Chapter 3. Numbers

> *That action is best which procures the greatest happiness for the greatest numbers.*
>
> —Francis Hutcheson

In this chapter we 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`)

In a way, they're like atoms. We use them individually in this chapter, and in later chapters you'll see how to combine them into larger "molecules" like lists and dictionaries.

Each type has specific rules for its usage and is handled differently by the computer. I also show how to use *literal* values like `97` and `3.1416`, and the *variables* that I mentioned in Chapter 2.

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.

Booleans

In Python, the only values for the boolean data type are `True` and `False`. Sometimes, you'll use these directly; other times you'll evaluate the "truthiness" of other types from their values. The special Python function `bool()` can convert any Python data type to a boolean.

Functions get their own chapter in Chapter 9, but for now you just need

to know that a function has a name, zero or more comma-separated input *arguments* surrounded by parentheses, and zero or more *return values*. The `bool()` function takes any value as its argument and returns the boolean equivalent.

Nonzero numbers are considered `True`:

```
>>> bool(True)
True
>>> bool(1)
True
>>> bool(45)
True
>>> bool(-45)
True
```

And zero-valued ones are considered `False`:

```
>>> bool(False)
False
>>> bool(0)
False
>>> bool(0.0)
False
```

You'll see the usefulness of booleans in Chapter 4. In later chapters, you'll see how lists, dictionaries, and other types can be considered `True` or `False`.

Integers

Integers are whole numbers—no fractions, no decimal points, nothing fancy. Well, aside from a possible initial sign. And bases, if you want to express numbers in other ways than the usual decimal (base 10).

Literal Integers

Any sequence of digits in Python represents a *literal integer*:

```
>>> 5
```

```
5
```

A plain zero (`0`) is valid:

```
>>> 0
0
```

But you can't have an initial `0` followed by a digit between `1` and `9` :

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

NOTE

This Python *exception* warns that you typed something that breaks Python's rules. I explain what this means in "Bases". You'll see many more examples of exceptions in this book because they're Python's main error handling mechanism.

You can start an integer with `0b` , `0o` , or `0x` . See "Bases".

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't have any commas in the integer:

```
>>> 1,000,000
```

```
(1, 0, 0)
```

Instead of a million, you'd get a *tuple* (see Chapter 7 for more information on tuples) with three values. But you *can* use the underscore (_) character as a digit separator:[1]

```
>>> million = 1_000_000
>>> million
1000000
```

Actually, you can put underscores anywhere after the first digit; they're just ignored:

```
>>> 1_2_3
123
```

Integer Operations

For the next few pages, I show examples of Python acting as a simple calculator. You can do normal arithmetic with Python by using the 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`

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
```

Note that you're not required to have a space between each number and operator:

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

It just looks better stylewise and is easier to read.

Multiplication is also straightforward:

```
>>> 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 (*floats* are coming later in this chapter):

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

Truncating integer division returns an integer answer, throwing away any remainder:

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

Instead of tearing a hole in the space-time continuum, 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
```

Integers and Variables

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
```

You'll remember from Chapter 2 that `a` is a name that points to an integer object. When I said `a - 3`, I didn't assign the result back 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
```

Again, this would not be a legal math equation, but it's how you reassign a value to a variable in Python. In Python, the expression on the right side of the `=` is calculated first, and 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 = 92
>>> a += 8
>>> a
100
```

And this is like `a = a * 2`:

```
>>> a = 100
>>> a *= 2
>>> a
200
```

Here's a floating-point division example, like `a = a / 3`:

```
>>> a = 200
>>> a /= 3
>>> a
66.66666666666667
```

Now let's try the shorthand for `a = a // 4` (truncating integer division):

```
>>> a = 13
>>> a //= 4
>>> a
3
```

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 *tuple*. As I mentioned earlier, tuples will take a bow in Chapter 7; functions debut in Chapter 9.

One last math feature is exponentiation with `**`, which also lets you mix integers and floats:

```
>>> 2**3
8
>>> 2.0 ** 3
8.0
>>> 2 ** 3.0
8.0
>>> 0 ** 3
0
```

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 E 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 example with exponents

```
>>> -5 ** 2
-25
```

is the same as

```
>>> - (5 ** 2)
-25
```

and probably not what you wanted. Parentheses make it clear:

```
>>> (-5) ** 2
25
```

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 and 10 toes, so we count `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`. Next, we run out of single 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." Unlike Roman numerals, Arabic numbers don't have a single character that represents "10" 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`. This is the famous *bit*. `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 with these integer prefixes:

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

These bases are all powers of two, and are handy in some cases, although you may never need to use anything other than good old decimal integers.

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) `0b10`, which means *1 (decimal) two and 0 ones*:

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

Octal (base 8) `0o10` stands for *1 (decimal) eight and 0 ones*:

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

Hexadecimal (base 16) `0x10` means *1 (decimal) sixteen and 0 ones*:

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

You can go the other direction, converting an integer to a string with any of these bases:

```
>>> value = 65
>>> bin(value)
'0b1000001'
>>> oct(value)
```

```
'0o101'
>>> hex(value)
'0x41'
```

The `chr()` function converts an integer to its single-character string equivalent:

```
>>> chr(65)
'A'
```

And `ord()` goes the other way:

```
>>> ord('A')
65
```

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 different bases from 10? They're useful in *bit-level* operations, which are described in Chapter 12, along with more details about converting numbers from one base to another.

Cats normally have five digits on each forepaw and four on each hindpaw, for a total of 18. If you ever encounter cat scientists in their lab coats, they're often discussing base-18 arithmetic. My cat Chester, seen lounging about in Figure 3-1, is a *polydactyl*, giving him a total of 22 or so (they're hard to distinguish) toes. If he wanted to use all of them to count food fragments surrounding his bowl, he would likely use a base-22 system (hereafter, the *chesterdigital* system), using `0` through `9` and `a` through `l`.

Figure 3-1. Chester—a fine furry fellow, and inventor of the chesterdigital system

Type Conversions

To change other Python data types to an integer, use the `int()` function.

The `int()` function takes one input argument and returns one value, the integer-ized equivalent of the input argument. This will keep the whole number and discard any fractional part.

As you saw at the start of this chapter, 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
```

Turning this around, the `bool()` function returns the boolean equivalent

of an integer:

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

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

```
>>> int(98.6)
98
>>> int(1.0e4)
10000
```

Converting a float to a boolean is no surprise:

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

Finally, here's an example of getting the integer value from a text string (Chapter 5) that contains only digits, possibly with `_` digit separators or an initial `+` or `-` sign:

```
>>> int('99')
99
>>> int('-23')
-23
>>> int('+12')
12
>>> int('1_000_000')
1000000
```

If the string represents a nondecimal integer, you can include the base:

```
>>> int('10', 2) # binary
2
>>> int('10', 8) # octal
8
```

```
>>> int('10', 16) # hexadecimal
16
>>> int('10', 22) # chesterdigital
22
```

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 wa
>>> 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.

`int()` will make integers from floats or strings of digits, but it 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` could be limited to 32 or 64 bits, depending on your CPU; 32 bits can store store any integer from –2,147,483,648 to 2,147,483,647.

A `long` had 64 bits, allowing values from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. In Python 3, the `long` type is long gone, and an `int` can be *any* size—even greater than 64 bits. You can play with big numbers like a *googol* (one followed by a hundred zeroes, named in 1920 by a nine-year-old boy):

```
>>>
>>> googol = 10**100
>>> googol
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000
>>> googol * googol
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
```

A *googolplex* is `10**googol` (a thousand zeroes, if you want to try it yourself). This was a suggested name for Google before they decided on *googol*, but didn't check its spelling before registering the domain name `google.com`.

In many languages, trying this would cause something called *integer overflow*, where the number would need more space than the computer al-

lowed for it, with various bad effects. Python handles googoly integers with no problem.

Floats

Integers are whole numbers, but *floating-point* numbers (called *floats* in Python) have decimal points:

```
>>> 5.
5.0
>>> 5.0
5.0
>>> 05.0
5.0
```

Floats can include a decimal integer exponent after the letter `e`:

```
>>> 5e0
5.0
>>> 5e1
50.0
>>> 5.0e1
50.0
>>> 5.0 * (10 ** 1)
50.0
```

You can use underscore (`_`) to separate digits for clarity, as you can for integers:

```
>>> million = 1_000_000.0
>>> million
1000000.0
>>> 1.0_0_1
1.001
```

Floats are handled similarly to integers: you can use the operators (`+`, `-`, `*`, `/`, `//`, `**`, and `%`) and the `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
```

When you mix integers and floats, Python automatically *promotes* the integer values to float values:

```
>>> 43 + 2.
45.0
```

Python also promotes booleans to integers or floats:

```
>>> False + 0
0
>>> False + 0.
0.0
>>> True + 0
1
>>> True + 0.
1.0
```

Math Functions

Python supports complex numbers and has the usual math functions such as square roots, cosines, and so on. Let's save them for Chapter 22, in which we also discuss using Python in science contexts.

Coming Up

In the next chapter, you finally graduate from one-line Python examples. With the `if` statement, you'll learn how to make decisions with code.

Things to Do

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

3.1 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`).

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

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

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

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

3.6 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` ?

For
Python
3.6
and
newer.