Chapter 2

Python Basics

2.1 Exponentiation

In Python, `2` raised to the power `3` is denoted `2**3`, *not* `2^3`.

Exercise 2.1

Compute `2**3` and `2^3`. (Include `print()` if needed.)

The caret symbol ^ is a "bitwise XOR operator." You don't need to understand this right now. Just remember that exponentiation uses the `**` notation.

2.2 Order of operations and parentheses

We can string together calculations—for example,

```
3*4+5**2
```

yields the result `37`. Python follows the standard order of operations: powers come first, then multiplication and division, then addition and subtraction. It is often wise to use parentheses to make sure your expressions are interpreted in the way that you intend. For example, the above is equivalent to

```
(3*4)+(5**2)
```

Parentheses can help make your expressions easier to read.

Exercise 2.2

Use Python to compute `(4*2)+(3*5)` and `4*(2+3)*5`. Are the answers correct? What result does Python give for `1+5*3**6/2`? Express this statement more clearly using one or more sets of parentheses, and check your results.

2.3 Integer division

The sum, difference, or product of two integers is always an integer. For example, $7+4$, $7-4$, and $7\cdot 4$ yield the integers 11, 3 and 28, respectively. The same is not true for division. We expect the result of 7/4 to be the real number 1.75.

The way computers store numbers into memory depends on whether the number is real (such as 7.0 or 4.0) or integer (such as 7 or 4). As a result, in many programming languages, the character `/` between two integers will return an integer. For example, `7/4` might evaluate to `1`, which is the integer part of 1.75. Of course, in any programming language, real number division should work as expected with `7.0/4.0` giving `1.75`.

Older versions of Python (before version 3.0) treated integer division in this way, with `7/4` evaluating to `1`. Most likely you are using a newer version of Python that treats integer division the way you expect, with `7/4` yielding `1.75`.

The concept of "integer division," where the quotient of integers n and m is defined as the integer part of n/m, can be useful. Newer versions of Python define such an operation by `//`. The related operator `%` gives the remainder.

Exercise 2.3

Evaluate the expressions `7/4`, `7//4` and `7%4`. What are the results? Replace the integers `7` and `4` with real numbers `7.0` and `4.0`. Do the results change?

2.4 NumPy

The core Python language is somewhat limited, but its vocabulary can be extended by the addition of *libraries*. For scientific computing we often use a library called NumPy (short for "numerical Python"). NumPy can be imported with this command:

```
import numpy
```

NumPy defines a number of constants; for example, π and e are given by

```
numpy.pi
numpy.e
```

NumPy also defines common mathematical functions such as the square root, trig functions, exponentials and logarithms.

Exercise 2.4

Import NumPy and evaluate the following statements.

```
numpy.sqrt(2.0)
numpy.sqrt(25.0)
numpy.sin(30)
numpy.sin(numpy.pi)
numpy.cos(45)
numpy.cos(numpy.pi/2)
numpy.exp(1.0)
numpy.exp(3.0)
numpy.log(numpy.e)
numpy.log(numpy.e*numpy.e)
numpy.log10(10)
numpy.log10(100)
```

- Do NumPy trig functions use radians or degrees?
- What is the base of the `numpy.log()` function?
- What is the base of `numpy.log10()`?

An alternative to `numpy.exp(2.0)` is `numpy.e**2.0`. Similarly, we could write `2.0**0.5` in place of `numpy.sqrt(2.0)`. Most experienced programmers avoid the `**` notation in these contexts and use the NumPy functions `exp()` and `sqrt()` instead.

2.5 Strings

A character is anything that you can type on a keyboard. A *string* is a sequence of one or more characters. Strings (or text) are always surrounded by single or double quotation marks. For example, `'cat'` and `"tree"` are strings.

Observe that `"17.3"` is a string, whereas `17.3` is a number. Python interprets `"17.3"` as a sequence of characters, like any other string. Python interprets `17.3` as a real number.

Strings can be used with the `print()` function; for example,

```
print("Did the cat climb the tree?")
```

Strings are often used to explain the output of a code, as in this example:

```
print("ln(10) = ")
print(numpy.log(10))
```

Exercise 2.5

Run the code above. Modify your code to combine the two print statements into one:

```
print("ln(10) = ", numpy.log(10))
```

2.6 Variables and the = operator

In Python, as in most computer languages, the character `=` is an *assignment operator*. It tells the computer to evaluate whatever is on the right–hand side then assign the result to the *variable* named on the left–hand side. For example,

```
x = 3 + 5
print("x = ", x)
```

The first line instructs the computer to evaluate `3 + 5`, which is `8`, and to assign `8` to the variable `x`. The second line instructs the computer to print the string `"x = "` and the contents of the variable `x`.

Here is another example:

```
fruit = "banana"
print("fruit = ", fruit)
```

The first line assigns the string `"banana"` to the variable `fruit`. The second line prints the string `"fruit = "` and the contents of the variable `fruit`.

In the world of ordinary mathematics we refer to the character $=$ as an "equals sign." Consider the following statements:

```
3 = 1 + 2
x + y = 5 + 2
z*3 = 15
```

Exercise 2.6a

Execute the above in Python. What are the results?

These statements are perfectly fine in ordinary mathematics but are not allowed in Python. This is because `3`, `x + y`, and `z*3` are not variables.

Variable names in Python can only contain letters, numbers and the underscore character, and they must begin with a letter or an underscore (not a number.) Thus, `xyz_123` is a valid variable name but `123_xyz` is not. Also note that Python is case sensitive. For example, `fruit` and `Fruit` and `fruiT` are all different variable names.

Certain words have special meanings in Python, and cannot be used as variables. These *keywords* include `for`, `if`, `else`, `while`, `break`, `continue`, `and`, `or`, `not`, `return`, and others.[1]

Remember, the assignment operator `=` should have a variable on the left and an expression to evaluate on the right. The "expression to evaluate" can be a mathematical expression like `3 + 5`. It can also be a single number like `8`, or a single string like `"banana"`.

As an assignment operator, the character `=` can be used in ways that would not make sense in ordinary mathematics. For example, the code

```
x = 4
x = x + 7
```

is perfectly fine in Python. The first statement assigns the number `4` to the variable `x`. The second statement evaluates `x + 7`, which is `11`; Python then assigns the result `11` to the variable `x`, replacing the previous value in the process.

Exercise 2.6b

Execute the above in Python. Insert the statement `print(x)` at various places in the code. Does `x` equal `4` or `11`?

[1]Python will let you know if you try to use a keyword as a variable name. In scientific computing the only keyword that you might be tempted to use as a variable name is `lambda`.

Exercise 2.6c

Assume that `x` has been assigned the value `13`. Which of these Python statements is valid? Invalid? Why?

- `x2x = x*2*x`
- `x = x + "banana"`
- `banana = "3 + 5"`
- `y = z = 27`
- `4 = 4`
- `y = 6 = 13`
- `apple&orange = "fruit"`

2.7 Projectile motion

A projectile near Earth follows a trajectory in the x–y plane (the x axis is horizontal and the y axis is vertical) given by

$$x(t) = x_0 + v_{0x}t \ , \qquad (2.1a)$$

$$y(t) = y_0 + v_{0y}t - \frac{1}{2}gt^2 \ . \qquad (2.1b)$$

Here, x_0, y_0 are the components of the initial position, v_{0x}, v_{0y} are the components of the initial velocity, and g is the acceleration due to gravity. Of course t is time. Let's write a Python program to calculate the position of the projectile at $t = 2.0\,\mathrm{s}$, given the initial conditions $x_0 = 0.0\,\mathrm{m}$, $y_0 = 5.0\,\mathrm{m}$, $v_{0x} = 10.0\,\mathrm{m/s}$ and $v_{0y} = 15.0\,\mathrm{m/s}$:

```
# Position of a projectile near Earth at time t

x0 = 0.0      # x-component of initial position (m)
y0 = 5.0      # y-component of initial position (m)
v0x = 10.0    # x-component of initial velocity (m/s)
v0y = 15.0    # y-component of initial velocity (m/s)
g = 9.8       # acceleration due to gravity (m/s**2)
t = 2.0       # time (s)

# Compute the position at time t
x = x0 + v0x*t
y = y0 + v0y*t - 0.5*g*t**2

# Print the results
print("Position at t = ", t, "seconds:")
print("x = ", x, "meters")
print("y = ", y, "meters")
```

This program can be created in an IDE text editor, or in one or more code cells of a Jupyter notebook.

Observe the following features of the program.

- Comments are used throughout. Any text that follows `#` is a comment. Adding comments to your code will make it easier for other people to understand. Comments will help you as well—as time goes by it can be difficult to remember the details and logic of your own codes.
- Blank lines are inserted between sections of code. We begin with a description of the code, then a section where variables are defined. The position at time t is computed in the next section, and the results are printed in the final section. Separating the code into sections makes it easier to read.
- Numbers are assigned to variables. This is useful because we might want change the numbers in the calculation. For example, we can easily change the time from `2.0` to `3.0` by modifying the single statement `t = 2.0`. There is no need to search through the code for every occurance of the number `2.0`, and then decide if it should be changed to `3.0`.
- The variable names correspond closely to the variables used in the formulas (2.1). This makes the variables easy to recognize. For example, the x–component of initial velocity is denoted v_{0x} in the formulas and `v0x` in the code. Another good name would be `v_0x`.
- The print statements include text strings that explain the output.

Exercise 2.7

Run the code to compute the position of the projectile at various times. Experiment with different values for the initial position and velocity.

2.8 User input

We can change the value of time in our projectile code by modifying the statement `t = 2.0`. Another option is to let the user choose the value of `t` when the code is executed. This can be done with the `input()` function. Simply replace the statement `t = 2.0` with

```
t = input("Choose time t: ")
t = float(t)
```

When the program reaches the line with `input()`, it pauses and waits for user input. Python interprets the user input as a string, and assigns that string to the variable `t`. In the next line, the function `float()` converts the string `t` into a real number (also called a "float").

You can prompt the user for multiple inputs. In place of the assignment statements for `t`, `v0x`, and `v0y` in the projectile code, we could write

```
t, v0x, v0y = input("Choose t, v0x, v0y: ").split()
```

then use `float()` to convert each variable `t`, `v0x` and `v0y` into a real number. The user's input values must be separated by spaces.

Exercise 2.8

Modify your projectile code to include user input for time `t` and the components of velocity, `v0x` and `v0y`.

Remember, `input()` interprets user input as a string. If the string is to be treated as a number, it must be converted to a real number or an integer. The function `float()` converts a string of digits with or without a decimal point into a real number. The function `int()` converts a string of digits without a decimal point into an integer.

2.9 Output formatting

Let's ask Python to compute 5.0/3.0 and print the result:

```
x = 5.0/3.0
print("five-thirds =", x)
```

We can achieve the same output with this code:

```
x = 5.0/3.0
print(f"five-thirds = {x}")
```

The `f` tells Python that what follows is an *f-string* (or *formatted string–literal*), which can include variables in curly brackets. The curly brackets can also contain expressions such as `x**2` or `3*x`.

The output of each code above is

```
five-thirds = 1.6666666666666667
```

We might not want to see so many digits. The output can be displayed in a more readable form by adding formatting instructions:

```
x = 5.0/3.0
print(f"five-thirds = {x:.3f}")
```

This yields the output

```
five-thirds = 1.667
```

The expression `.3f` after the colon tells Python to display `x` as a real number rounded to 3 decimal places. (The `f` stands for “float,” another name for a real number.)

Exercise 2.9

Modify the `print()` statements in your projectile code so that the results are displayed to 5 decimal places.

2.10 Lists

A Python *list* is a collection of numbers or strings surrounded by square brackets:

```
mylist = [1.0, 'elephant', -12.3, 3]
```

Lists can be added, for example

```
secondlist = [44, -7.6, 'tiger']
biglist = mylist + secondlist
print(biglist)
```

Later, we will learn other ways to manipulate lists.

Exercise 2.10

Create two lists in Python and add them. Can you multiply a list by an integer? Can you multiply two lists?

2.11 Line continuation and indentation

Most Python commands occupy a single line of code. If a command is long, it might be easier to read if it is extended across multiple lines. You can tell Python that a command extends to the next line using the line continuation character \. Note that \ should be the last character on the line that is to be continued. Make sure there are no spaces after \.

Exercise 2.11a

Consider the code

```
x = 2 + 3
print(f"x = {x}")
```

which outputs `x = 5`. Place a line continuation character \ after the `+` sign, and move the number `3` to the next line. Rerun the code.

Python can sometimes recognize that a command is incomplete and must extend to the next line. Typically, an opening left parenthesis `(` without a closing right parenthesis `)` signals an incomplete command that will be continued on the next line. Likewise, an opening square bracket `[` without a closing square bracket `]` tells Python that the command is incomplete. In these cases line continuation characters are not needed.

Exercise 2.11b

Modify the code above by placing parentheses around `2 + 3`. Break the command into two lines, somewhere between the parentheses. Does the code run? Now replace the statement `x = 2 + 3` with `x = [2, 3]`, turning `x` into a list. Extend the list across two lines, splitting it after the comma. Does the code run?

In later chapters we will see that some Python commands need to be indented. (None of the commands discussed so far should be indented.) For commands that are spread across multiple lines, the first line must be indented properly. Indentation for the subsequent lines doesn't matter—you can choose any pattern of indentation that makes your code easy to read.

Exercise 2.11c

Some programmers like to spread out the elements of a list; for example

```
x = ["cat",
     "dog",
     "fish",
     "bird"]
print(x)
```

Run this code and experiment with different amounts of indentation.

2.12 Order of execution

Consider the sequence of Python commands

```
x = 3.0
y = x + x**2
print(y)
```

If you run this code from an IDE editor, the commands will be executed in order, one after the other: `x = 3.0` is executed first, then `y = x + x**2`, then `print(y)`. Likewise, you could enter this code into a single cell of a Jupyter notebook. When the cell is executed, the commands will be executed in the order in which they appear. Alternatively, you might spread these commands across two or more cells of a Jupyter notebook. In this case the contents of each cell are executed when you type shift-enter (or shift-return). This is not necessarily the order in which the cells appear in the notebook.

For example, you could place

```
y = x + x**2
print(y)
```

in the first cell of a Jupyter notebook, and

```
x = 3.0
```

in the second cell. Although this is confusing and not recommended, the results will be fine if the second cell is executed before the first cell.

Exercise 2.12

Place `y = x + x**2` and `print(y)` in the first cell of a Jupyter notebook, and `x = 3.0` in the second cell. Execute the second cell (shift–enter), then the first. What happens? Now restart Python (often referred to as "restarting the kernel") and execute the two commands in the opposite order (the order in which they appear). What happens?