and so on.

```
>>> letters = 'abcdefghijklmnopqrstuvwxyz'
>>> letters[0]
'a'
>>> letters[1]
'b'
>>> letters[-1]
'z'
>>> letters[-2]
'y'
>>> letters[25]
'z'
>>> letters[5]
'f'
```

If you specify an offset that is the length of the string or longer (remember, offsets go from 0 to length–1), you'll get an exception:

```
>>> letters[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Indexing works the same with the other sequence types (lists and tuples), which we cover in Chapter 3.

Because strings are immutable, you can't insert a character directly into one or change the character at a specific index. Let's try to change `'Henny'` to `'Penny'` and see what happens:

```
>>> name = 'Henny'
>>> name[0] = 'P'
```

```
Traceback (most recent call last):
  File "<stdin>", line  , in <module>
TypeError: 'str' object does not support item assignment
```

Instead you need to use some combination of string functions such as `replace()` or a *slice* (which you'll see in a moment):

```
>>> name = 'Henny'
>>> name.replace('H', 'P')
'Penny'
>>> 'P' + name[ :]
'Penny'
```

Slice with [*start* : *end* : *step*]

You can extract a *substring* (a part of a string) from a string by using a *slice*. You define a slice by using square brackets, a *start* offset, an *end* offset, and an optional *step* size. Some of these can be omitted. The slice will include characters from offset *start* to one before *end*.

- `[:]` extracts the entire sequence from start to end.
- `[ start :]` specifies from the *start* offset to the end.
- `[: end ]` specifies from the beginning to the *end* offset minus 1.
- `[ start : end ]` indicates from the *start* offset to the *end* offset minus 1.
- `[ start : end : step ]` extracts from the *start* offset to the *end* offset minus 1, skipping characters by *step*.

As before, offsets go 0, 1, and so on from the start to the right, and –1,–2, and so forth from the end to the left. If you don't specify *start*, the slice uses 0 (the beginning). If you don't specify *end*, it uses the end of the string.

Let's make a string of the lowercase English letters:

```
>>> letters = 'abcdefghijklmnopqrstuvwxyz'
```

Using a plain `:` is the same as `0:` (the entire string):

```
>>> letters[:]
'abcdefghijklmnopqrstuvwxyz'
```

Here's an example from offset 20 to the end:

```
>>> letters[20:]
'uvwxyz'
```

Now, from offset 10 to the end:

```
>>> letters[10:]
'klmnopqrstuvwxyz'
```

And another, offset 12 to 14 (Python does not include the last offset):

```
>>> letters[12:15]
'mno'
```

The three last characters:

```
>>> letters[-3:]
'xyz'
```

In this next example, we go from offset 18 to the fourth before the end; notice the difference from the previous example, in which starting at –3 gets the x, but ending at –3 actually stops at –4, the w:

```
>>> letters[18:-3]
'stuvw'
```

In the following, we extract from 6 before the end to 3 before the end:

```
>>> letters[-6:-2]
'uvwx'
```

If you want a step size other than 1, specify it after a second colon, as shown in the next series of examples.

From the start to the end, in steps of 7 characters:

```
>>> letters[::7]
'ahov'
```

From offset 4 to 19, by 3:

```
>>> letters[4:20:3]
'ehknqt'
```

From offset 19 to the end, by 4:

```
>>> letters[19::4]
'tx'
```

From the start to offset 20 by 5:

```
>>> letters[:21:5]
'afkpu'
```

(Again, the *end* needs to be one more than the actual offset.)

And that's not all! Given a negative step size, this handy Python slicer can also step backward. This starts at the end and ends at the start, skipping nothing:

```
>>> letters[-1::-1]
'zyxwvutsrqponmlkjihgfedcba'
```

It turns out that you can get the same result by using this:

```
>>> letters[::-1]
'zyxwvutsrqponmlkjihgfedcba'
```

Slices are more forgiving of bad offsets than are single-index lookups. A slice offset earlier than the beginning of a string is treated as `0`, and one after the end is treated as `-1`, as is demonstrated in this next series of examples.

From 50 before the end to the end:

```
>>> letters[-50:]
'abcdefghijklmnopqrstuvwxyz'
```

From 51 before the end to 50 before the end:

```
>>> letters[-51:-50]
''
```

From the start to 69 after the start:

```
>>> letters[:70]
'abcdefghijklmnopqrstuvwxyz'
```

From 70 after the start to 70 after the start:

```
>>> letters[70:71]
''
```

Get Length with len()

So far, we've used special punctuation characters such as `+` to manipulate strings. But there are only so many of these. Now, we start to use some of Python's built-in *functions*: named pieces of code that perform certain operations.

The `len()` function counts characters in a string:

```
>>> len(letters)
26
>>> empty = ""
>>> len(empty)
0
```

You can use `len()` with other sequence types, too, as is described in Chapter 3.

Split with split()

Unlike `len()`, some functions are specific to strings. To use a string function, type the name of the string, a dot, the name of the function, and any *arguments* that the function needs: *`string`* `.` *`function`* `(` *`arguments`* `)`. You'll see a longer discussion of functions in "Functions" on page 85.

You can use the built-in string `split()` function to break a string into a *list* of smaller strings based on some *separator*. You'll see lists in the next chapter. A list is a sequence of values, separated by commas and surrounded by square brackets.

```
>>> todos = 'get gloves,get mask,give cat vitamins,call ambulance'
>>> todos.split(',')
['get gloves', 'get mask', 'give cat vitamins', 'call ambulance']
```

In the preceding example, the string was called todos and the string function was called `split()`, with the single separator argument `','`. If you don't specify a separator, `split()` uses any sequence of white space characters—newlines, spaces, and tabs.

```
>>> todos.split()
['get', 'gloves,get', 'mask,give', 'cat', 'vitamins,call', 'ambulance']
```

You still need the parentheses when calling `split` with no arguments—that's how Python knows you're calling a function.

Combine with join()

In what may not be an earthshaking revelation, the `join()` function is the opposite of `split()`: it collapses a list of strings into a single string. It looks a bit backward because you specify the string that glues everything together first, and then the list of strings to glue: *`string`* `.join(` *`list`* `)`. So, to join the list `lines` with separating newlines, you would say `'\n'.join(lines)`. In the following example, let's join some names in a list with a comma and a space:

```
>>> crypto_list = ['Yeti', 'Bigfoot', 'Loch Ness Monster']
>>> crypto_string = ', '.join(crypto_list)
>>> print('Found and signing book deals:', crypto_string)
Found and signing book deals: Yeti, Bigfoot, Loch Ness Monster
```

Playing with Strings

Python has a large set of string functions. Let's explore how the most common of them work. Our test subject is the following string containing the text of the immortal poem "What Is Liquid?" by Margaret Cavendish, Duchess of Newcastle:

```
>>> poem = '''All that doth flow we cannot liquid name
Or else would fire and water be the same;
But that is liquid which is moist and wet
Fire that property can never get.
Then 'tis not cold that doth the fire put out
But 'tis the wet that makes it die, no doubt.'''
```

To begin, get the first 13 characters (offsets 0 to 12):

```
>>> poem[:13]
'All that doth'
```

How many characters are in this poem? (Spaces and newlines are included in the count.)

```
>>> len(poem)
250
```

Does it start with the letters `All`?

```
>>> poem.startswith('All')
True
```

Does it end with `That's all, folks!`?

```
>>> poem.endswith('That\'s all, folks!')
False
```

Now, let's find the offset of the first occurrence of the word `the` in the poem:

```
>>> word = 'the'
>>> poem.find(word)
73
```

And the offset of the last `the`:

```
>>> poem.rfind(word)
214
```

How many times does the three-letter sequence `the` occur?

```
>>> poem.count(word)
3
```

Are all of the characters in the poem either letters or numbers?

```
>>> poem.isalnum()
False
```

Nope, there were some punctuation characters.

Case and Alignment

In this section, we'll look at some more uses of the built-in string functions. Our test string is the following:

```
>>> setup = 'a duck goes into a bar...'
```

Remove `.` sequences from both ends:

```
>>> setup.strip('.')
'a duck goes into a bar'
```

Because strings are immutable, none of these examples actually changes the `setup` string. Each example just takes the value of `setup`, does something to it, and returns the result as a new string.

Capitalize the first word:

```
>>> setup.capitalize()
'A duck goes into a bar...'
```

Capitalize all the words:

```
>>> setup.title()
'A Duck Goes Into A Bar...'
```

Convert all characters to uppercase:

```
>>> setup.upper()
'A DUCK GOES INTO A BAR...'
```

Convert all characters to lowercase:

```
>>> setup.lower()
'a duck goes into a bar...'
```

Swap upper- and lowercase:

```
>>> setup.swapcase()
'a DUCK GOES INTO A BAR...'
```

Now, we'll work with some layout alignment functions. The string is aligned within the specified total number of spaces (30 here).

Center the string within 30 spaces:

```
>>> setup.center(30)
'  a duck goes into a bar...   '
```

Left justify:

```
>>> setup.ljust(30)
'a duck goes into a bar...     '
```

Right justify:

```
>>> setup.rjust(30)
'     a duck goes into a bar...'
```

I have much more to say about string formatting and conversions in Chapter 7, including how to use % and format().

Substitute with replace()

You use replace() for simple substring substitution. You give it the old substring, the new one, and how many instances of the old substring to replace. If you omit this final count argument, it replaces all instances. In this example, only one string is matched and replaced:

```
>>> setup.replace('duck', 'marmoset')
'a marmoset goes into a bar...'
```

Change up to 100 of them:

```
>>> setup.replace('a ', 'a famous ', 100)
'a famous duck goes into a famous bar...'
```

When you know the exact substring(s) you want to change, `replace()` is a good choice. But watch out. In the second example, if we had substituted for the single character string `'a'` rather than the two character string `'a '` (a followed by a space), we would have also changed a in the middle of other words:

```
>>> setup.replace('a', 'a famous', 100)
'a famous duck goes into a famous ba famousr...'
```

Sometimes, you want to ensure that the substring is a whole word, or the beginning of a word, and so on. In those cases, you need *regular expressions*, which are described in detail in Chapter 7.

More String Things

Python has many more string functions than I've shown here. Some will turn up in later chapters, but you can find all the details at the standard documentation link.

Things to Do

This chapter introduced the atoms of Python: numbers, strings, and variables. Let's try a few small exercises with them in the interactive interpreter.

2.1 How many seconds are in an hour? Use the interactive interpreter as a calculator and multiply the number of seconds in a minute (`60`) by the number of minutes in an hour (also `60`).

2.2 Assign the result from the previous task (seconds in an hour) to a variable called `seconds_per_hour`.

2.3 How many seconds are in a day? Use your `seconds_per_hour` variable.

2.4 Calculate seconds per day again, but this time save the result in a variable called `seconds_per_day`.

2.5 Divide `seconds_per_day` by `seconds_per_hour`. Use floating-point (`/`) division.

2.6 Divide `seconds_per_day` by `seconds_per_hour`, using integer (`//`) division. Did this number agree with the floating-point value from the previous question, aside from the final `.0`?

Unit - I

CHAPTER 3

Py Filling: Lists, Tuples, Dictionaries, and Sets

In Chapter 2 we started at the bottom with Python's basic data types: booleans, integers, floats, and strings. If you think of those as atoms, the data structures in this chapter are like molecules. That is, we combine those basic types in more complex ways. You will use these every day. Much of programming consists of chopping and glueing data into specific forms, and these are your hacksaws and glue guns.

Lists and Tuples

Most computer languages can represent a sequence of items indexed by their integer position: first, second, and so on down to the last. You've already seen Python *strings*, which are sequences of characters. You've also had a little preview of lists, which you'll now see are sequences of anything.

Python has two other sequence structures: *tuples* and *lists*. These contain zero or more elements. Unlike strings, the elements can be of different types. In fact, each element can be *any* Python object. This lets you create structures as deep and complex as you like.

Why does Python contain both lists and tuples? Tuples are *immutable*; when you assign elements to a tuple, they're baked in the cake and can't be changed. Lists are *mutable*, meaning you can insert and delete elements with great enthusiasm. I'll show many examples of each, with an emphasis on lists.

By the way, you might hear two different pronunciations for *tuple*. Which is right? If you guess wrong, do you risk being considered a Python poseur? No worries. Guido van Rossum, the creator of Python, tweeted "I pronounce tuple too-pull on Mon/Wed/Fri and tub-pull on Tue/Thu/Sat. On Sunday I don't talk about them. :)"

Lists

Lists are good for keeping track of things by their order, especially when the order and contents might change. Unlike strings, lists are mutable. You can change a list in-place, add new elements, and delete or overwrite existing elements. The same value can occur more than once in a list.

Create with [] or list()

wee kday [5]

A list is made from zero or more elements, separated by commas, and surrounded by square brackets:

```
>>> empty_list = [ ]
>>> weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
>>> big_birds = ['emu', 'ostrich', 'cassowary']
>>> first_names = ['Graham', 'John', 'Terry', 'Terry', 'Michael']
```

You can also make an empty list with the `list()` function:

```
>>> another_empty_list = list()
>>> another_empty_list
[]
```

"Comprehensions" on page 81 shows one more way to create a list, called a *list comprehension*.

The `weekdays` list is the only one that actually takes advantage of list order. The `first_names` list shows that values do not need to be unique.

If you only want to keep track of unique values and don't care about order, a Python *set* might be a better choice than a list. In the previous example, `big_birds` could have been a set. You'll read about sets a little later in this chapter.

Convert Other Data Types to Lists with list()

Python's `list()` function converts other data types to lists. The following example converts a string to a list of one-character strings:

```
>>> list('cat')
['c', 'a', 't']
```

This example converts a *tuple* (coming up after lists in this chapter) to a list:

```
>>> a_tuple = ('ready', 'fire', 'aim')
>>> list(a_tuple)
['ready', 'fire', 'aim']
```

As I mentioned earlier in "Split with split()" on page 35, use `split()` to chop a string into a list by some separator string:

```
>>> birthday = '1/6/1952'
>>> birthday.split('/')
['1', '6', '1952']
```

What if you have more than one separator string in a row in your original string? Well, you get an empty string as a list item:

```
>>> splitme = 'a/b//c/d///e'
>>> splitme.split('/')
['a', 'b', '', 'c', 'd', '', '', 'e']
```

If you had used the two-character separator string `//` instead, you would get this:

```
>>> splitme = 'a/b//c/d///e'
>>> splitme.split('//')
>>>
['a/b', 'c/d', '/e']
```

Get an Item by Using [*offset*]

As with strings, you can extract a single value from a list by specifying its offset:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[0]
'Groucho'
>>> marxes[1]
'Chico'
>>> marxes[2]
'Harpo'
```

Again, as with strings, negative indexes count backward from the end:

```
>>> marxes[-1]
'Harpo'
>>> marxes[-2]
'Chico'
>>> marxes[-3]
'Groucho'
>>>
```

The offset has to be a valid one for this list—a position you have assigned a value previously. If you specify an offset before the beginning or after the end, you'll get an exception (error). Here's what happens if we try to get the sixth Marx brother (offset 5 counting from 0), or the fifth before the end:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

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

Lists of Lists

Lists can contain elements of different types, including other lists, as illustrated here:

```
>>> small_birds = ['hummingbird', 'finch']
>>> extinct_birds = ['dodo', 'passenger pigeon', 'Norwegian Blue']
>>> carol_birds = [3, 'French hens', 2, 'turtledoves']
>>> all_birds = [small_birds, extinct_birds, 'macaw', carol_birds]
```

So what does all_birds, a list of lists, look like?

```
>>> all_birds
[['hummingbird', 'finch'], ['dodo', 'passenger pigeon', 'Norwegian Blue'], 'macaw',
[3, 'French hens', 2, 'turtledoves']]
```

Let's look at the first item in it:

```
>>> all_birds[0]
['hummingbird', 'finch']
```

The first item is a list: in fact, it's small_birds, the first item we specified when creating all_birds. You should be able to guess what the second item is:

```
>>> all_birds[1]
['dodo', 'passenger pigeon', 'Norwegian Blue']
```

Scanned with CamScanner
Scanned with CamScanner

It's the second item we specified, `extinct_birds`. If we want the first item of `extinct_birds`, we can extract it from `all_birds` by specifying two indexes:

```
>>> all_birds[1][0]
'dodo'
```

The `[1]` refers to the list that's the second item in `all_birds`, whereas the `[0]` refers to the first item in that inner list.

Change an Item by [*offset*]

Just as you can get the value of a list item by its offset, you can change it:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[2] = 'Wanda'
>>> marxes
['Groucho', 'Chico', 'Wanda']
```

Again, the list offset needs to be a valid one for this list.

You can't change a character in a string in this way, because strings are immutable. Lists are mutable. You can change how many items a list contains, and the items themselves.

Get a Slice to Extract Items by Offset Range

You can extract a subsequence of a list by using a *slice*:

```
>>> marxes = ['Groucho', 'Chico,' 'Harpo']
>>> marxes[0:2]
['Groucho', 'Chico']
```

A slice of a list is also a list.

As with strings, slices can step by values other than one. The next example starts at the beginning and goes right by 2:

```
>>> marxes[::2]
['Groucho', 'Harpo']
```

Here, we start at the end and go left by 2:

```
>>> marxes[::-2]
['Harpo', 'Groucho']
```

And finally, the trick to reverse a list:

```
>>> marxes[::-1]
['Harpo', 'Chico', 'Groucho']
```

Add an Item to the End with append()

The traditional way of adding items to a list is to append() them one by one to the end. In the previous examples, we forgot Zeppo, but that's all right because the list is mutable, so we can add him now:

```
>>> marxes.append('Zeppo')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo']
```

Combine Lists by Using extend() or +=

You can merge one list into another by using extend(). Suppose that a well-meaning person gave us a new list of Marxes called others, and we'd like to merge them into the main marxes list:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes.extend(others)
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', 'Gummo', 'Karl']
```

Alternatively, you can use +=:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes += others
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', 'Gummo', 'Karl']
```

If we had used append(), others would have been added as a *single* list item rather than merging its items:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes.append(others)
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', ['Gummo', 'Karl']]
```

This again demonstrates that a list can contain elements of different types. In this case, four strings, and a list of two strings.

Add an Item by Offset with insert()

The append() function adds items only to the end of the list. When you want to add an item before any offset in the list, use insert(). Offset 0 inserts at the beginning. An offset beyond the end of the list inserts at the end, like append(), so you don't need to worry about Python throwing an exception.

```
>>> marxes.insert(3, 'Gummo')
>>> marxes
```

```
['Groucho', 'Chico', 'Harpo', 'Gummo', 'Zeppo']
>>> marxes.insert(10, 'Karl')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Gummo', 'Zeppo', 'Karl']
```

Delete an Item by Offset with del

Our fact checkers have just informed us that Gummo was indeed one of the Marx Brothers, but Karl wasn't. Let's undo that last insertion:

```
>>> del marxes[-1]
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Gummo', 'Zeppo']
```

When you delete an item by its position in the list, the items that follow it move back to take the deleted item's space, and the list's length decreases by one. If we delete `'Harpo'` from the last version of the `marxes` list, we get this as a result:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Gummo', 'Zeppo']
>>> marxes[2]
'Harpo'
>>> del marxes[2]
>>> marxes
['Groucho', 'Chico', 'Gummo', 'Zeppo']
>>> marxes[2]
'Gummo'
```

`del` is a Python *statement*, not a list method—you don't say `marxes[-2].del()`. It's sort of the reverse of assignment (=): it detaches a name from a Python object and can free up the object's memory if that name was the last reference to it.

Delete an Item by Value with remove()

If you're not sure or don't care where the item is in the list, use `remove()` to delete it by value. Goodbye, Gummo:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Gummo', 'Zeppo']
>>> marxes.remove('Gummo')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo']
```

Get an Item by Offset and Delete It by Using pop()

You can get an item from a list and delete it from the list at the same time by using `pop()`. If you call `pop()` with an offset, it will return the item at that offset; with no argument, it uses `-1`. So, `pop(0)` returns the head (start) of the list, and `pop()` or `pop(-1)` returns the tail (end), as shown here:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> marxes.pop()
'Zeppo'
>>> marxes
['Groucho', 'Chico', 'Harpo']
>>> marxes.pop(1)
'Chico'
>>> marxes
['Groucho', 'Harpo']
```

It's computing jargon time! Don't worry, these won't be on the final exam. If you use `append()` to add new items to the end and `pop()` to remove them from the same end, you've implemented a data structure known as a *LIFO* (last in, first out) queue. This is more commonly known as a *stack*. `pop(0)` would create a *FIFO* (first in, first out) queue. These are useful when you want to collect data as they arrive and work with either the oldest first (FIFO) or the newest first (LIFO).

Find an Item's Offset by Value with index()

If you want to know the offset of an item in a list by its value, use `index()`:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> marxes.index('Chico')
1
```

Test for a Value with in

The Pythonic way to check for the existence of a value in a list is using `in`:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> 'Groucho' in marxes
True
>>> 'Bob' in marxes
False
```

The same value may be in more than one position in the list. As long as it's in there at least once, `in` will return `True`:

```
>>> words = ['a', 'deer', 'a' 'female', 'deer']
>>> 'deer' in words
True
```

If you check for the existence of some value in a list often and don't care about the order of items, a Python *set* is a more appropriate way to store and look up unique values. We'll talk about sets a little later in this chapter.

Count Occurrences of a Value by Using count()

To count how many times a particular value occurs in a list, use `count()`:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.count('Harpo')
1
>>> marxes.count('Bob')
0

>>> snl_skit = ['cheeseburger', 'cheeseburger', 'cheeseburger']
>>> snl_skit.count('cheeseburger')
3
```

Convert to a String with join()

"Combine with join()" on page 36 discusses `join()` in greater detail, but here's another example of what you can do with it:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> ', '.join(marxes)
'Groucho, Chico, Harpo'
```

But wait: you might be thinking that this seems a little backward. `join()` is a string method, not a list method. You can't say `marxes.join(', ')`, even though it seems more intuitive. The argument to `join()` is a string or any iterable sequence of strings (including a list), and its output is a string. If `join()` were just a list method, you couldn't use it with other iterable objects such as tuples or strings. If you did want it to work with any iterable type, you'd need special code for each type to handle the actual joining. It might help to remember: *`join()` is the opposite of `split()`*, as demonstrated here:

```
>>> friends = ['Harry', 'Hermione', 'Ron']
>>> separator = ' * '
>>> joined = separator.join(friends)
>>> joined
'Harry * Hermione * Ron'
>>> separated = joined.split(separator)
>>> separated
['Harry', 'Hermione', 'Ron']
>>> separated == friends
True
```

Reorder Items with sort()

You'll often need to sort the items in a list by their values rather than their offsets. Python provides two functions:

- The list function `sort()` sorts the list itself, *in place*.
- The general function `sorted()` returns a sorted *copy* of the list.

If the items in the list are numeric, they're sorted by default in ascending numeric order. If they're strings, they're sorted in alphabetical order:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> sorted_marxes = sorted(marxes)
>>> sorted_marxes
['Chico', 'Groucho', 'Harpo']
```

`sorted_marxes` is a copy, and creating it did not change the original list:

```
>>> marxes
['Groucho', 'Chico', 'Harpo']
```

But, calling the list function `sort()` on the `marxes` list does change `marxes`:

```
>>> marxes.sort()
>>> marxes
['Chico', 'Groucho', 'Harpo']
```

If the elements of your list are all of the same type (such as strings in `marxes`), `sort()` will work correctly. You can sometimes even mix types—for example, integers and floats—because they are automatically converted to one another by Python in expressions:

```
>>> numbers = [2, 1, 4.0, 3]
>>> numbers.sort()
>>> numbers
[1, 2, 3, 4.0]
```

The default sort order is ascending, but you can add the argument `reverse=True` to set it to descending:

```
>>> numbers = [2, 1, 4.0, 3]
>>> numbers.sort(reverse=True)
>>> numbers
[4.0, 3, 2, 1]
```

Get Length by Using len()

`len()` returns the number of items in a list:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> len(marxes)
3
```

Assign with =, Copy with copy()

When you assign one list to more than one variable, changing the list in one place also changes it in the other, as illustrated here:

```
>>> a = [1, 2, 3]
>>> a
[1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = 'surprise'
>>> a
['surprise', 2, 3]
```

So what's in b now? Is it still [1, 2, 3], or ['surprise', 2, 3]? Let's see:

```
>>> b
['surprise', 2, 3]
```

Remember the sticky note analogy in Chapter 2? b just refers to the same list object as a; therefore, whether we change the list contents by using the name a or b, it's reflected in both:

```
>>> b
['surprise', 2, 3]
>>> b[0] = 'I hate surprises'
>>> b
['I hate surprises', 2, 3]
>>> a
['I hate surprises', 2, 3]
```

You can *copy* the values of a list to an independent, fresh list by using any of these methods:

- The list copy() function
- The list() conversion function
- The list slice [:]

Our original list will be a again. We'll make b with the list copy() function, c with the list() conversion function, and d with a list slice:

```
>>> a = [1, 2, 3]
>>> b = a.copy()
>>> c = list(a)
>>> d = a[:]
```

Again, b, c, and d are *copies* of a: they are new objects with their own values and no connection to the original list object [1, 2, 3] to which a refers. Changing a does *not* affect the copies b, c, and d:

```
>>> a[0] = 'integer lists are boring'
>>> a
['integer lists are boring', 2, 3]
>>> b
[1, 2, 3]
>>> c
[1, 2, 3]
>>> d
[1, 2, 3]
```

Tuples is a collection which is ordered & unchang-eable

Similar to lists, tuples are sequences of arbitrary items. Unlike lists, tuples are *immutable*, meaning you can't add, delete, or change items after the tuple is defined. So, a tuple is similar to a constant list.

Create a Tuple by Using ()

The syntax to make tuples is a little inconsistent, as we'll demonstrate in the examples that follow.

Let's begin by making an empty tuple using `()`:

```
>>> empty_tuple = ()
>>> empty_tuple
()
```

To make a tuple with one or more elements, follow each element with a comma. This works for one-element tuples:

```
>>> one_marx = 'Groucho',
>>> one_marx
('Groucho',)
```

If you have more than one element, follow all but the last one with a comma:

```
>>> marx_tuple = 'Groucho', 'Chico', 'Harpo'
>>> marx_tuple
('Groucho', 'Chico', 'Harpo')
```

Python includes parentheses when echoing a tuple. You don't need them—it's the trailing commas that really define a tuple—but using parentheses doesn't hurt. You can use them to enclose the values, which helps to make the tuple more visible:

```
>>> marx_tuple = ('Groucho', 'Chico', 'Harpo')
>>> marx_tuple
('Groucho', 'Chico', 'Harpo')
```

Tuples let you assign multiple variables at once:

```
>>> marx_tuple = ('Groucho', 'Chico', 'Harpo')
>>> a, b, c = marx_tuple
>>> a
'Groucho'
>>> b
'Chico'
>>> c
'Harpo'
```

This is sometimes called *tuple unpacking*.

You can use tuples to exchange values in one statement without using a temporary variable:

Interchange

```
>>> password = 'swordfish'
>>> icecream = 'tuttifrutti'
>>> password, icecream = icecream, password
>>> password
'tuttifrutti'
>>> icecream
'swordfish'
>>>
```

The tuple() conversion function makes tuples from other things:

```
>>> marx_list = ['Groucho', 'Chico', 'Harpo']
>>> tuple(marx_list)
('Groucho', 'Chico', 'Harpo')
```

Tuples versus Lists

You can often use tuples in place of lists, but they have many fewer functions—there is no append(), insert(), and so on—because they can't be modified after creation. Why not just use lists instead of tuples everywhere?

- Tuples use less space.
- You can't clobber tuple items by mistake.
- You can use tuples as dictionary keys (see the next section).
- *Named tuples* (see "Named Tuples" on page 141) can be a simple alternative to objects.
- Function arguments are passed as tuples (see "Functions" on page 85).

I won't go into much more detail about tuples here. In everyday programming, you'll use lists and dictionaries more. Which is a perfect segue to...

Dictionaries

A *dictionary* is similar to a list, but the order of items doesn't matter, and they aren't selected by an offset such as 0 or 1. Instead, you specify a unique *key* to associate with each value. This key is often a string, but it can actually be any of Python's immutable types: boolean, integer, float, tuple, string, and others that you'll see in later chapters. Dictionaries are mutable, so you can add, delete, and change their key-value elements.

If you've worked with languages that support only arrays or lists, you'll love dictionaries.

In other languages, dictionaries might be called *associative arrays*, *hashes*, or *hashmaps*. In Python, a dictionary is also called a *dict* to save syllables.

Create with {}

To create a dictionary, you place curly brackets ({}) around comma-separated *key* : *value* pairs. The simplest dictionary is an empty one, containing no keys or values at all:

```
>>> empty_dict = {}
>>> empty_dict
{}
```

Let's make a small dictionary with quotes from Ambrose Bierce's *The Devil's Dictionary*:

```
>>> bierce = {
...     "day": "A period of twenty-four hours, mostly misspent",
...     "positive": "Mistaken at the top of one's voice",
...     "misfortune": "The kind of fortune that never misses",
...     }
>>>
```

Typing the dictionary's name in the interactive interpreter will print its keys and values:

```
>>> bierce
{'misfortune': 'The kind of fortune that never misses',
'positive': "Mistaken at the top of one's voice",
'day': 'A period of twenty-four hours, mostly misspent'}
```

In Python, it's okay to leave a comma after the last item of a list, tuple, or dictionary. Also, you don't need to indent, as I did in the preceding example, when you're typing keys and values within the curly braces. It just helps readability.

Convert by Using dict()

You can use the `dict()` function to convert two-value sequences into a dictionary. (You might run into such key-value sequences at times, such as "Strontium, 90, Carbon, 14", or "Vikings, 20, Packers, 7".) The first item in each sequence is used as the key and the second as the value.

First, here's a small example using `lol` (a list of two-item lists):

```
>>> lol = [ ['a', 'b'], ['c', 'd'], ['e', 'f'] ]
>>> dict(lol)
{'c': 'd', 'a': 'b', 'e': 'f'}
```

Remember that the order of keys in a dictionary is arbitrary, and might differ depending on how you add items.

Scanned with CamScanner
Scanned with CamScanner

We could have used any sequence containing two-item sequences. Here are other examples.

A list of two-item tuples:

```
>>> lot = [ ('a', 'b'), ('c', 'd'), ('e', 'f') ]
>>> dict(lot)
{'c': 'd', 'a': 'b', 'e': 'f'}
```

A tuple of two-item lists:

```
>>> tol = ( ['a', 'b'], ['c', 'd'], ['e', 'f'] )
>>> dict(tol)
{'c': 'd', 'a': 'b', 'e': 'f'}
```

A list of two-character strings:

```
>>> los = [ 'ab', 'cd', 'ef' ]
>>> dict(los)
{'c': 'd', 'a': 'b', 'e': 'f'}
```

A tuple of two-character strings:

```
>>> tos = ( 'ab', 'cd', 'ef' )
>>> dict(tos)
{'c': 'd', 'a': 'b', 'e': 'f'}
```

The section "Iterate Multiple Sequences with zip()" on page 79 introduces you to a function called `zip()` that makes it easy to create these two-item sequences.

Add or Change an Item by [*key*]

Adding an item to a dictionary is easy. Just refer to the item by its key and assign a value. If the key was already present in the dictionary, the existing value is replaced by the new one. If the key is new, it's added to the dictionary with its value. Unlike lists, you don't need to worry about Python throwing an exception during assignment by specifying an index that's out of range.

Let's make a dictionary of most of the members of Monty Python, using their last names as keys, and first names as values:

```
>>> pythons = {
...     'Chapman': 'Graham',
...     'Cleese': 'John',
...     'Idle': 'Eric',
...     'Jones': 'Terry',
...     'Palin': 'Michael',
...     }
>>> pythons
{'Cleese': 'John', 'Jones': 'Terry', 'Palin': 'Michael',
'Chapman': 'Graham', 'Idle': 'Eric'}
```

We're missing one member: the one born in America, Terry Gilliam. Here's an attempt by an anonymous programmer to add him, but he's botched the first name:

```
>>> pythons['Gilliam'] = 'Gerry'
>>> pythons
{'Cleese': 'John', 'Gilliam': 'Gerry', 'Palin': 'Michael',
'Chapman': 'Graham', 'Idle': 'Eric', 'Jones': 'Terry'}
```

And here's some repair code by another programmer who is Pythonic in more than one way:

```
>>> pythons['Gilliam'] = 'Terry'
>>> pythons
{'Cleese': 'John', 'Gilliam': 'Terry', 'Palin': 'Michael',
'Chapman': 'Graham', 'Idle': 'Eric', 'Jones': 'Terry'}
```

By using the same key (`'Gilliam'`), we replaced the original value `'Gerry'` with `'Terry'`.

Remember that dictionary keys must be *unique*. That's why we used last names for keys instead of first names here—two members of Monty Python have the first name `Terry`! If you use a key more than once, the last value wins:

```
>>> some_pythons = {
...     'Graham': 'Chapman',
...     'John': 'Cleese',
...     'Eric': 'Idle',
...     'Terry': 'Gilliam',
...     'Michael': 'Palin',
...     'Terry': 'Jones',
...     }
>>> some_pythons
{'Terry': 'Jones', 'Eric': 'Idle', 'Graham': 'Chapman',
'John': 'Cleese', 'Michael': 'Palin'}
```

We first assigned the value `'Gilliam'` to the key `'Terry'` and then replaced it with the value `'Jones'`.

Combine Dictionaries with update()

You can use the `update()` function to copy the keys and values of one dictionary into another.

Let's define the `pythons` dictionary, with all members:

```
>>> pythons = {
...     'Chapman': 'Graham',
...     'Cleese': 'John',
...     'Gilliam': 'Terry',
...     'Idle': 'Eric',
...     'Jones': 'Terry',
...     'Palin': 'Michael',
...     }
>>> pythons
{'Cleese': 'John', 'Gilliam': 'Terry', 'Palin': 'Michael',
'Chapman': 'Graham', 'Idle': 'Eric', 'Jones': 'Terry'}
```

We also have a dictionary of other humorous persons called `others`:

```
>>> others = { 'Marx': 'Groucho', 'Howard': 'Moe' }
```

Now, along comes another anonymous programmer who thinks the members of `others` should be members of Monty Python:

```
>>> pythons.update(others)
>>> pythons
{'Cleese': 'John', 'Howard': 'Moe', 'Gilliam': 'Terry',
'Palin': 'Michael', 'Marx': 'Groucho', 'Chapman': 'Graham',
'Idle': 'Eric', 'Jones': 'Terry'}
```

What happens if the second dictionary has the same key as the dictionary into which it's being merged? The value from the second dictionary wins:

```
>>> first = {'a': 1, 'b': 2}
>>> second = {'b': 'platypus'}
>>> first.update(second)
>>> first
{'b': 'platypus', 'a': 1}
```

Delete an Item by Key with del

Our anonymous programmer's code was correct—technically. But, he shouldn't have done it! The members of `others`, although funny and famous, were not in Monty Python. Let's undo those last two additions:

```
>>> del pythons['Marx']
>>> pythons
{'Cleese': 'John', 'Howard': 'Moe', 'Gilliam': 'Terry',
'Palin': 'Michael', 'Chapman': 'Graham', 'Idle': 'Eric',
'Jones': 'Terry'}
>>> del pythons['Howard']
>>> pythons
{'Cleese': 'John', 'Gilliam': 'Terry', 'Palin': 'Michael',
'Chapman': 'Graham', 'Idle': 'Eric', 'Jones': 'Terry'}
```

Delete All Items by Using clear()

To delete all keys and values from a dictionary, use `clear()` or just reassign an empty dictionary (`{}`) to the name:

```
>>> pythons.clear()
>>> pythons
{}
>>> pythons = {}
>>> pythons
{}
```

Test for a Key by Using in

If you want to know whether a key exists in a dictionary, use `in`. Let's redefine the `pythons` dictionary again, this time omitting a name or two:

```
>>> pythons = {'Chapman': 'Graham', 'Cleese': 'John',
'Jones': 'Terry', 'Palin': 'Michael'}
```

Now let's see who's in there:

```
>>> 'Chapman' in pythons
True
>>> 'Palin' in pythons
True
```

Did we remember to add Terry Gilliam this time?

```
>>> 'Gilliam' in pythons
False
```

Drat.

Get an Item by [*key*]

This is the most common use of a dictionary. You specify the dictionary and key to get the corresponding value:

```
>>> pythons['Cleese']
'John'
```

If the key is not present in the dictionary, you'll get an exception:

```
>>> pythons['Marx']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Marx'
```

There are two good ways to avoid this. The first is to test for the key at the outset by using `in`, as you saw in the previous section:

```
>>> 'Marx' in pythons
False
```

The second is to use the special dictionary `get()` function. You provide the dictionary, key, and an optional value. If the key exists, you get its value:

```
>>> pythons.get('Cleese')
'John'
```

If not, you get the optional value, if you specified one:

```
>>> pythons.get('Marx' , 'Not a Python')
'Not a Python'
```

Otherwise, you get None (which displays nothing in the interactive interpreter):

```
>>> pythons.get('Marx' )
>>>
```

Get All Keys by Using keys()

You can use keys() to get all the keys in a dictionary. We'll use a different sample dictionary for the next few examples:

```
>>> signals = {'green' : 'go', 'yellow': 'go faster', 'red': 'smile for the camera'}
>>> signals.keys()
dict_keys(['green', 'red', 'yellow'])
```

In Python 2, keys() just returns a list. Python 3 returns dict_keys(), which is an iterable view of the keys. This is handy with large dictionaries because it doesn't use the time and memory to create and store a list that you might not use. But often you actually *do* want a list. In Python 3, you need to call list() to convert a dict_keys object to a list.

```
>>> list( signals.keys() )
['green', 'red', 'yellow']
```

In Python 3, you also need to use the list() function to turn the results of values() and items() into normal Python lists. I'm using that in these examples.

Get All Values by Using values()

To obtain all the values in a dictionary, use values():

```
>>> list( signals.values() )
['go', 'smile for the camera', 'go faster']
```

Get All Key-Value Pairs by Using items()

When you want to get all the key-value pairs from a dictionary, use the items() function:

```
>>> list( signals.items() )
[('green', 'go'), ('red', 'smile for the camera'), ('yellow', 'go faster')]
```

Each key and value is returned as a tuple, such as ('green', 'go').

Assign with =, Copy with copy()

As with lists, if you make a change to a dictionary, it will be reflected in all the names that refer to it.

```
>>> signals = {'green': 'go', 'yellow': 'go faster', 'red': 'smile for the camera'}
>>> save_signals = signals
>>> signals['blue'] = 'confuse everyone'
>>> save_signals
{'blue': 'confuse everyone', 'green': 'go',
'red': 'smile for the camera', 'yellow': 'go faster'}
```

To actually copy keys and values from a dictionary to another dictionary and avoid this, you can use copy():

```
>>> signals = {'green': 'go', 'yellow': 'go faster', 'red': 'smile for the camera'}
>>> original_signals = signals.copy()
>>> signals['blue'] = 'confuse everyone'
>>> signals
{'blue': 'confuse everyone', 'green': 'go',
'red': 'smile for the camera', 'yellow': 'go faster'}
>>> original_signals
{'green': 'go', 'red': 'smile for the camera', 'yellow': 'go faster'}
```

Sets is a collection which is unordered & unindexed

A *set* is like a dictionary with its values thrown away, leaving only the keys. As with a dictionary, each key must be unique. You use a set when you only want to know that something exists, and nothing else about it. Use a dictionary if you want to attach some information to the key as a value.

[illegible]

At some bygone time, in some places, set theory was taught in elementary school along with basic mathematics. If your school skipped it (or covered it and you were staring out the window as I often did), Figure 3-1 shows the ideas of union and intersection.

Suppose that you take the union of two sets that have some keys in common. Because a set must contain only one of each item, the union of two sets will contain only one of each key. The *null* or *empty* set is a set with zero elements. In Figure 3-1, an example of a null set would be female names beginning with X.

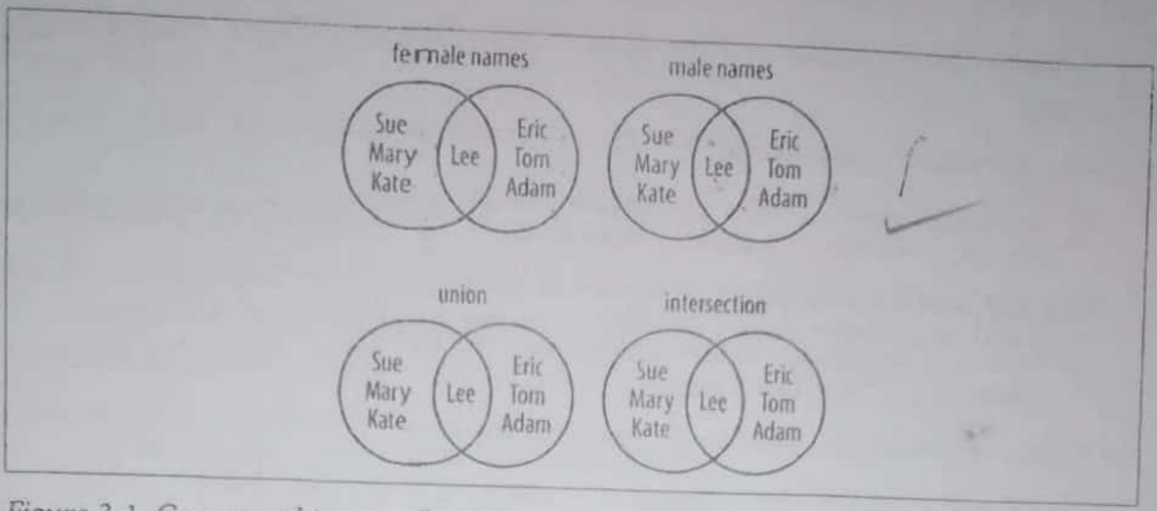

Figure 3-1. Common things to do with sets

Create with set()

To create a set, you use the `set()` function or enclose one or more comma-separated values in curly brackets, as shown here:

```
>>> empty_set = set()
>>> empty_set
set()
>>> even_numbers = {0, 2, 4, 6, 8}
>>> even_numbers
{0, 8, 2, 4, 6}
>>> odd_numbers = {1, 3, 5, 7, 9}
>>> odd_numbers
{9, 3, 1, 5, 7}
```

As with dictionary keys, sets are unordered.

> Because [] creates an empty list, you might expect {} to create an empty set. Instead, {} creates an empty dictionary. That's also why the interpreter prints an empty set as `set()` instead of {}. Why? Dictionaries were in Python first and took possession of the curly brackets.

Convert from Other Data Types with set()

You can create a set from a list, string, tuple, or dictionary, discarding any duplicate values.

First, let's take a look at a string with more than one occurrence of some letters:

```
>>> set( 'letters' )
{'l', 'e', 't', 'r', 's'}
```

Notice that the set contains only one 'e' or 't', even though 'letters' contained two of each.

Now, let's make a set from a list:

```
>>> set( ['Dasher', 'Dancer', 'Prancer', 'Mason-Dixon'] )
{'Dancer', 'Dasher', 'Prancer', 'Mason-Dixon'}
```

This time, a set from a tuple:

```
>>> set( ('Ummagumma', 'Echoes', 'Atom Heart Mother') )
{'Ummagumma', 'Atom Heart Mother', 'Echoes'}
```

When you give set() a dictionary, it uses only the keys:

```
>>> set( {'apple': 'red', 'orange': 'orange', 'cherry': 'red'} )
{'apple', 'cherry', 'orange'}
```

Test for Value by Using in

This is the most common use of a set. We'll make a dictionary called drinks. Each key is the name of a mixed drink, and the corresponding value is a set of its ingredients:

```
>>> drinks = {
...     'martini': {'vodka', 'vermouth'},
...     'black russian': {'vodka', 'kahlua'},
...     'white russian': {'cream', 'kahlua', 'vodka'},
...     'manhattan': {'rye', 'vermouth', 'bitters'},
...     'screwdriver': {'orange juice', 'vodka'}
...     }
```

Even though both are enclosed by curly braces ({ and }), a set is just a sequence of values, and a dictionary is one or more *key* : *value* pairs.

Which drinks contain vodka? (Note that I'm previewing the use of for, if, and, and or from the next chapter for these tests.)

```
>>> for name, contents in drinks.items():
...     if 'vodka' in contents:
...         print(name)
...
screwdriver
martini
black russian
white russian
```

We want something with vodka but are lactose intolerant, and think vermouth tastes like kerosene:

```
>>> for name, contents in drinks.items():
...     if 'vodka' in contents and not ('vermouth' in contents or
...         'cream' in contents):
...         print(name)
...
```

Scanned with CamScanner
Scanned with CamScanner

```
screwdriver
black russian
```

We'll rewrite this a bit more succinctly in the next section.

Combinations and Operators

What if you want to check for combinations of set values? Suppose that you want to find any drink that has orange juice or vermouth? We'll use the *set intersection operator*, which is an ampersand (&):

```
>>> for name, contents in drinks.items():
...     if contents & {'vermouth', 'orange juice'}:
...         print(name)
...
screwdriver
martini
manhattan
```

The result of the & operator is a set, which contains all the items that appear in both lists that you compare. If neither of those ingredients were in contents, the & returns an empty set, which is considered False.

Now, let's rewrite the example from the previous section, in which we wanted vodka but neither cream nor vermouth:

```
>>> for name, contents in drinks.items():
...     if 'vodka' in contents and not contents & {'vermouth', 'cream'}:
...         print(name)
...
screwdriver
black russian
```

Let's save the ingredient sets for these two drinks in variables, just to save typing in the coming examples:

```
>>> bruss = drinks['black russian']
>>> wruss = drinks['white russian']
```

The following are examples of all the set operators. Some have special punctuation, some have special functions, and some have both. We'll use test sets a (contains 1 and 2) and b (contains 2 and 3):

```
>>> a = {1, 2}
>>> b = {2, 3}
```

You get the *intersection* (members common to both sets) with the special punctuation symbol & or the set intersection() function, as demonstrated here:

```
>>> a & b
{2}
```

```
>>> a.intersection(b)
{2}
```

This snippet uses our saved drink variables:

```
>>> bruss & wruss
{'kahlua', 'vodka'}
```

In this example, you get the *union* (members of either set) by using | or the set `union()` function:

```
>>> a | b
{1, 2, 3}
>>> a.union(b)
{1, 2, 3}
```

And here's the alcoholic version:

```
>>> bruss | wruss
{'cream', 'kahlua', 'vodka'}
```

The *difference* (members of the first set but not the second) is obtained by using the character - or `difference()`:

```
>>> a - b
{1}
>>> a.difference(b)
{1}

>>> bruss - wruss
set()
>>> wruss - bruss
{'cream'}
```

By far, the most common set operations are union, intersection, and difference. I've included the others for completeness in the examples that follow, but you might never use them.

The *exclusive or* (items in one set or the other, but not both) uses ^ or `symmetric_difference()`:

```
>>> a ^ b
{1, 3}
>>> a.symmetric_difference(b)
{1, 3}
```

This finds the exclusive ingredient in our two russian drinks:

```
>>> bruss ^ wruss
{'cream'}
```

You can check whether one set is a *subset* of another (all members of the first set are also in the second set) by using <= or `issubset()`:

```
>>> a <= b
False
>>> a.issubset(b)
False
```

Adding cream to a black russian makes a white russian, so wruss is a superset of bruss:

```
>>> bruss <= wruss
True
```

Is any set a subset of itself? Yup.

```
>>> a <= a
True
>>> a.issubset(a)
True
```

To be a *proper subset*, the second set needs to have all the members of the first and more. Calculate it by using <:

```
>>> a < b
False
>>> a < a
False

>>> bruss < wruss
True
```

A *superset* is the opposite of a subset (all members of the second set are also members of the first). This uses >= or issuperset():

```
>>> a >= b
False
>>> a.issuperset(b)
False

>>> wruss >= bruss
True
```

Any set is a superset of itself:

```
>>> a >= a
True
>>> a.issuperset(a)
True
```

And finally, you can find a *proper superset* (the first set has all members of the second, and more) by using >:

```
>>> a > b
False

>>> wruss > bruss
True
```

You can't be a proper superset of yourself:

Compare Data Structures

To review: you make a list by using square brackets ([ ]), a tuple by using commas, and a dictionary by using curly brackets ({}). In each case, you access a single element with square brackets:

```
>>> marx_list = ['Groucho', 'Chico', 'Harpo']
>>> marx_tuple = 'Groucho', 'Chico', 'Harpo'
>>> marx_dict = {'Groucho': 'banjo', 'Chico': 'piano', 'Harpo': 'harp'}
>>> marx_list[2]
'Harpo'
>>> marx_tuple[2]
'Harpo'
>>> marx_dict['Harpo']
'harp'
```

For the list and tuple, the value between the square brackets is an integer offset. For the dictionary, it's a key. For all three, the result is a value.

Make Bigger Data Structures

We worked up from simple booleans, numbers, and strings to lists, tuples, sets, and dictionaries. You can combine these built-in data structures into bigger, more complex structures of your own. Let's start with three different lists:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> pythons = ['Chapman', 'Cleese', 'Gilliam', 'Jones', 'Palin']
>>> stooges = ['Moe', 'Curly', 'Larry']
```

We can make a tuple that contains each list as an element:

```
>>> tuple_of_lists = marxes, pythons, stooges
>>> tuple_of_lists
(['Groucho', 'Chico', 'Harpo'],
['Chapman', 'Cleese', 'Gilliam', 'Jones', 'Palin'],
['Moe', 'Curly', 'Larry'])
```

And, we can make a list that contains the three lists:

```
>>> list_of_lists = [marxes, pythons, stooges]
>>> list_of_lists
[['Groucho', 'Chico', 'Harpo'],
['Chapman', 'Cleese', 'Gilliam', 'Jones', 'Palin'],
['Moe', 'Curly', 'Larry']]
```

Finally, let's create a dictionary of lists. In this example, let's use the name of the comedy group as the key and the list of members as the value:

```
>>> dict_of_lists = {'Marxes': marxes, 'Pythons': pythons, 'Stooges': stooges}
>> dict_of_lists
{'Stooges': ['Moe', 'Curly', 'Larry'],
'Marxes': ['Groucho', 'Chico', 'Harpo'],
'Pythons': ['Chapman', 'Cleese', 'Gilliam', 'Jones', 'Palin']}
```

Your only limitations are those in the data types themselves. For example, dictionary keys need to be immutable, so a list, dictionary, or set can't be a key for another dictionary. But a tuple can be. For example, you could index sites of interest by GPS coordinates (latitude, longitude, and altitude; see "Maps" on page 358 for more mapping examples):

```
>>> houses = {
        (44.79, -93.14, [illegible]): 'My House',
        (38.89, -77.03, [illegible]): 'The White House'
        }
```

Things to Do

In this chapter, you saw more complex data structures: lists, tuples, dictionaries, and sets. Using these and those from Chapter 2 (numbers and strings), you can represent elements in the real world with great variety.

3.1. Create a list called `years_list`, starting with the year of your birth, and each year thereafter until the year of your fifth birthday. For example, if you were born in 1980, the list would be `years_list = [1980, 1981, 1982, 1983, 1984, 1985]`.

If you're less than five years old and reading this book, I don't know what to tell you.

3.2. In which year in `years_list` was your third birthday? Remember, you were 0 years of age for your first year.

3.3. In which year in `years_list` were you the oldest?

3.4. Make a list called `things` with these three strings as elements: `"mozzarella"`, `"cinderella"`, `"salmonella"`.

3.5. Capitalize the element in `things` that refers to a person and then print the list. Did it change the element in the list?

3.6. Make the cheesy element of `things` all uppercase and then print the list.

3.7. Delete the disease element from `things`, collect your Nobel Prize, and print the list.

3.8. Create a list called `surprise` with the elements `"Groucho"`, `"Chico"`, and `"Harpo"`.

3.9. Lowercase the last element of the `surprise` list, reverse it, and then capitalize it.

3.10. Make an English-to-French dictionary called `e2f` and print it. Here are your starter words: `dog` is `chien`, `cat` is `chat`, and `walrus` is `morse`.

3.11. Using your three-word dictionary e2f, print the French word for walrus.

3.12. Make a French-to-English dictionary called f2e from e2f. Use the items method.

3.13. Using f2e, print the English equivalent of the French word chien.

3.14. Make and print a set of English words from the keys in e2f.

3.15. Make a multilevel dictionary called life. Use these strings for the topmost keys: 'animals', 'plants', and 'other'. Make the 'animals' key refer to another dictionary with the keys 'cats', 'octopi', and 'emus'. Make the 'cats' key refer to a list of strings with the values 'Henri', 'Grumpy', and 'Lucy'. Make all the other keys refer to empty dictionaries.

3.16. Print the top-level keys of life.

3.17. Print the keys for life['animals'].

3.18. Print the values for life['animals']['cats'].

—— I unit ——