Chapter 2. Data: Types, Values, Variables, and Names

> *A good name is rather to be chosen than great riches.*
>
> —Proverbs 22:1

Under the hood, everything in your computer is just a sequence of *bits* (see Appendix A). One of the insights of computing is that we can interpret those bits any way we want—as data of various sizes and types (numbers, text characters) or even as computer code itself. We use Python to define chunks of these bits for different purposes, and to get them to and from the CPU.

We begin with Python's data *types* and the *values* that they can contain. Then we see how to represent data as *literal* values and *variables*.

Python Data Are Objects

You can visualize your computer's memory as a long series of shelves. Each slot on one of those memory shelves is one byte wide (eight bits), and slots are numbered from 0 (the first) to the end. Modern computers have billions of bytes of memory (gigabytes), so the shelves would fill a huge imaginary warehouse.

A Python program is given access to some of your computer's memory by your operating system. That memory is used for the code of the program itself, and the data that it uses. The operating system ensures that the program cannot read or write other memory locations without somehow getting permission.

Programs keep track of *where* (memory location) their bits are, and *what* (data type) they are. To your computer, it's all just bits. The same bits mean different things, depending on what type we say they are. The same

bit pattern might stand for the integer `65` or the text character `A`.

Different types may use different numbers of bits. When you read about a "64-bit machine," this means that an integer uses 64 bits (8 bytes).

Some languages plunk and pluck these raw values in memory, keeping track of their sizes and types. Instead of handling such raw data values directly, Python wraps each data value—booleans, integers, floats, strings, even large data structures, functions, and programs—in memory as an *object*. There's a whole chapter (Chapter 10) on how to define your own objects in Python. For now, we're just talking about objects that handle the basic built-in data types.

Using the memory shelves analogy, you can think of objects as variable-sized boxes occupying spaces on those shelves, as shown in Figure 2-1. Python makes these object boxes, puts them in empty spaces on the shelves, and removes them when they're no longer used.

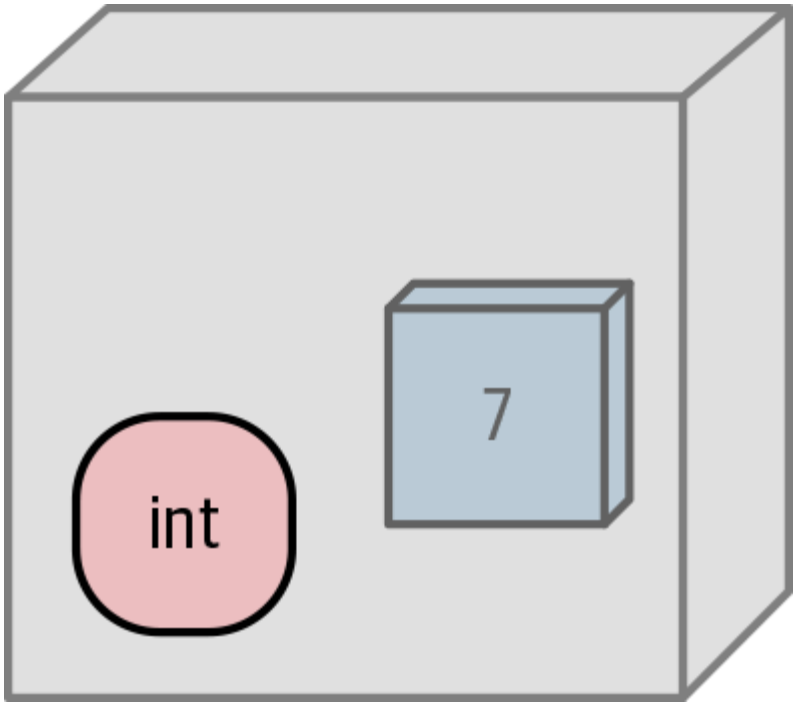

Figure 2-1. An object is like a box; this one is an integer with value 7

In Python, an object is a chunk of data that contains at least the following:

- A *type* that defines what it can do (see the next section)
- A unique *id* to distinguish it from other objects
- A *value* consistent with its type

- A *reference count* that tracks how often this object is used

Its *id* is like its location on the shelf, a unique identifier. Its *type* is like a factory stamp on the box, saying what it can do. If a Python object is an integer, it has the type `int`, and could (among other things, which you'll see in Chapter 3) be added to another `int`. If we picture the box as being made of clear plastic, we can see the *value* inside. You'll learn the use of the *reference count* a few sections from now, when we talk about variables and names.

Types

Table 2-1 shows the basic data types in Python. The second column (Type) contains the Python name of that type. The third column (Mutable?) indicates whether the value can be changed after creation, which I explain more in the next section. Examples shows one or more literal examples of that type. And the final column (Chapter) points you to the chapter in this book with the most details on this type.

Table 2-1. Python's basic data types

Name	Type	Mutable?	Examples	Chapter
Boolean	`bool`	no	`True`, `False`	Chapter 3
Integer	`int`	no	`47`, `25000`, `25_000`	Chapter 3
Floating point	`float`	no	`3.14`, `2.7e5`	Chapter 3
Complex	`complex`	no	`3j`, `5 + 9j`	Chapter 22
Text string	`str`	no	`'alas'`, `"alack"`, `'''a verse attack'''`	Chapter 5
List	`list`	yes	`['Winken', 'Blinken', 'Nod']`	Chapter 7
Tuple	`tuple`	no	`(2, 4, 8)`	Chapter 7
Bytes	`bytes`	no	`b'ab\xff'`	Chapter 12
ByteArray	`bytearray`	yes	`bytearray(...)`	Chapter 12
Set	`set`	yes	`set([3, 5, 7])`	Chapter 8
Frozen set	`frozenset`	no	`frozenset(['Elsa', 'Otto'])`	Chapter 8

Name	Type	Mutable?	Examples	Chapter
Dictionary	`dict`	yes	`{'game': 'bingo', 'dog': 'dingo', 'drummer': 'Ringo'}`	Chapter 8

After the chapters on these basic data types, you'll see how to make new types in Chapter 10.

Mutability

> *Nought may endure but Mutability.*
>
> —Percy Shelley

The type also determines whether the data *value* contained by the box can be changed (*mutable*) or is constant (*immutable*). Think of an immutable object as a sealed box, but with clear sides, like Figure 2-1; you can see the value but you can't change it. By the same analogy, a mutable object is like a box with a lid: 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).

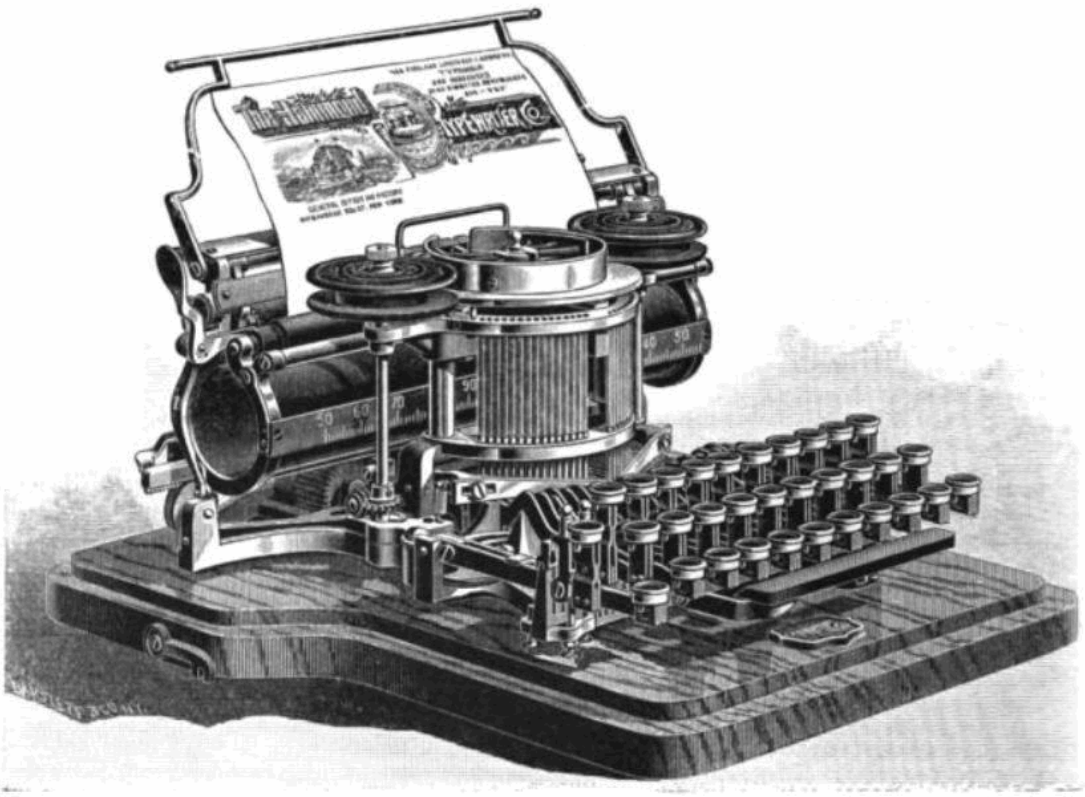

Figure 2-2. *Strong typing* does not mean push the keys harder

Literal Values

There are two ways of specifying data values in Python:

- *Literal*
- *Variable*

In coming chapters, you'll see the details on how to specify literal values for different data types—integers are a sequence of digits, floats contain a decimal point, text strings are surrounded by quotes, and so on. But, for the rest of this chapter, to avoid calloused fingertips, our examples will use only short decimal integers and a Python list or two. Decimal integers are just like integers in math: a sequence of digits from `0` to `9`. There are a few extra integer details (like signs and nondecimal bases) that we look at in Chapter 3.

Variables

Now, we've arrived at a key concept in computing languages.

Python, like most computer languages, lets you define *variables*—names for values in your computer's memory that you want to use in a program.

Python variable names have some rules:

- They can contain only these characters:
  - Lowercase letters (`a` through `z`)
  - Uppercase letters (`A` through `Z`)
  - Digits (`0` through `9`)
  - Underscore (`_`)
- They are *case-sensitive*: `thing` , `Thing` , and `THING` are different names.
- They must begin with a letter or an underscore, not a digit.
- Names that begin with an underscore are treated specially (which you can read about in Chapter 9).
- They cannot be one of Python's *reserved words* (also known as *keywords*).

The reserved words[1] are:

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

Within a Python program, you can find the reserved words with

```
>>> help("keywords")
```

or:

```
>>> import keyword
```

```
>>> keyword.kwlist
```

These are valid names:

- `a`
- `a1`
- `a_b_c___95`
- `_abc`
- `_1a`

These names, however, are not valid:

- `1`
- `1a`
- `1_`
- `name!`
- `another-name`

Assignment

In Python, you use `=` to *assign* a value to a variable.

NOTE

We all learned in grade school arithmetic 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.

Programs are *not* like algebra. When you learned math in school, you saw equations like this:

```
y = x + 12
```

You would solve the equation by "plugging in" a value for `x`. If you gave `x` the value `5`, `5 + 12` is `17`, so the value of `y` would be `17`. Plug in `6` for `x` to get `18` for `y`, and so on.

Computer program lines may look like equations, but their meaning is different. In Python and other computer languages, `x` and `y` are *variables*. Python knows that a bare sequence of digits like `12` or `5` is a literal integer. So here's a tiny Python program that mimics this equation, printing the resulting value of `y`:

```
>>> x = 5
>>> y = x + 12
>>> y
17
```

Here's the big difference between math and programs: in math, `=` means *equality* of both sides, but in programs it means *assignment*: *assign the value on the right side to the variable on the left side*.

Also in programs, everything on the right side needs to have a value (this is called being *initialized*). The right side can be a literal value, or a variable that has already been assigned a value, or a combination. Python knows that `5` and `12` are literal integers. The first line assigns the integer value `5` to the variable `x`. Now we can use the variable `x` in the next line. When Python reads `y = x + 12`, it does the following:

- Sees the `=` in the middle
- Knows that this is an assignment
- Calculates the right side (gets the value of the object referred to by `x` and adds it to `12`)
- Assigns the result to the left-side variable, `y`

Then typing the name of the variable `y` (in the interactive interpreter) will print its new value.

If you started your program with the line `y = x + 12`, Python would generate an *exception* (an error), because the variable `x` doesn't have a value yet:

```
>>> y = x + 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

You'll get the full rundown on exceptions in Chapter 9. In computerese, we'd say that this `x` was *uninitialized*.

In algebra, you can work backward, and assign a value to `y` to calculate `x`. To do this in Python, you'd need to get the literal values and initialized variables on the right side before assigning to `x` on the left:

```
>>> y = 5
>>> x = 12 - y
>>> x
7
```

Variables Are Names, Not Places

Now it's time to make a crucial point about variables in Python: *variables are just names*. This is different from many other computer languages, and a key thing to know about Python, especially when we get to *mutable* objects like lists. 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. Visualize a name as a tag with a string attached to the object box somewhere else in the computer's memory (Figure 2-3).

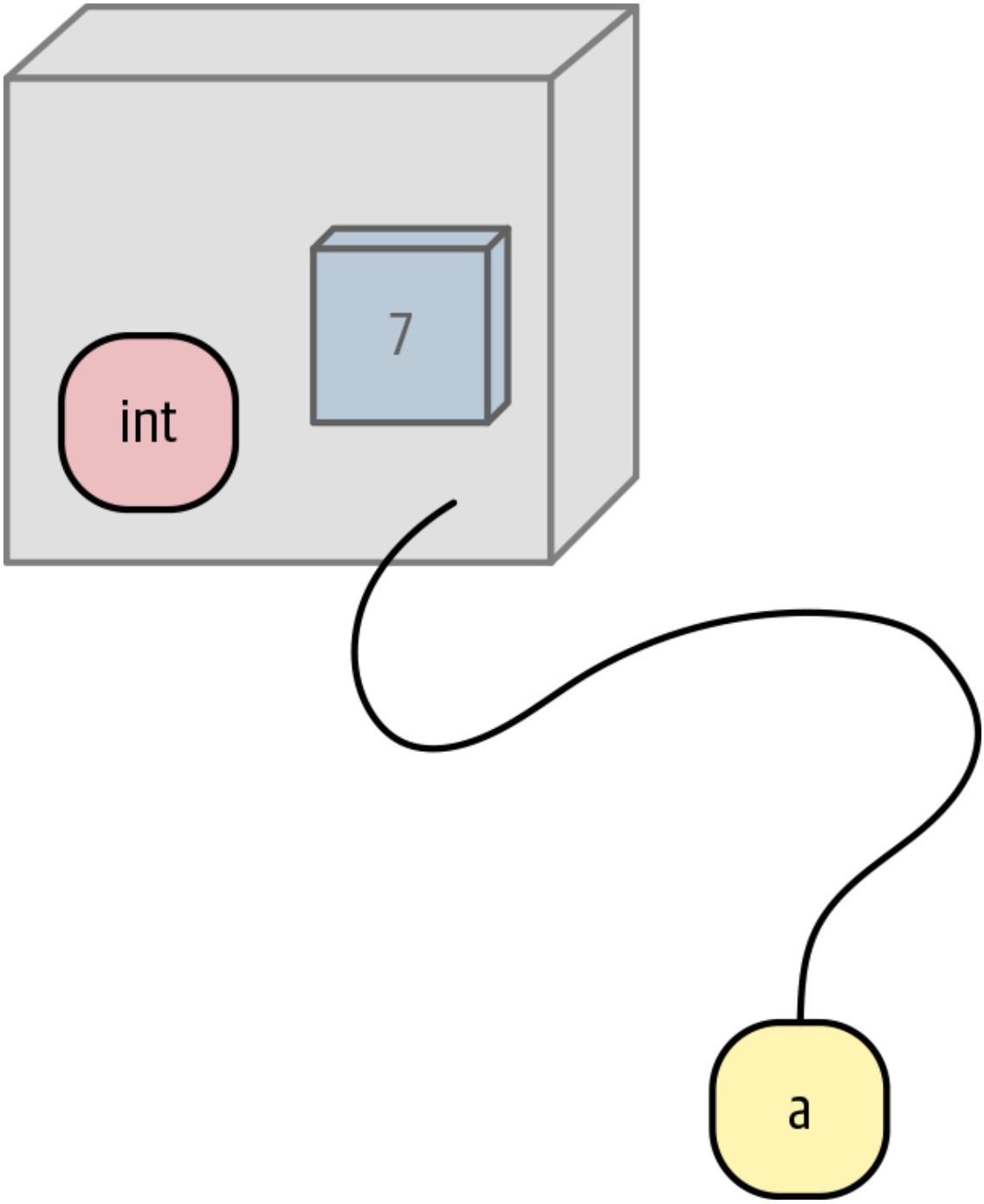

Figure 2-3. Names point to objects (variable `a` points to an integer object with value `7`)

In other languages, the variable itself has a type, and binds to a memory location. You can change the value at that location, but it needs to be of the same type. This is why *static* languages make you declare the types of variables. Python doesn't, because a name can refer to anything, and we get the value and type by "following the string" to the data object itself. This saves time, but there are some downsides:

- You may misspell a variable and get an exception because it doesn't refer to anything, and Python doesn't automatically check this as static languages do. Chapter 19 shows ways of checking your variables

ahead of time to avoid this.
- Python's raw speed is slower than a language like C. It makes the computer do more work so you don't have to.

Try this with the interactive interpreter (visualized in Figure 2-4):

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 point to the object box containing `7`.
4. Print the value of `b`:

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

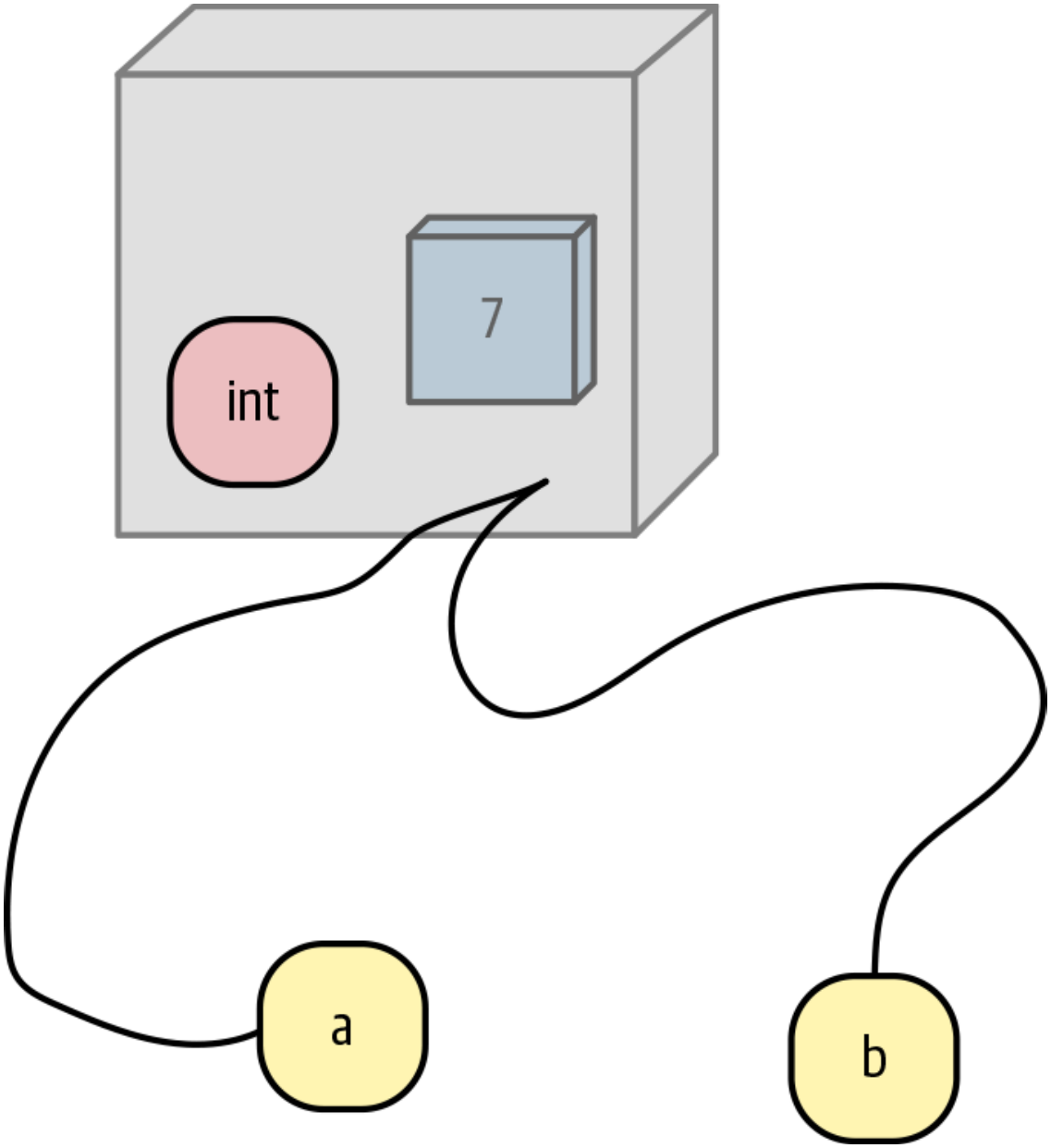

Figure 2-4. Copying a name (now variable `b` also points to the same integer object)

In Python, if you want to know the type of anything (a variable or a literal value), you can use `type(` *`thing`* `)`. `type()` is one of Python's built-in functions. If you want to check whether a variable points to an object of a specific type, use `isinstance(` *`type`* `)`:

```
>>> type(7)
<class 'int'>
>>> type(7) == int
True
>>> isinstance(7, int)
True
```

NOTE

When I mention a function, I'll put parentheses (`()`) after it to emphasize that it's a function rather than a variable name or something else.

Let's try it with more literal values (`58` , `99.9` , `'abc'`) and variables (`a` , `b`):

```
>>> a = 7
>>> b = a
>>> 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 10 covers classes in greater detail. In Python, "class" and "type" mean pretty much the same thing.

As you've seen, when you use a variable in Python, it looks up the object that it refers to. Behind the scenes, Python is busy, often creating temporary objects that will be discarded a line or two later.

Let's repeat an earlier example:

```
>>> y = 5
>>> x = 12 - y
>>> x
7
```

In this code snippet, Python did the following:

- Created an integer object with the value `5`

- Made a variable `y` point to that `5` object
- Incremented the reference count of the object with value `5`
- Created another integer object with the value `12`
- Subtracted the value of the object that `y` points to (`5`) from the value `12` in the (anonymous) object with that value
- Assigned this value (`7`) to a new (so far, unnamed) integer object
- Made the variable `x` point to this new object
- Incremented the reference count of this new object that `x` points to
- Looked up the value of the object that `x` points to (`7`) and printed it

When an object's reference count reaches zero, no names are pointing to it, so it doesn't need to stick around. Python has a charmingly named *garbage collector* that reuses the memory of things that are no longer needed. Picture someone behind those memory shelves, yanking obsolete boxes for recycling.

In this case, we no longer need the objects with the values `5`, `12`, or `7`, or the variables `x` and `y`. The Python garbage collector may choose to send them to object heaven,[2] or keep some around for performance reasons given that small integers tend to be used a lot.

Assigning to Multiple Names

You can assign a value to more than one variable name at the same time:

```
>>> two = deux = zwei = 2
>>> two
2
>>> deux
2
>>> zwei
2
```

Reassigning a Name

Because names point to objects, changing the value assigned to a name

just makes the name point to a new object. The reference count of the old object is decremented, and the new one's is incremented.

Copying

As you saw in Figure 2-4, assigning an existing variable `a` to a new variable named `b` just makes `b` point to the same object that `a` does. If you pick up either the `a` or `b` tag and follow their strings, you'll get to the same object.

If the object is immutable (like an integer), its value can't be changed, so both names are essentially read-only. Try this:

```
>>> x = 5
>>> x
5
>>> y = x
>>> y
5
>>> x = 29
>>> x
29
>>> y
5
```

When we assigned `x` to `y`, that made the name `y` point to the integer object with value `5` that `x` was also pointing to. Changing `x` made it point to a new integer object with value `29`. It did not change the one containing `5`, which `y` still points to.

But if both names point to a *mutable* object, you can change the object's value via either name, and you'll see the changed value when you use either name. If you didn't know this first, it could surprise you.

A *list* is a mutable array of values, and Chapter 7 covers them in gruesome detail. For this example, `a` and `b` each point to a list with three integer members:

```
>>> a = [2, 4, 6]
>>> b = a
>>> a
[2, 4, 6]
>>> b
[2, 4, 6]
```

These list members (`a[0]` , `a[1]` , and `a[2]`) are themselves like names, pointing to integer objects with the values `2` , `4` , and `6` . The list object keeps its members in order.

Now change the first list element, through the name `a` , and see that `b` also changed:

```
>>> a[0] = 99
>>> a
[99, 4, 6]
>>> b
[99, 4, 6]
```

When the first list element is changed, it no longer points to the object with value `2` , but a new object with value `99` . The list is still of type `list` , but its value (the list elements and their order) is mutable.

Choose Good Variable Names

> *He said true things, but called them by wrong names.*
>
> —Elizabeth Barrett Browning

It's surprising how important it is to choose good names for your variables. In many of the code examples so far, I've been using throwaway names like `a` and `x` . In real programs, you'll have many more variables to keep track of at once, and you'll need to balance brevity and clarity. For example, it's faster to type `num_loons` rather than `number _of_loons` or `gaviidae_inventory` , but it's more explanatory than `n` .

Coming Up

Numbers! They're as exciting as you expect. Well, maybe not that bad.[3] You'll see how to use Python as a

cal-
cu-
la-
tor
and
how
a
cat
founded
a
dig-
i-
tal
sys-
tem.

T h i n g s t o D o

2.1
Assign
the
in-

te-
ger
value
`9`
`9`
to
the
vari-
able
`p`
`r`
`i`
`n`
`c`
`e`,
and
print
it.

2.2
What
type
is
the
value
`5`?

2.3
What
type
is
the
value
`2`
`.`
`0`?

2.4 What type is the expression `5 + 2.0`?

`async` and `await` are new in Python 3.7.

Or the Island of Misfit

Objects.

8

looks
like
a
snow-
man!