Chapter 5
Numbers, Booleans and None

5.1 Introduction

In this chapter we will explore the different ways that numbers can be represented by the *built-in* types in Python. We will also introduce the Boolean type used to represent `True` and `False`. As part of this discussion we will also look at both numeric and assignment operators in Python. We will conclude by introducing the special value known as `None`.

5.2 Types of Numbers

There are three types used to represent numbers in Python; these are integers (or integral) types, floating point numbers and complex numbers.

This begs the question why? Why have different ways of representing numbers; after all humans can easily work with the number 4 and the number 4.0 and don't need completely different approaches to writing them (apart from the '.' of course).

This actually comes down to efficiency in terms of both the amount of memory needed to represent a number and the amount of processing power needed to work with that number. In essence integers are simpler to work with and can take up less memory than real numbers. Integers are whole numbers that do not need to have a fractional element. When two integers are added, multiplied or subtracted they will always generate another integer number.

In Python real numbers are represented as floating point numbers (or floats). These can contain a fractional part (the bit after the decimal point). Computers can best work with integers (actually of course only really 1s and 0s). They therefore need a way to represent a floating point or real number. Typically this involves representing the digits before and after the decimal point.

J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_5

The term floating point is derived from the fact that there is no fixed number of digits before or after the decimal point; that is, the decimal point can float.

Operations on floating point numbers such as addition, subtract, multiplication etc. will generate new real numbers which must also be represented. It is also much harder to ensure that the results are correct as potentially very small and very large fractional parts may be involved. Indeed, most floating-point numbers are actually represented as approximations. This means that one of the challenges in handling floating-point numbers is in ensuring that the approximations lead to reasonable results. If this is not done appropriately, small discrepancies in the approximations can snowball to the point where the final results become meaningless.

As a result, most computer programming languages treat integers such as 4 as being different from real numbers such as 4.000000004.

Complex numbers are an extension of real numbers in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of −1), where the imaginary part is often written in mathematics using an 'i' while in engineering it is often written using a 'j'.

Python has built-in support for complex numbers, which are written using the engineering notation; that is the imaginary part is written with a j suffix, e.g. 3 + 1j.

5.3 Integers

All integer values, no matter how big or small are represented by the integral (or int) type in Python 3. For example:

```
x = 1
print(x)
print(type(x))
x =
10000000000000000000000000000000000000000000000000000000000000001
print(x)
print(type(x))
```

If this code is run then the output will show that both numbers are of type int:

```
1
<class 'int'>
10000000000000000000000000000000000000000000000000000000000000001
<class 'int'>
```

This makes it very easy to work with integer numbers in Python. Unlike some programming languages such as C# and Java have different integer types depending on the size of the number, small numbers having to be converted into larger types in some situations.

5.3.1 Converting to Ints

It is possible to convert another type into an integer using the `int()` function. For example, if we want to convert a string into an int (assuming the string contains a integer number) then we can do this using the `int()` function. For example

```
total = int('100')
```

This can be useful when used with the `input()` function.

The `input()` function always returns a string. If we want to ask the user to input an integer number, then we will need to convert the string returned from the `input()` function into an `int`. We can do this by wrapping the call to the `input()` function in a call to the `int()` function, for example:

```
age = int(input('Please enter your age:'))
print(type(age))
print(age)
```

Running this gives:

```
Please enter your age: 21
<class 'int'>
21
```

The `int()` function can also be used to convert a floating point number into an `int`, for example:

```
i = int(1.0)
```

5.4 Floating Point Numbers

Real numbers, or floating point numbers, are represented in Python using the IEEE 754 double-precision binary floating-point number format; for the most part you do not need to know this but it is something you can look up and read about if you wish.

The type used to represent a floating point number is called `float`.

Python represents floating point numbers using a decimal point to separate the whole part from the fractional part of the number, for example:

```
exchange_rate = 1.83
print(exchange_rate)
print(type(exchange_rate))
```

This produces output indicating that we are storing the number 1.83 as a floating point number:

```
1.83
<class 'float'>
```

5.4.1 Converting to Floats

As with integers it is possible to convert other types such as an int or a string into a float. This is done using the `float()` function:

```
int_value = 1
string_value = '1.5'
float_value = float(int_value)
print('int value as a float:', float_value)
print(type(float_value))
float_value = float(string_value)
print('string value as a float:', float_value)
print(type(float_value))
```

The output from this code snippet is:

```
int value as a float: 1.0
<class 'float'>
string value as a float: 1.5
<class 'float'>
```

5.4.2 Converting an Input String into a Floating Point Number

As we have seen the `input()` function returns a string; what happens if we want the user to input a floating point (or real) number? As we have seen above, a string can be converted into a floating point number using the `float()` function and therefore we can use this approach to convert an input from the user into a float:

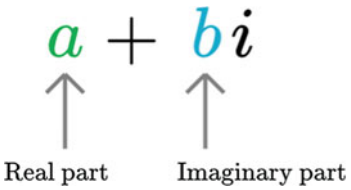
```
exchange_rate = float(input("Please enter the exchange rate to
use: "))
print(exchange_rate)
print(type(exchange_rate))
```

Using this we can input the string `1.83` and convert it to a floating-point number:

```
Please enter the exchange rate to use: 1.83
1.83
<class 'float'>
```

5.5 Complex Numbers

Complex numbers are Pythons third type of built-in numeric type. A complex number is defined by a real part and an imaginary part and has the form a + b*i* (where *i* is the imaginary part and a and b are real numbers):



The real part of the number (a) is the real number that is being added to the pure imaginary number.

The imaginary part of the number, or b, is the real number coefficient of the pure imaginary number.

The letter 'j' is used in Python to represent the imaginary part of the number, for example:

```
c1 = 1j
c2 = 2j
print('c1:', c1, ', c2:', c2)
print(type(c1))
print(c1.real)
print(c1.imag)
```

We can run this code and the output will be:

```
c1: 1j , c2: 2j
<class 'complex'>
0.0
1.0
```

As you can see the type of the number is 'complex' and when the number is printed directly it is done so by printing both the real and imaginary parts together.

Don't worry if this is confusing; it is unlikely that you will need to use complex numbers unless you are doing some very specific coding, for example within a scientific field.

5.6 Boolean Values

Python supports another type called Boolean; a Boolean type can only be one of `True` or `False` (and nothing else). Note that these values are `True` (with a capital T) and `False` (with a capital F); true and false in Python are not the same thing and have no meaning on their own.

The equivalent of the `int` or `float` class for Booleans is `bool`.

The following example illustrates storing the two Boolean values into a variable `all_ok`:

```
all_ok = True
print(all_ok)
all_ok = False
print(all_ok)
print(type(all_ok))
```

The output of this is

```
True
False
<class 'bool'>
```

The Boolean type is actually a sub type of integer (but with only the values `True` and `False`) so it is easy to translate between the two, using the functions `int()` and `bool()` to convert from Booleans to Integers and vice versa. For example:

```
print(int(True))
print(int(False))
print(bool(1))
print(bool(0))
```

Which produces

```
1
0
True
False
```

You can also convert strings into Booleans as long as the strings contain either `True` or `False` (and nothing else). For example:

```
status = bool(input('OK to proceed: '))
print(status)
print(type(status))
```

When we run this

```
OK to proceed: True
True
<class 'bool'>
```

5.7 Arithmetic Operators

Arithmetic operators are used to perform some form of mathematical operation such as addition, subtraction, multiplication and division etc. In Python they are represented by one or two characters. The following table summarises the Python arithmetic operators:

Operator	Description	Example
+	Add the left and right values together	1 + 2
−	Subtract the right value from the left value	3 − 2
*	Multiple the left and right values	3 * 4
/	Divide the left value by the right value	12/3
//	Integer division (ignore any remainder)	12//3
%	Modulus (aka the remainder operator)—only return any remainder	13%3
**	Exponent (or power of) operator—with the left value raised to the power of the right	3 ** 4

5.7.1 Integer Operations

Two integers can be added together using +, for example `10 + 5`. In turn two integers can be subtracted (`10 − 5`) and multiplied (`10 * 4`). Operations such as +, − and * between integers always produce integer results.

This is illustrated below:

```
home = 10
away = 15
print(home + away)
print(type(home + away))

print(10 * 4)
print(type(10*4))

goals_for = 10
goals_against = 7
print(goals_for - goals_against)
print(type(goals_for - goals_against))
```

The output from this is

```
25
<class 'int'>
40
<class 'int'>
3
<class 'int'>
```

However, you may notice that we have missed out division with respect to integers, why is this? It is because it depends on which division operator you use as to what the returned type actually is.

For example, if we divide the integer 100 by 20 then the result you might reasonably expect to produce might be 5; but it is not, it is actually 5.0:

```
print(100 / 20)
print(type(100 / 20))
```

The output is

```
5.0
<class 'float'>
```

And as you can see from this the type of the result is `float` (that is a floating point number). So why is this the case?

The answer is that division does not know whether the two integers involved divide into one another exactly or not (i.e. is there a remainder). It therefore defaults to producing a floating point (or real) number which can have a fractional part. This is of course necessary in some situations, for example if we divide 3 by 2:

```
res1 = 3/2
print(res1)
print(type(res1))
```

In this case 3 cannot be exactly divided by 2, we might say that 2 goes into 3 once with a remainder. This is what is shown by Python:

```
1.5
<class 'float'>
```

The result is that 2 goes into 3, 1.5 times with the type of the result being a `float`.

If you are only interested in the number of times 2 does go into 3 and are happy to ignore the fractional part then there is an alternative version of the divide operator `//`. This operator is referred to as the integer division operator:

```
res1 = 3//2
print(res1)
print(type(res1))
```

which produces

```
1
<class 'int'>
```

But what if you are only interested in the remainder part of a division, the integer division operator has lost that? Well in that case you can use the modulus operation ('`%`'). This operator returns the remainder of a division operation: for example:

```
print('Modulus division 4 % 2:', 4 % 2)
print('Modulus division 3 % 2:', 3 % 2)
```

Which produces:

```
Modulus division 4 % 2: 0
Modulus division 3 % 2: 1
```

A final integer operator we will look at is the power operator that can be used to raise an integer by a given power, for example 5 to the power of 3. The power operator is '**', this is illustrated below:

```
a = 5
b = 3
print(a ** b)
```

Which generates the number 125.

5.7.2 Negative Number Integer Division

It is also worth just exploring what happens in integer and true division when negative numbers are involved. For example,

```
print('True division 3/2:', 3 / 2)
print('True division 3//2:', -3 / 2)
print('Integer division 3//2:', 3 // 2)
print('Integer division 3//2:', -3 // 2)
```

The output from this is:

```
True division 3/2: 1.5
True division 3//2: -1.5
Integer division 3//2: 1
Integer division 3//2: -2
```

The first three of these might be exactly what you expect given our earlier discussion; however, the output of the last example may seem a bit surprising, why does `3//2` generate `1` but `−3//2` generates −2?

The answer is that Python always rounds the result of *integer* division towards minus infinity (which is the smallest negative number possible). This means it pulls the result of the integer division to the smallest possible number, 1 is smaller than 1.5 but −2 is smaller than −1.5.

5.7.3 Floating Point Number Operators

We also have the multiple, subtract, add and divide operations available for floating point numbers. All of these operators produce new floating point numbers:

```
print(2.3 + 1.5)
print(1.5 / 2.3)
print(1.5 * 2.3)
print(2.3 - 1.5)
print(1.5 - 2.3)
```

These statements produce the output given below:

```
3.8
0.6521739130434783
3.4499999999999997
0.7999999999999998
-0.7999999999999998
```

5.7.4 Integers and Floating Point Operations

Any operation that involves both integers and floating point numbers will always produce a floating point number. That is, if one of the sides of an operation such as add, subtract, divide or multiple is a floating point number then the result will be a floating point number. For example, given the integer 3 and the floating point number 0.1, if we multiple them together then we get a floating point number:

```
i = 3 * 0.1
print(i)
```

Executing this we get

```
0.30000000000000004
```

Which may or may not have been what you expected (you might have expected 0.3); however this highlights the comment at the start of this chapter relating to floating point (or real) numbers being represented as an approximation within a computer system. If this was part of a larger calculation (such as the calculation of the amount of interest to be paid on a very large loan over a 10 year period) then the end result might well be out by a significant amount.

It is possible to overcome this issue using one of Pythons modules (or libraries). For example, the `decimal` module provides the Decimal class that will appropriately handle multiplying 3 and 0.1.

5.7.5 *Complex Number Operators*

Of course you can use operators such as multiply, add, subtract and divide with complex numbers. For example:

```
c1 = 1j
c2 = 2j
c3 = c1 * c2
print(c3)
```

We can run this code and the output will be:

```
(-2+0j)
```

You can also convert another number or a string into a complex number using the `complex()` function. For example:

```
complex(1) # generates (1+0j)
```

In addition the `math` module provides mathematical functions for complex numbers.

5.8 Assignment Operators

In Chap. 3 we briefly introduced the assignment operator ('=') which was used to assign a value to a variable. There are in fact several different assignment operators that could be used with numeric values.

These assignment operators are actually referred to as *compound operators* as they combine together a numeric operation (such as add) with the assignment operator. For example, the += compound operator is a combination of the add operator and the = operator such that

```
x = 0
x += 1   # has the same behaviour as x = x + 1
```

Some developers like to use these compound operators as they are more concise to write and can be interpreted more efficiently by the Python interpreter.

The following table provides a list of the available compound operators

Operator	Description	Example	Equivalent
+=	Add the value to the left-hand variable	x += 2	x = x + 2
−=	Subtract the value from the left-hand variable	x −= 2	x = x − 2
*=	Multiple the left-hand variable by the value	x *= 2	x = x * 2
/=	Divide the variable value by the right-hand value	x /= 2	x = x/2
//=	Use integer division to divide the variable's value by the right-hand value	x //= 2	x = x//2
%=	Use the modulus (remainder) operator to apply the right-hand value to the variable	x %= 2	x = x % 2
**=	Apply the power of operator to raise the variable's value by the value supplied	x **= 3	x = x ** 3

5.9 None Value

Python has a special type, the NoneType, with a single value, None.

This is used to represent null values or *nothingness*.

It is not the same as False, or an empty string or 0; it is a *non-value*. It can be used when you need to create a variable but don't have an initial value for it. For example:

```
winner = None
```

You can then test for the presence of None using 'is' and 'is not', for example:

```
print(winner is None)
```

This will print out True if and only if the variable winner is currently set to None.

Alternatively you can also write:

```
print(winner is not None)
```

Which will print out True only if the value of winner is *not* None.

Several example using the value None and the 'is' and is not' operators are given below:

```
winner = None
print('winner:', winner)
print('winner is None:', winner is None)
print('winner is not None:', winner is not None)
print(type(winner))
print('Set winner to True')
winner = True
print('winner:', winner)
print('winner is None:', winner is None)
print('winner is not None:', winner is not None)
print(type(winner))
```

The output of this code snippet is:

```
winner: None
winner is None: True
winner is not None: False
<class 'NoneType'>
Set winner to True
winner: True
winner is None: False
winner is not None: True
<class 'bool'>
```

5.10 Online Resources

See the Python Standard Library documentation for:

- https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex Numeric Types.
- https://docs.python.org/3/library/stdtypes.html#truth-value-testing Boolean values and boolean testing.
- https://docs.python.org/3/library/decimal.html which provides information on the Python decimal module.
- https://docs.python.org/3/library/cmath.html which discusses mathematical functions for complex numbers.

If you are interested in how floating point numbers are represented then a good starting points are:

- https://en.wikipedia.org/wiki/Double-precision_floating-point_format which provides an overview of floating point representation.
- https://en.wikipedia.org/wiki/IEEE_754 which is the Wikipedia page on the IEEE 754 Double-precision floating-point number format.

5.11 Exercises

The aim of the exercises in this chapter is to explore the numeric types we have been looking at.

5.11.1 General Exercise

Try to explore the different number types available in Python.

You should try out the different numeric operators available and mix up the numbers being used, for example, 1 and well as 1.0 etc.

Check to see the results you get are what you expect.

5.11.2 Convert Kilometres to Miles

The aim of this exercise is to write a program to convert a distance in Kilometres into a distance in miles.

1. Take input from the user for a given distance in Kilometres. This can be done using the `input()` function.
2. Convert the value returned by the `input()` function from a string into an integer using the `int()` function.
3. Now convert this value into miles—this can be done by dividing the kilometres by 0.6214
4. Print out a message telling the user what the kilometres are in miles.