Table of Contents

1 Everything in your computer is just a sequence of bits

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.

2 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.

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.

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.

Using the memory shelves analogy, you can think of objects as variable sized boxes occupying spaces on those shelves, as shown in Figure.

Python makes these object boxes, puts them in empty spaces on the shelves, and removes them when they're no longer used.

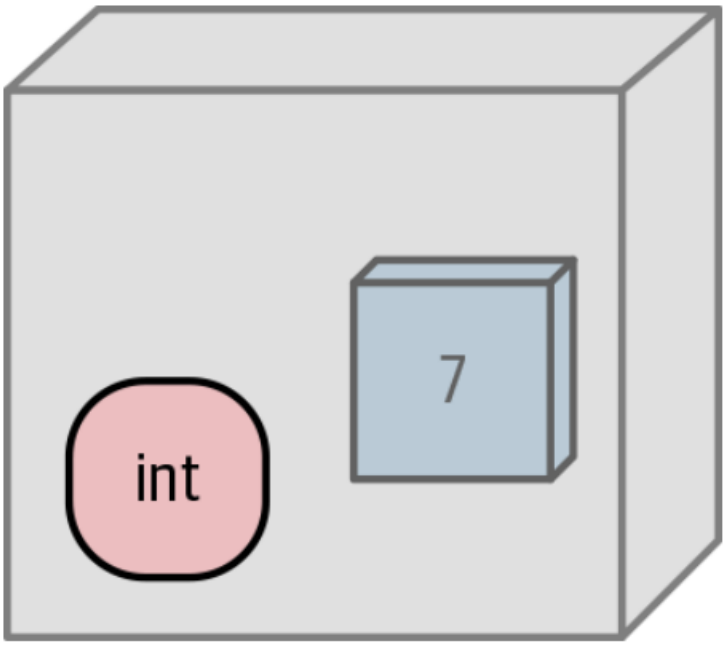


Figure 2-1. An object is like a box; this one is an integer with value 7

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

1. A type that defines what it can do (see the next section)
2. A unique id to distinguish it from other objects
3. A value consistent with its type
4. 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.

3 Type and Mutability:

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
Dictionary	`dict`	yes	`{'game': 'bingo', 'dog': 'dingo', 'drummer': 'Ringo'}`	Chapter 8

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.

3.1 Python is strongly typed, which means that the type of an object does not change, even if its value is mutable.

4 Literal Values

There are two ways of specifying data values in Python:

- Literal
- Variable

4.1 Variables:

variables: Names for values in your computer's memory that you want to use in a program.

4.2 Variable names have some rules:

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

4.3 Reserved words(keywords)

In [3]:

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

executed in 9ms, finished 21:50:28

```
Here is a list of the Python keywords.  Enter any key
word to get more help.

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

In [6]:

```
import keyword
print(keyword.kwlist)
print("Total Keywords: ",len(keyword.kwlist))
```

executed in 6ms, finished 21:51:45

```
['False', 'None', 'True', 'and', 'as', 'assert', 'asy
nc', 'await', 'break', 'class', 'continue', 'def', 'd
el', 'elif', 'else', 'except', 'finally', 'for', 'fro
m', 'global', 'if', 'import', 'in', 'is', 'lambda',
'nonlocal', 'not', 'or', 'pass', 'raise', 'return',
'try', 'while', 'with', 'yield']
Total Keywords:  35
```

- These are valid names:

1. a
2. a1
3. a_b_c___95
4. _abc
5. _1a

- These names, however, are not valid:

1. 1
2. 1a
3. 1_
4. name!
5. another-name

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.

The Assignment Operator An operator is a symbol, such as +, that performs an operation on one or more values.

- For example, the + operator takes two numbers, one to the left of the operator and one to the right, and adds them together.

Values are assigned to variable names using a special symbol called the assignment operator (=).

The = operator takes the value to the right of the operator and assigns it to the name on the left.

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.

In [7]:

```
y = x + 12
```

executed in 193ms, finished 22:00:15

```
---------------------------------------------------------
----------------------
NameError                                 Traceback
 (most recent call last)
<ipython-input-7-88c1a5748a34> in <module>
----> 1 y = x + 12

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 [8]:

```
y = 5
x = 12 - y
x
```

executed in 25ms, finished 22:00:53

Out[8]:

7

5 Variables Are Names, Not Places:

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

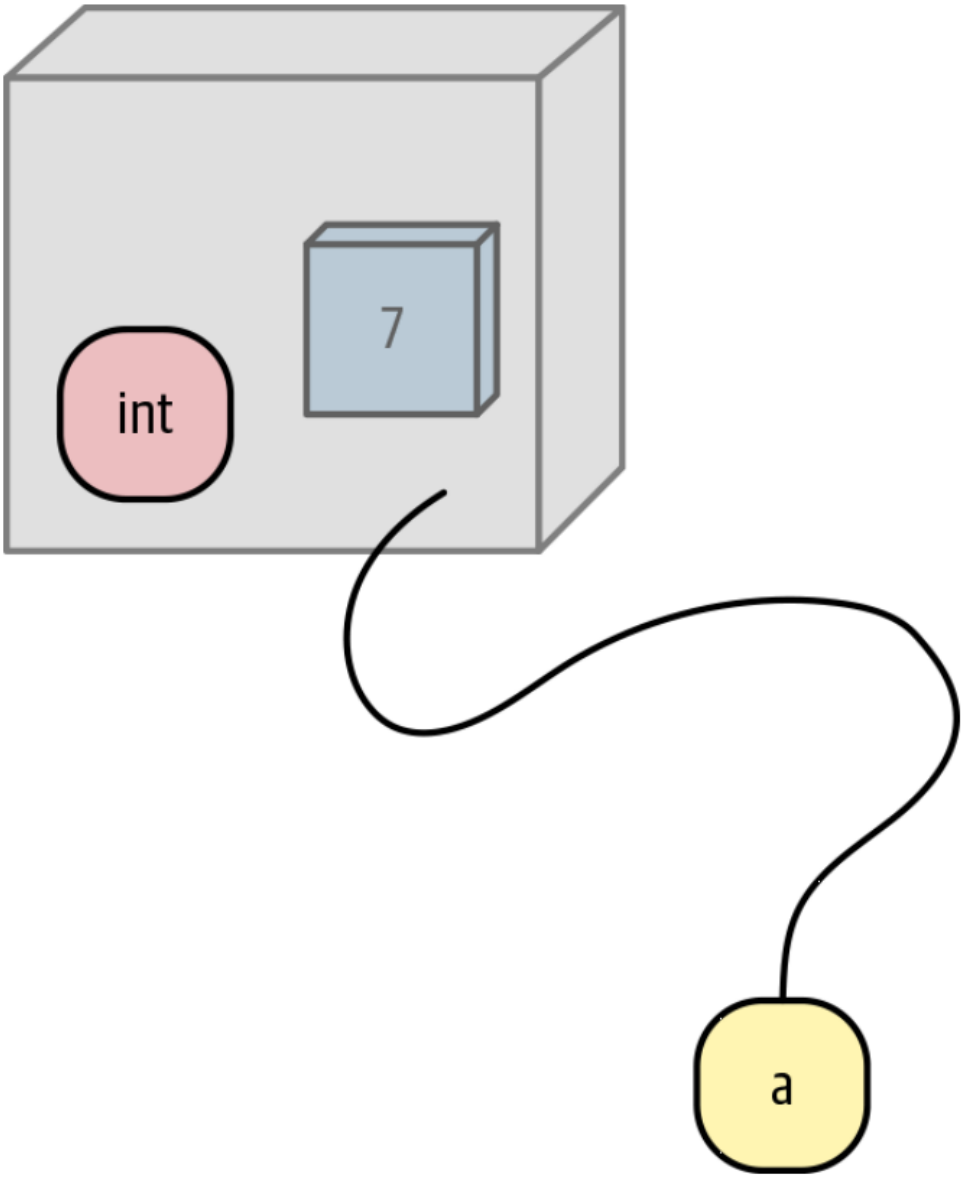


Figure 2-3. Names point to objects (variable a points to an integer object with value 7)

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

In [9]:

```
a = 5
print(a)

b = a
print(b)
```

executed in 7ms, finished 22:08:51

```
5
5
```

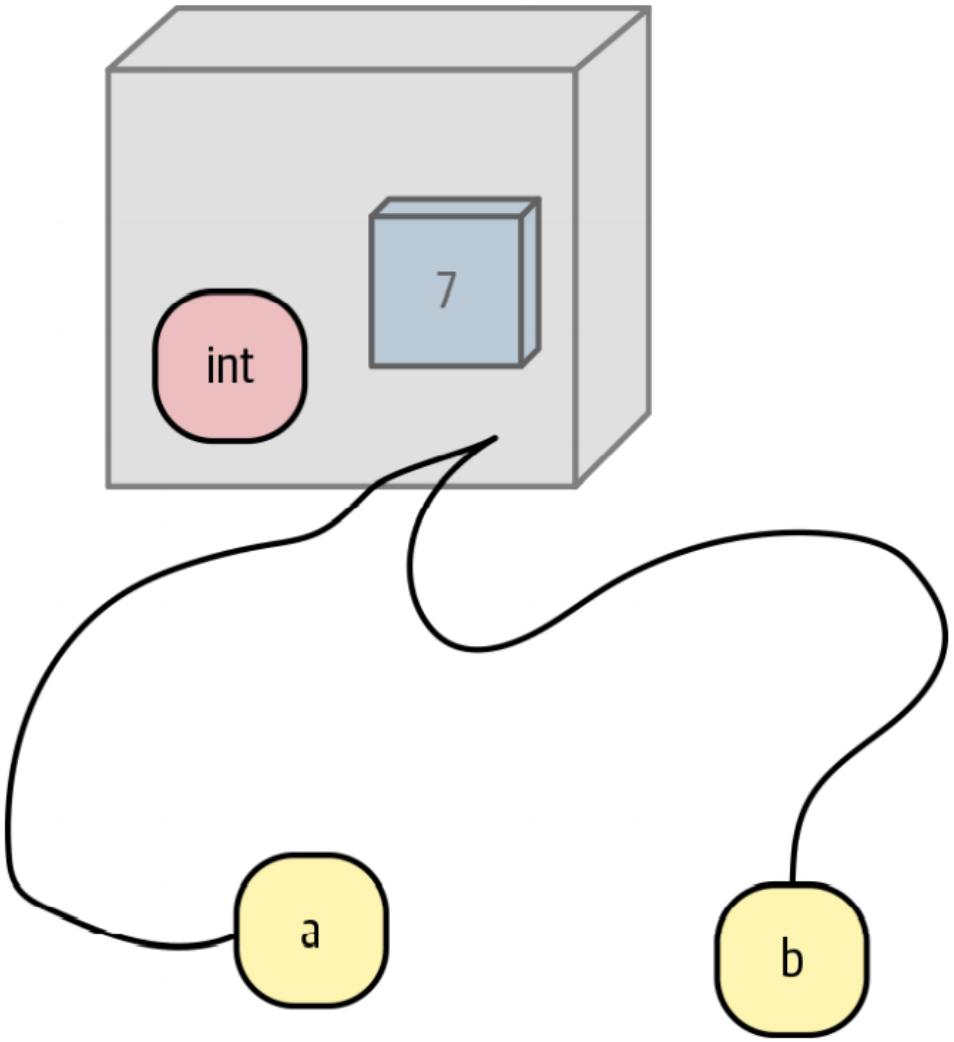


Figure 2-4. Copying a name (now variable b also points to the same integer object)

5.1 Types and instances:

type() and isinstance() is inbuilt function.

In [10]:

```
type(7)
```

executed in 22ms, finished 22:11:51

Out[10]:

```
int
```

In [11]:

```
type(7) == int
```

executed in 9ms, finished 22:12:01

Out[11]:

```
True
```

In [12]:

```
isinstance(7,int)
```

executed in 9ms, finished 22:12:10

Out[12]:

```
True
```

In [13]:

```
isinstance(7,float)
```

executed in 18ms, finished 22:12:20

Out[13]:

```
False
```

In [14]:

```
a = 7
b = a
```

executed in 14ms, finished 22:14:31

In [16]:

```
print(type(a))
print(type(b))
```

executed in 31ms, finished 22:14:56

```
<class 'int'>
<class 'int'>
```

In [18]:

```
print(type(55))
print(type(99.9))
print(type('abc'))
```

executed in 14ms, finished 22:15:41

```
<class 'int'>
<class 'float'>
<class 'str'>
```

A class is the definition of an object. In Python, `class` and `type` mean pretty much the same thing.

6 Execution of python program:

In [20]:

```
y = 5
x = 12 - y
x
```

executed in 9ms, finished 22:18:21

Out[20]:

```
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 prints 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

In [26]:

```
import sys
sys.getrefcount(y)
```

executed in 7ms, finished 22:23:38

Out[26]:

```
710
```

In [23]:

```
new_var_for_count = 13
```

executed in 28ms, finished 22:23:01

In [27]:

```
a = 10
b = a
c = b
d = c
d
```

executed in 22ms, finished 22:25:04

Out[27]:

```
10
```

In [28]:

```
import sys
sys.getrefcount(a)
```

executed in 19ms, finished 22:25:08

Out[28]:

```
466
```

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.

Reference counting

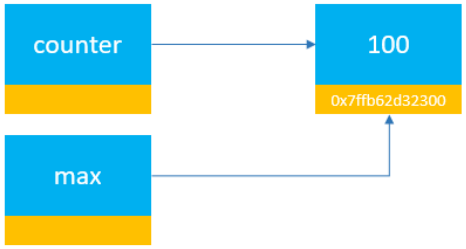
An object in the memory address can have one or more references. For example:

```
counter = 100
```

The integer object with the value of 100 has one reference which is the `counter` variable. If you assign the `counter` to another variable e.g., `max`:

```
counter = 100
max = counter
```

Now, both `counter` and `max` variables reference the same integer object. The integer object with value 100 has two references:

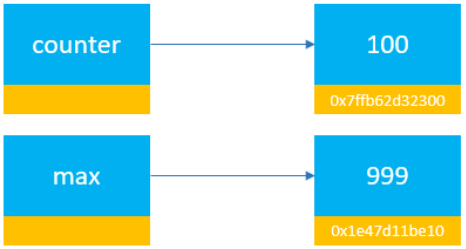


Reference	Count
0x7ffb62d32300	2

If you assign a different value to the `max` variable:

```
max = 999
```

...the integer object with value 100 has one reference, which is the `counter` variable:

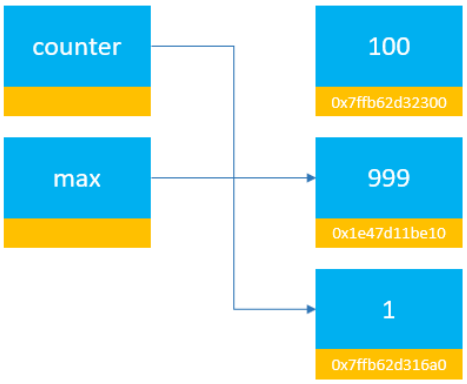


Reference	Count
0x7ffb62d32300	1
0x1e47d11be10	1

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.

And the number of references of the integer object with a value of 100 will be zero if you assign a different value to the `counter` variable:

```
counter = 1
```



Reference	Count
0x7ffb62d32300	0
0x1e47d11be10	1
0x7ffb62d316a0	1

Once an object doesn't have any reference, Python Memory Manager will destroy that object and reclaim the memory.

7 Assigning values to variables

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

In [33]:

```
a = b = c = 6
print(a)
print(b)
print(c)
```

executed in 25ms, finished 22:35:07

```
6
6
6
```

7.1 Copying

In [38]:

```
x = 5
y = x
```

executed in 20ms, finished 22:39:27

In [39]:

```
print(x)
print(y)
```

executed in 6ms, finished 22:39:27

```
5
5
```

In [40]:

```
x = 20
x
```

executed in 16ms, finished 22:39:28

Out[40]:

```
20
```

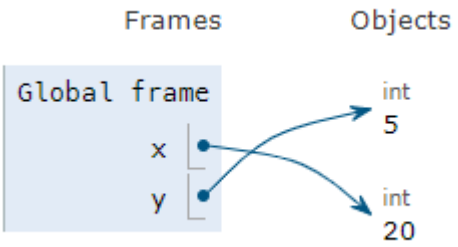
In [41]:

```
y
```

executed in 16ms, finished 22:39:28

Out[41]:

```
5
```



7.2 Case for mutable object:

A list is a mutable array of values,

In [47]:

```
a = [2,4,5]
b = a
print(a)
print(b)
```

executed in 29ms, finished 22:43:02

```
[2, 4, 5]
[2, 4, 5]
```

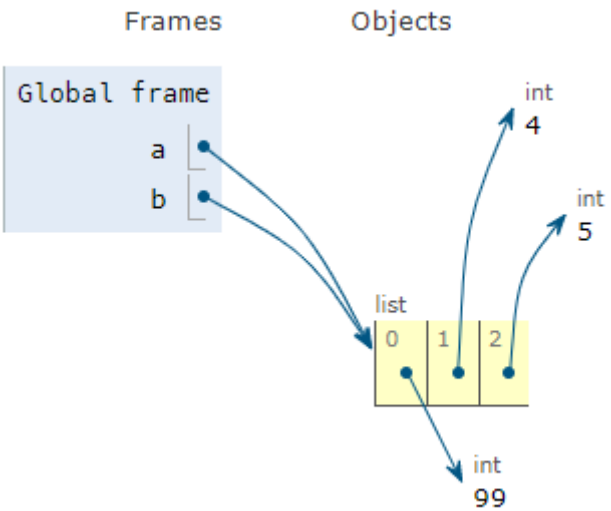
These list members (a[0], a[1], and a[2]) are themselves like names, pointing to integer objects with the values 2, 4, and 6. The list object keeps its members in order.

In [48]:

```
a[0] = 99
print(a)
print(b)
```

executed in 15ms, finished 22:43:40

```
[99, 4, 5]
[99, 4, 5]
```



In [ ]: