CHAPTER 2

Py Ingredients: Numbers, Strings, and Variables

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

- *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.

PDF-XChange
Click to buy NOW!
www.docu-track.com

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).

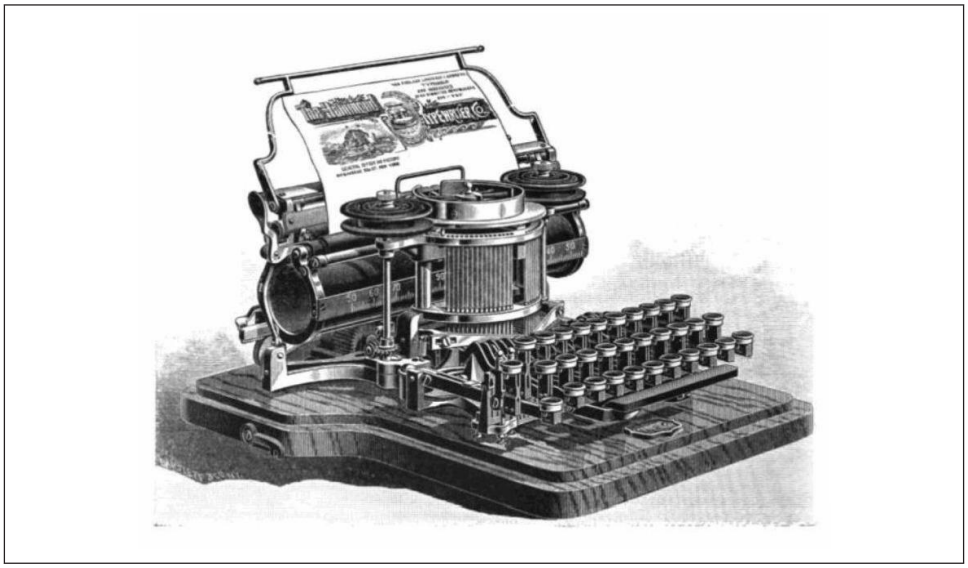

Figure 2-2. Strong typing does not mean push the keys harder

PDF-XChange
Click to buy NOW!
www.docu-track.com

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).

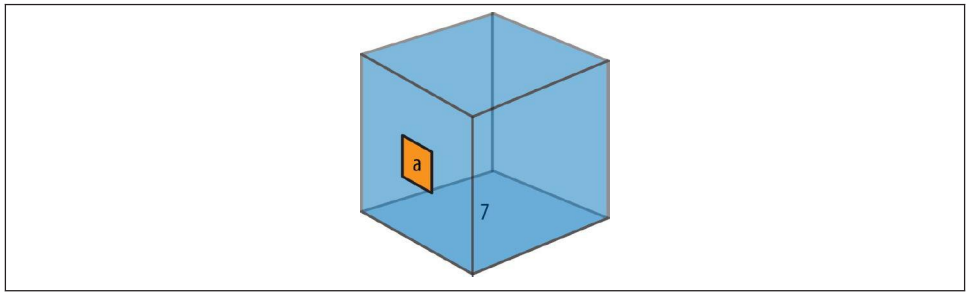

Figure 2-3. Names stick to objects

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

PDF-XChange
Click to buy NOW!
www.docu-track.com

```
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:

+	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

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:

PDF-XChange
Click to buy NOW!
www.docu-track.com

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

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
```

PDF-XChange
Click to buy NOW!
www.docu-track.com

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
```