The fundamentals of the Python language and Jupyter notebooks

In [1]:
```
# Copyright (c) Thalesians Ltd, 2019-2023. All rights reserved.
# Copyright (c) Paul Alexander Bilokon, 2019-2023. All rights reserved.
# Author: Paul Alexander Bilokon <paul@thalesians.com>
# This version: 2.0 (2023.11.17)
# Version: 1.1 (2020.04.07)
# Email: info@thalesians.com
```

Motivation

Programming is one of the most important skills for a data scientist, and Python is the *de facto lingua franca* — the programming language of choice — for Data Science.

Data Scientists perform much of this programming inside the Jupyter environment.

In this Chapter we introduce just enough Python (and Jupyter) to get you started in Data Science.

Objectives

1. To introduce the Python programming language.
2. To explain where and how the reader can download the Anaconda Python distribution.
3. To introduce the Jupyter notebooks.
4. To demonstrate how different types of Jupyter notebook cells can be used.
5. To introduce the Python programming language.
6. To introduce variables.
7. To explain how to use Python's numeric data types: `int`s and `float`s.
8. To introduce type casting.
9. To demonstrate how to use Python libraries, using `math` as an example.
10. To explain the concept of dynamic typing.
11. To introduce strings.
12. To introduce `None`.
13. To introduce arithmetic expressions.
14. To introduce functions and explain their role in code reuse.
15. To explain why functions are first-class citizens in Python.
16. To introduce `bool`eans and logic.
17. To introduce comparison operators.

18. To explain how comparison operators can be combined with logical operators, such as `not`, `and`, and `or`.
19. To introduce `all` and `any`.
20. To explain that any value can be cast to a `bool`.
21. To introduce control flow and `if` statements.
22. To introduce key data structures: lists, tuples, dictionaries, and sets.
23. To explain the difference between the shallow copy and the deep copy.
24. To explain iteration, and introduce the `while` loop and the `for` loop.
25. To introduce the temporal types: `date`, `time`, and `datetime`.
26. To provide examples and exercises on this material, so the reader can practise programming.
27. To introduce the Python literature and web resources on Python.

What are Python and Jupyter

Python is a **programming language** that was created by Guido van Rossum and first released in 1991.

Its distinguishing characteristics are *straightforwardness* and *readability*, especially in comparison with other programming languages, such as C++ and Java. At the same time, Python is very expressive, powerful, and laconic, enabling programmers to express complex ideas in very little code.

Python is not only a language of choice for data science. It is frequently employed by web designers (for making websites), system administrators (for writing scripts and automation), hackers (also for writing scripts) and anyone who needs to process numeric and textual data in bulk.

There are two "lineages" of the Python language in existence. There is Python 2.x (the latest being version 2.7.18) and there is Python 3.x (the latest being version 3.12.0 at the time of writing). Python 3.x is supposed to supersede Python 2.x, but because so much systems code is powered by Python 2.x, Python 2.x is still supported and distributed. We shall stick with Python 3.x in the present work.

There are several Python **distributions** to choose from. The **Anaconda distribution** is a popular choice among Data Scientists. You can download the latest version of the Anaconda distribution for your operating system from https://www.anaconda.com/

If you have a 64-bit operating system, we suggest that you download the 64-bit version.

Once you have downloaded the distribution, install it.

Launch **Anaconda Navigator**. Once the Anaconda Navigator window shows up, launch **Jupyter notebook** from it. When it shows up in the browser, click on "New", then "Python 3". A blank Jupyter notebook should show up inviting you to enter some Python code.

It is worth noting that Jupyter notebooks are not the only way to write Python code. You could launch the Python **interpreter** (`python.exe` on Windows) from the **Anaconda**

Prompt and type in Python code closer to the metal. Or you could write your Python code in a text file, save it as `something.py` and end up with a standalone Python module or multiple such modules, forming a complex software product. This is something that we would do for a finished, polished solution in **production**. For **research** and **prototyping**, though, Jupyter notebooks are a perfect environment. (While Python is perfectly good for many production use cases, for others you may consider migrating to a language like C++, C#, or Java.)

For completeness, we shall mention that you don't have to use Python in Jupyter notebooks. The name "Jupyter" itself stands for "Julia, Python, R" — indeed, other programming languages, such as kdb+'s q, can be used in Jupyter notebooks, although we shall stick with Python in this work.

Introduction to Jupyter

Jupyter notebooks are at the core of Python's research environment. In Jupyter notebooks, the data is

- loaded,
- cleaned,
- visualised,
- analysed,
- documented,

possibly over multiple iterations, until the desired result is obtained. It is therefore unsurprising that Jupyter notebooks are often quite messy. Until they are finally cleaned up to present the conclusions of the research work. In fact, what you are reading right now is also a Jupyter notebook.

Cells

A Jupyter notebook comprises a column of basic building blocks called **cells**.

To insert a new cell in Jupyter, first click on an existing cell, then click on "Insert" in Jupyter's menu and select "Insert Cell Above" or "Insert Cell Below".

Under the menu, in the toolbar, there is a drop-down box with cell types: "Code", "Markdown", "Raw NBConvert", and "Heading". Click on an existing cell, then alter its type by selecting a different value from that drop-down box.

The most important cell types for us are "Code" and "Markdown".

"Code" cells, such as the one below...

In [2]:
```
3 + 5
```

Out[2]:
```
8
```

allow you to enter Python code (in our example, the numeric expression `3 + 5`) as "In" (input) and display the result as "Out" (output, in our example, `8`). Don't forget to press [Shift] + [Enter], once you have entered the code in your "Code" cell, to evaluate it and display the result in "Out". (The cursor will automatically move to the next cell.)

Markdown cells, such as the one you are currently reading, enable you to document your code. Moreover, you can use markdown syntax, such as `*this*` (to *italicise* the text), `**this**` (to make the text **bold**), include `# Headings` (prefixed with `#`), bulleted lists (prefixed with `*`, such as

- this
- simple
- list),

and numbered lists (prefixed with `1.`, such as

1. this
2. simple
3. list).

It is possible to include snippets of Python code, between two backticks, which will be `rendered in a special font`.

Finally, if you are a mathematician, you will be pleased to hear that you can include mathematical formulae, in $\LaTeX$, between two dollar signs (or double dollar signs for standalone equations). $\LaTeX$ looks pretty in Jupyter notebooks, such as this Euler's formula, $e^{ix} = \cos x + i \sin x$.

If we use double dollar signs, then we get

$$e^{ix} = \cos x + i \sin x.$$

The mathematical formulae introduced using a pair of single dollar signs, such as `$x^2$`: x^2, are called inline mathematical formulae, whereas those introduced using a pair of double dollar signs are called standalone.

Unfortunately, teaching you $\LaTeX$, Donald Knuth's mathematics typesetting language, is outside the scope of this work, but you will find plenty of resources on it online.

However, by now we hope that we have shown you the power of Markdown, Jupyter's language for documenting Python. The work that you are reading now is written in Markdown. You can read up on Markdown in Wikipedia: https://en.wikipedia.org/wiki/Markdown

Exercise

Typeset the following in Markdown:

In algebra, a **quadratic equation** (from the Latin *quadratus* for "square") is any equation having the form

$$ax^2 + bx + c = 0,$$

where

- x represents an unknown, and
- a, b, and c represent known numbers, with $a \neq 0$.

If $a = 0$, then the equation is linear, not quadratic, as there is no ax^2 term.

The numbers a, b, and c are the **coefficients** of the equation and may be distinguished by calling them, respectively, the **quadratic coefficient**, the **linear coefficient**, and the **constant** or **free term.**

The values of x that satisfy the equation are called **solutions** of the equation, and **roots** or **zeros** of its left-hand side. A quadratic equation has at most two solutions.

The value $b^2 - 4ac$ is known as the **discriminant**.

1. If the discriminant is *positive*, there are two real solutions given by the formula

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

2. If the discriminant is *zero*, there is one real solution (referred to as a **double root**) given by the formula

$$x = \frac{-b}{2a}.$$

3. If the discriminant is *negative*, there are no (real) solutions.

You can learn more about the quadratic equations on Wikipedia:
https://en.wikipedia.org/wiki/Quadratic_equation

Solution

In algebra, a **quadratic equation** (from the Latin *quadratus* for "square") is any equation having the form

$$ax^2 + bx + c = 0,$$

where * x represents an unknown, and * a, b, and c represent known numbers, with $a \neq 0$. If $a = 0$, then the equation is linear, not quadratic, as there is no ax^2 term. The numbers a, b, and c are the **coefficients** of the equation and may be distinguished by calling them, respectively, the **quadratic coefficient**, the **linear coefficient**, and the **constant** or **free term**. The values of x that satisfy the equation are called **solutions** of the equation, and **roots** or **zeros** of its left-hand side. A quadratic equation has at most two solutions. The value $b^2 - 4ac$ is known as the **discriminant**. 1. If the discriminant is *positive*, there are two real solutions given by the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

1. If the discriminant is *zero*, there is one real solution (referred to as a **double root**) given by the formula

$$x = \frac{-b}{2a}.$$

1. If the discriminant is *negative*, there are no (real) solutions. You can learn more about the quadratic equations on Wikipedia: https://en.wikipedia.org/wiki/Quadratic_equation

Introduction to Python

We have already entered our first piece of Python code, namely

In [3]:
```
3 + 5
```

Out[3]:
```
8
```

Exercise

Compute, using Python, (i) the product of seven and eight, (ii) the difference between 2190 and 518, (iii) the result of dividing 100 by four (iv) the result of multiplying by 10 of the difference between 2190 and 518.

Solution

In [4]:
```
7 * 8
```

Out[4]:
```
56
```

In [5]:
```
2190 - 518
```

Out[5]:
```
1672
```

In [6]:
```
100 / 4
```

Out[6]:
```
25.0
```

In [7]:
```
10 * (2190 - 518)
```

Out[7]:
```
16720
```

or

In [8]:
```
(2190 - 518) * 10
```

Out[8]:
```
16720
```

It should now be clear why we call Python a "supercalculator". Indeed, you could use Python as a calculator (but it is so much more). To start harnessing its power we should introduce

Variables

A **variable** is one of the most important concepts in programming. Essentially, it is a named value. Moreover, as the name suggests, this named value can be varied (changed), while keeping the name the same.

Let us create a variable named `a`. We create a variable by assigning to it, using the **assignment operator** `=`, its initial value:

In [9]:
```
a = 5
```

This **statement** (command) essentially says "set the variable `a` to value 5".

Once the variable `a` has been created and **initialised** (set to its initial value), we can use it in **expressions**, such as `a + 3`. When we write `a` in expressions, its value (5) will be substituted for `a`, so the result of the arithmetic expression `a + 3` will be `5 + 3`, in other words, 8:

In [10]:
```
a + 3
```

Out[10]:
```
8
```

We note that the difference between the statements and expressions is that the latter evaluate to a result.

As we said, the value of the variable can be varied (changed). Let us assign to `a` a different value, say, 7:

In [11]:
```
a = 7
```

Now when we evaluate the expression `a + 3`, we will get a different result, namely 10:

In [12]:
```
a + 3
```

Out[12]:
```
10
```

What if we now assign to `a` the result of the expression `a + 3`?

In [13]:
```
a = a + 3
```

First, the expression `a + 3` on the right-hand side of `=` is evaluated (it is 7 + 3, i.e. 10). Next, it is assigned to `a` as its new value. So, as a result of this assignment, the value of the variable `a` has become

In [14]:
```
a
```

Out[14]:
```
10
```

We may now introduce a different variable, say `b`,

In [15]:
```
b = 5
```

and use it in arithmetic expressions alongside `a`:

In [16]:
```
a + b + 3
```

Out[16]:
```
18
```

Notice that the values of the variables persist (are remembered) as we go from one Jupyter cell to the next.

We could write all of the above more succinctly in a single cell:

In [17]:
```
a = 7
a = a + 3
b = 5
a + b + 3
```

Out[17]:
```
18
```

Notice that only the result of the last expression, `a + b + 3` is returned as the output ("Out") by Jupyter.

Sometimes there is no "Out" to be printed, as is the case with assignment to a variable:

In [18]:
```
a = 10
```

However, as we said, variables persist throughout the Jupyter session, so we can inspect them in one of the following cells:

In [19]:
```
a
```

Out[19]:
```
10
```

Remember that only the result from the last expression is printed:

In [20]:
```
2 + 2
3 + 7
```

Out[20]:
```
10
```

However, you can print multiple things using the `print` **function**. This is convenient for inspecting intermediate results in your code:

In [21]:
```
print(2 + 2)
print(3 + 5)
3 + 4
```

Out[21]:
```
4
8
7
```

In the example above, `4` and `8` are displayed by the two `print` functions, whereas the output of the cell is `7`, which is the result of evaluating the last expression in the cell, `3 + 4`.

Note that variable names in Python are case-sensitive, so `a` is not the same as `A`, `myvar` is not the same as `myVar`:

In [22]:
```
a = 3
A = 5
a
```

Out[22]:
```
3
```

Whereas

In [23]:
```
A
```

Out[23]:
```
5
```

Exercise

Set the variable `a` to `15`, the variable `b` to `7`, then, without typing in any digits, swap the values of the two variables, so the variable `a` becomes equal to `7` and the variable `b` to `15`.

Solution

In [24]:
```
a = 15
b = 7
temp = a
a = b
b = temp
print('a:', a)  # or simply print(a)
print('b:', b)  # or simply print(b)
```

```
a: 7
b: 15
```

Notice that in Python we use `#` to introduce comments.

Here is another solution, without using a temporary variable. It's a bit trickier:

In [25]:
```
a = 15
b = 7
a = a + b
b = a - b
a = a - b
print(a)
print(b)
```

```
7
15
```

Numerics

So far all the values that we have dealt with in Python have been **numeric**, such as `3` and `5` in the expression

In [26]:
```
3 + 5
```

Out[26]:
```
8
```

The result, `8`, is also numeric.

Moreover, these values are all integers. An **integer** in programming is the same as in mathematics: a whole number with no digits after the decimal point:

In [27]:
```
8
```

Out[27]:
```
8
```

We can use the built-in Python function `type` to confirm that the **type** of 8 is indeed an integer (or `int` for short):

In [28]:
```
type(8)
```

Out[28]:
```
int
```

We can assign this value to a variable

In [29]:
```
my_int = 8
```

And then that variable will have the type integer:

In [30]:
```
type(my_int)
```

Out[30]:
```
int
```

We could print out the value of `my_int` along with the type of its value using `print`:

In [31]:
```
print(my_int, type(my_int))
```

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

Python supports fractions (mathematically speaking, **real numbers**), as well as integers. Fractions are implemented using a different type, the **floating point** type, `float`:

In [32]:
```
type(3.57)
```

Out[32]:
```
float
```

We can force a **literal** to be interpreted as a float (rather than as an integer) by including the decimal point:

In [33]:
```
type(42.)
```

Out[33]:
```
float
```

whereas

In [34]:
```
type(42)
```

Out[34]:
```
int
```

We say that `42.` is a `float` literal, whereas `42` is an `int` literal.

So let us get the terminology right. In the following code

In [35]:
```
a = 3
b = 7 + 5.3
print(b)
10 + a + b
```

Out[35]:
```
12.3
25.3
```

Here, `a = 3`, `b = 7 + 5.3`, `print(b)` are statements (i.e. commands: "assign a value to a variable", "print something"). Whereas "things" that evaluate the values (including the values themselves), `3`, `7 + 5.3`, `10 + a + b` are expressions. `3`, `7` and `10` are literals of type `int`. 5.3 is a literal of type `float`. The value of the variable `a` is of type `int`, as is the value of the literal `3`. Notice that you can mix values of different types in the same expression, as in `7 + 5.3`. The type of the result in this particular instance will be

In [36]:
```
type(7 + 5.3)
```

Out[36]:
```
float
```

`float` is also the type of the expression `10 + a + b`.

We could also **cast** a value of type `int` to `float`:

In [37]:
```
float(42)
```

Out[37]:
```
42.0
```

In [38]:
```
type(float(42))
```

Out[38]:
```
float
```

whereas

In [39]:
```
type(42)
```

Out[39]:
```
int
```

When casting a value of type `float` to type `int` we may end up losing precision as we lose all digits after the decimal point:

In [40]:
```
int(3.57)
```

Out[40]:
```
3
```

In [41]:
```
type(int(3.57))
```

Out[41]:
```
int
```

The `float` data type is used throughout data science to represent numerical values in arithmetic operations.

Exercise

Is the sum of `3` and `3.57` an `int` or a `float`? Will you lose precision by casting `3` to a `float` then back to an `int`? Will you lose precision by casting `3.57` to an `int` then back to a `float`?

Solution

In [42]:
```
3 + 3.57
```

Out[42]:
```
6.57
```

In [43]:
```
type(3 + 3.57)
```

Out[43]:
```
float
```

In [44]:
```
int(float(3))
```

Out[44]:
```
3
```

We haven't lost any precision.

In [45]:
```
float(int(3.57))
```

Out[45]:
```
3.0
```

This time we have lost precision — the digits after the decimal point.

Standard python libraries

The power of Python is in its **libraries** — pre-written collections of Python code that do useful stuff for us. We make use of libraries by `import`ing their **modules**:

In [46]:
```
import math
```

Once we have imported the standard Python library module `math`, we can start using functions defined in it, such as `sqrt` for the square root:

In [47]:
```
math.sqrt(3.57)
```

Out[47]:
```
1.8894443627691184
```

We can use the results of these functions in expressions:

In [48]:
```
4.5 + 2 * math.sqrt(3.57)
```

Out[48]:
```
8.278888725538238
```

Modules may define other things in addition to functions, such as constants. In particular, the `math` module defines the mathematical π ("pi") constant, which relates the radius of a circle to its circumference (via $C = 2\pi r$, where r is the radius, C the circumference):

In [49]:
```
math.pi
```

Out[49]:
```
3.141592653589793
```

As a side comment, many fractions, such as the **transcendental** number π, cannot be represented exactly using floating point. Floating point arithmetics relies on truncated, approximate representations of real numbers, which may lead to all sorts of **numerical issues** (often subtle) in scientific computing. However, what we are doing here is too basic for us to worry about these numerical issues. If you want to *really* understand floating point

numbers, have a look at the paper *What Every Computer Scientist Should Know About Floating-Point Arithmetic* by David Goldberg (Google it).

To check what functions, constants, etc. are defined in a library you can either Google its documentation, or use

In [50]:
```
dir(math)
```

Out[50]:
```
['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 'atan',
 'atan2',
 'atanh',
 'cbrt',
 'ceil',
 'comb',
 'copysign',
 'cos',
 'cosh',
 'degrees',
 'dist',
 'e',
 'erf',
 'erfc',
 'exp',
 'exp2',
 'expm1',
 'fabs',
 'factorial',
 'floor',
 'fmod',
 'frexp',
 'fsum',
 'gamma',
 'gcd',
 'hypot',
 'inf',
 'isclose',
 'isfinite',
 'isinf',
 'isnan',
 'isqrt',
 'lcm',
 'ldexp',
 'lgamma',
 'log',
 'log10',
 'log1p',
 'log2',
 'modf',
 'nan',
 'nextafter',
 'perm',
 'pi',
 'pow',
 'prod',
 'radians',
 'remainder',
 'sin',
 'sinh',
 'sqrt',
 'tan',
 'tanh',
 'tau',
```

```
 'trunc',
 'ulp']
```

Exercise

In one of the previous exercises we have already mentioned quadratic equations. Use `math` to find both solutions of the quadratic equation $2x^2 - 3x + \frac{1}{2} = 0$.

Solution

In [51]:
```
a = 2.
b = -3.
c = .5
discriminant = b * b - 4 * a * c
discriminant
```

Out[51]:
```
5.0
```

The discriminant is positive, so we should indeed have two real solutions:

In [52]:
```
x1 = (-b + math.sqrt(discriminant)) / (2. * a)
print('x1:', x1)
x2 = (-b - math.sqrt(discriminant)) / (2. * a)
print('x2:', x2)
```

```
x1: 1.3090169943749475
x2: 0.19098300562505255
```

Let us check:

In [53]:
```
a * x1 * x1 + b * x1 + c
```

Out[53]:
```
0.0
```

In [54]:
```
a * x2 * x2 + b * x2 + c
```

Out[54]:
```
5.551115123125783e-17
```

We have encountered a (very minor) floating point arithmetics, numerical issue: instead of zero, we got a very small number (`...e-17` should be read as $\ldots \cdot 10^{-17}$). For all practical purposes, this is zero, and `x2` is indeed a solution of our quadratic equation.

Dynamic typing

Let us set the variable `x`, so it equals 65:

In [55]:
```
x = 65
```

Its type, then, will be integer:

In [56]:
```
type(x)
```

Out[56]:
```
int
```

We could overwrite `x` with a value of a different type, such as a float:

In [57]:
```
x = 3.57
```

The type of `x` has now changed:

In [58]:
```
type(x)
```

Out[58]:
```
float
```

Some programming languages (such as Java, C++, C#, and many others) would not allow overwriting `x` with a value of a different type: once something is an `int`, it is always an `int`. We say that these languages are **statically typed**, whereas Python is **dynamically typed**. Types are important in Python, and Python is still a **strongly typed** language, although the type of a variable may change over the lifetime of the program, hence the expression: "dynamically typed".

Strings

The string type allows us define textual variables. A `string` literal is enclosed within two single `'` or double `"` quotation marks.

In [59]:
```
my_str = 'foo'
print(my_str, type(my_str))
```

```
foo <class 'str'>
```

It is customary for introductions to programming languages to include an example that prints out the string `'Hello, World!'` In Python, this is a one-liner:

In [60]:
```
print('Hello, World!')
```

```
Hello, World!
```

The function `len` returns the length of a string:

In [61]:
```
len('Hello, World!')
```

Out[61]:
```
13
```

We can access individual characters in a string using **indexing** with the square brackets. Notice that the indexing starts at zero, thus

In [62]:
```
'Hello, World!'[0]
```

Out[62]:
```
'H'
```

whereas

In [63]:
```
'Hello, World!'[1]
```

Out[63]:
```
'e'
```

We can also index from the back using negative indices:

In [64]:
```
'Hello, World!'[-1]
```

Out[64]:
```
'!'
```

Moreover, we can index longer **substrings**, rather than individual characters:

In [65]:
```
'Hello, World!'[3:7]
```

Out[65]:
```
'lo, '
```

Notice that the first index is inclusive, whereas the second exclusive, so the resulting substring consists of characters at indices 3, 4, 5, and 6 (but not 7).

When indexing, we can also provide a step:

In [66]:
```
'Hello, World!'[3:7:2]
```

Out[66]:
```
'l,'
```

If we skip the first index, we start at the beginning. If we skip the second index, we go until the end.

In [67]:
```
'Hello, World!'[::2]
```

Out[67]:
```
'Hlo ol!'
```

Of course, instead of repearing the string 'Hello, World!' so many times (while running the risk of mistyping it), we should have stored it in a variable...

In [68]:
```
greet = 'Hello, World!'
```

...and then indexed:

In [69]:
```
greet[::2]
```

Out[69]:
```
'Hlo ol!'
```

One of the most useful operations on strings is **concatenation**. It enables us to produce a single string from multiple:

In [70]:
```
'first' + 'second'
```

Out[70]:
```
'firstsecond'
```

In [71]:
```
separator = ', '
'first' + separator + 'second' + separator + 'third'
```

Out[71]:
```
'first, second, third'
```

Exercise

Use indexing and concatenation to obtain the string `'World, Hello!'` from `'Hello, World!'`.

Solution

In [72]:
```
greet = 'Hello, World!'
```

In [73]:
```
greet[7:12] + greet[5:7] + greet[:5] + greet[-1]
```

Out[73]:
```
'World, Hello!'
```

None

We can set Python variables to a special value, `None`,

In [74]:
```
a = None
```

of a special type,

In [75]:
```
type(a)
```

Out[75]:
```
NoneType
```

`None` is used to signal that the value is absent or missing.

In fact, this is the value implicitly returned by statements, such as

In [76]:
```
print(357)
```

```
357
```

Arithmetic expressions

Python supports the standard arithmetic operators:

In [77]:
```
print('Addition:', 5 + 3)
print('Subtraction:', 5 - 3)
print('Multiplication:', 5 * 3)
print('Division:', 5 / 3)
print('Exponentiataion:', 5**3)
print('Modulo:', 5 % 3)
```

```
Addition: 8
Subtraction: 2
Multiplication: 15
Division: 1.6666666666666667
Exponentiataion: 125
Modulo: 2
```

Python also supports integer division, which produces the largest integer less than or equal to, in the next example, `5 / 3`:

In [78]:
```
5 // 3
```

Out[78]:
```
1
```

If any of the arguments is a `float`, the result will also be of type `float`:

In [79]:
```
5.1 // 3.1
```

Out[79]:
```
1.0
```

Expressions such as

In [80]:
```
3 + 5
```

Out[80]:
```
8
```

In [81]:
```
2. * x + 7.
```

Out[81]:
```
14.14
```

evaluate to numbers (whether integers, or floating point numbers). They are known as **arithmetic** expressions.

We can perform some other common operations on numerics:

In [82]:
```
print('Absolute value:', abs(-5))
print('Rounding:', round(3.56))
print('Maximum value:', max(3, 2, 8, 10, 2, 5))
print('Minimum value:', min(3, 2, 8, 10, 2, 5))
```

```
Absolute value: 5
Rounding: 4
Maximum value: 10
Minimum value: 2
```

Functions

Suppose that we have written some code to compute the area of a circle:

In [83]:
```
radius = 5.
area = math.pi * radius * radius
print(area)
```

```
78.53981633974483
```

There is little point in rewriting it each time we encounter a new circle with a different radius. So we wrap it inside a **function**, which takes `radius` as its **parameter** (**argument**) and **returns** the result:

In [84]:
```
def area_of_circle(radius):
    area = math.pi * radius * radius
    return area
```

We can **call** our function with the values of the arguments that we need in each case:

In [85]:
```
area_of_circle(5.)
```

Out[85]:
```
78.53981633974483
```

In [86]:
```
r = 7.5
area_of_circle(r)
```

Out[86]:
```
176.71458676442586
```

Functions can have multiple arguments:

In [87]:
```
def area_of_triangle(base, height):
    print('Base:', base)
    print('Height:', height)
    area = .5 * base * height
    return area

area_of_triangle(3., 5.)
```

```
Base: 3.0
Height: 5.0
7.5
```

Out[87]:

Notice how the **block** of code was indented (we chose to indent it using four spaces, although some people prefer to use tabs) to dilimit it, designating it as the **body** of the function `area_of_triangle`. The function call that ensues, `area_of_triangle(3., 5.)`, is not indented, and is not part of that body.

The variables `base` and `height` are defined only within the body of the function. We say that those variables' **scope** is limited to the body of the function.

It is possible to call the function specifying the values of the arguments in order

In [88]:
```
area_of_triangle(3., 5.)
```

```
Base: 3.0
Height: 5.0
7.5
```

Out[88]:

or by name

In [89]:
```
area_of_triangle(height=5., base=3.)
```

```
Base: 3.0
Height: 5.0
7.5
```

Out[89]:

Functions can also specify default values for their arguments in their definitions:

In [90]:
```
def area_of_triangle(base, height=5.):
    return .5 * base * height
```

So calling

In [91]:
```
area_of_triangle(3., 5.)
```

```
7.5
```

Out[91]:

can now be equivalently done as

In [92]:
```
area_of_triangle(3.)
```

```
7.5
```

Out[92]:

Notice that like everything else (e.g. integers) functions are objects and **first-class citizens**. Thus we can think of `area_of_triangle` as a variable set to a value of type `function`:

In [93]:
```
type(area_of_triangle)
```

Out[93]:
```
function
```

Function objects can be passed to other functions as parameters:

In [94]:
```
def add(x, y):
    return x + y

def multiply(x, y):
    return x * y

def result_printer(op, x, y):
    print('The result is', op(x, y))

result_printer(add, 3, 5)
result_printer(multiply, 3, 5)
```

```
The result is 8
The result is 15
```

Good programmers are masters of **code reuse** therefore they wrap generally useful pieces of code into convenient functions.

If a library defines the function that we need, then we don't need to write our own. We have already seen (and used) the function

In [95]:
```
math.sqrt(9.)
```

Out[95]:
```
3.0
```

Exercise

Write two functions that will return the two roots of a given quadratic equation. Test them on the quadratic equation $2x^2 - 3x + \frac{1}{2} = 0$.

Solution

In [96]:
```
def discriminant(a, b=0., c=0.):
    return b * b - 4. * a * c
```

In [97]:
```
def root1(a, b=0., c=0.):
    d = discriminant(a, b, c)
    return (-b - math.sqrt(d)) / (2. * a)
```

In [98]:
```
def root2(a, b=0., c=0.):
    d = discriminant(a, b, c)
    return (-b + math.sqrt(d)) / (2. * a)
```

In [99]:
```
root1(2., -3., .5)
```

Out[99]:
```
0.19098300562505255
```

In [100…
```
root2(2., -3., .5)
```
Out[100]:
```
1.3090169943749475
```

Booleans and logic

`bool`ean is a binary variable type, that can either be `True` or `False`. It is so named after the self-taught English mathematician, philosopher, and logician George Boole: https://en.wikipedia.org/wiki/George_Boole

In [101…
```
my_bool = True
print(my_bool, type(my_bool))
```
```
True <class 'bool'>
```

In [102…
```
my_bool = False
print(my_bool, type(my_bool))
```
```
False <class 'bool'>
```

Let us set `x` to the integer 10:

In [103…
```
x = 10
```

Expressions that evaluate to either `True` or `False` are known as **boolean** expressions.

In [104…
```
x < 10
```
Out[104]:
```
False
```

In [105…
```
type(x < 10)
```
Out[105]:
```
bool
```

Different boolean expressions can be obtained by using different **comparison operators**, such as **less than**:

In [106…
```
x < 10
```
Out[106]:
```
False
```

less than or equals:

In [107…
```
x <= 10
```
Out[107]:
```
True
```

equals:

In [108…
```
x == 10
```
Out[108]:
```
True
```

greater than or equals:

In [109…
```
x >= 10
```
Out[109]:
```
True
```

greater than:

In [110…
```
x > 10
```
Out[110]:
```
False
```

not equal:

In [111…
```
x != 10
```
Out[111]:
```
False
```

And these comparison operators can be combined with **logical operators**, such as `not`, `and`, and `or`:

In [112…
```
x <= 10 and x % 2 == 1
```
Out[112]:
```
False
```

In [113…
```
x <= 10 or x % 2 == 1
```
Out[113]:
```
True
```

We can also use the built-in function `all`:

In [114…
```
all([x > 1, 5 <= x, 5 > 3, 7 != 1])
```
Out[114]:
```
True
```

which is equivalent to

In [115…
```
x > 1 and 5 <= x and 5 > 3 and 7 != 1
```
Out[115]:
```
True
```

Similarly,

In [116…
```
all([x > 1, 5 <= x, x == 5, 5 > 3, 7 != 1])
```
Out[116]:
```
False
```

is equivalent to

In [117…
```
x > 1 and 5 <= x and x == 5 and 5 > 3 and 7 != 1
```
Out[117]:
```
False
```

Another builtin function, `any`, enables us to write

In [118…
```
any([x > 1, 5 <= x, x == 5, 5 > 3, 7 != 1])
```

Out[118]:
```
True
```

which is somewhat more succinct and arguably more readable than the equivalent

In [119…
```
x > 1 or 5 <= x or x == 5 or 5 > 3 or 7 != 1
```

Out[119]:
```
True
```

Each data type can also be cast to `True` or `False`. As a general rule, objects like `string` if they do not contain anything, zeros, and `None` will be cast to `False`, while everything else will be cast to `True`:

In [120…
```
print(bool())
print(bool(''))
print(bool(' '))
print(bool(0))
print(bool(0.))
print(bool(1))
print(bool(1.5))
print(bool(None))
```

```
False
False
True
False
False
True
True
False
```

Control flow

We can control the flow of our programs using the basic logical operators and `if` statements. The `if` statement evaluates the `if` block if the given boolean expression is `True` and the `else` block (as long as it is present) if the given boolean expression is `False`. Else-if or `elif` lets us set a specific boolean expression to evaluate if the base case is not `True`.

In [121…
```
if x <= 7:
    print('x is less than or equal to seven')
else:
    print('x is greater than seven')
```

```
x is greater than seven
```

In [122…
```
if x <= 7:
    print('x is less than or equal to seven')
```

In this example, `x > 7` (so `x <= 7` is `False`) but there is no `else` block, so nothing is evaluated/printed.

In [123…
```
if x <= 7:
    print('x is less than or equal to seven')
elif x <= 10:
    print('x is greater than seven but less than or equal to ten')
elif x <= 15:
    print('x is greater than ten but less than or equal to fifteen')
```

```
else:
    print('x is greater than fifteen')
```

```
x is greater than seven but less than or equal to ten
```

We can also have nested `if-else` statements:

In [124…
```
if x % 2 == 0:
    print('x is divisible by 2')
    if x % 5 == 0:
        print('x is divisible by 2 and 5')
elif x % 5 == 0:
    print('x is divisible by 5 but not 2')
else:
    print('x is divisible by neither 2 nor 5')
```

```
x is divisible by 2
x is divisible by 2 and 5
```

To check whether a variable is `None` we use `is None` rather than `== None`:

In [125…
```
if x is None:
    print('x is None')
else:
    print('x is not None')
```

```
x is not None
```

Exercise

Write a function that will return the number of real solutions of a quadratic equation.

Solution

In [126…
```
def number_of_solutions(a, b=0, c=0):
    # We could have used a function to compute the discriminant, but didn't in this
    discriminant = b * b - 4. * a * c
    if discriminant > 0:
        return 2
    elif discriminant == 0:
        return 1
    else:
        return 0
```

In [127…
```
number_of_solutions(2., -3., .5)
```

Out[127]:
```
2
```

In [128…
```
number_of_solutions(1., 3., 5.)
```

Out[128]:
```
0
```

Exercise

The **Fibonacci sequence** is a sequence of integers, starting with zero and one, such that each term in the sequence is the sum of the previous two. Thus the first few terms of the sequence are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, etc. Write a function that, given `n`, will return the `n` th term of the Fibonacci sequence.

Solution

In [129…
```
def fibonacci(n):
    if n == 0: return 0
    elif n == 1: return 1
    else:
        return fibonacci(n - 2) + fibonacci(n - 1)
```

In [130…
```
fibonacci(0)
```

Out[130]:
```
0
```

In [131…
```
fibonacci(1)
```

Out[131]:
```
1
```

In [132…
```
fibonacci(2)
```

Out[132]:
```
1
```

In [133…
```
fibonacci(3)
```

Out[133]:
```
2
```

In [134…
```
fibonacci(4)
```

Out[134]:
```
3
```

In [135…
```
fibonacci(5)
```

Out[135]:
```
5
```

In [136…
```
fibonacci(6)
```

Out[136]:
```
8
```

Note that, when defining the function `fibonacci`, we have used what is known as **recursion**: the function returns specific values in the **base cases** (when `n = 0` and `n = 1`) and calls itself in the **recursive case.**

Data structures

As data scientists, we care a lot about **data structures** that let us store and access large amounts of data. Some such data structures, such as lists, tuples, dictionaries, and sets, are part of the Python standard. Others, such as multidimensional arrays and dataframes, are provided by third-party, but *de facto* standard libraries, such as NumPy and Pandas, respectively.

Lists

A list is arguably the most commonly used data structure in Python. Its core function is to allow storage of and access to various elements. Financial data in particular are often

represented as time-series, which are, collections of observed values with corresponding time. To define a `list` we use square brackets `[]`:

In [137…
```
my_list = [1, 5, 6, 3]
print(my_list, type(my_list))
```

```
[1, 5, 6, 3] <class 'list'>
```

Python allows us to combine elements of different types into the same `list`:

In [138…
```
my_list = [3, "hello world", True, None, 3, math.pi]
print(my_list)
```

```
[3, 'hello world', True, None, 3, 3.141592653589793]
```

Let's examine the length of our list:

In [139…
```
len(my_list)
```

Out[139]:
```
6
```

Notice that repeated values are counted as distinct elements.

Accessing elements of a `list` is performed using indexing with `[]`. Rememeber that the index of the first element of the list is `0`:

In [140…
```
print(my_list[0])
print(my_list[1])
print(my_list[3])
print(my_list[2])
print(my_list[4])
```

```
3
hello world
None
True
3
```

You may also access elements from the end of a list by using negative indexing:

In [141…
```
print(my_list[-1])
print(my_list[-2])
print(my_list[-3])
print(my_list[-4])
```

```
3.141592653589793
3
None
True
```

We may set an element in a `list` to a new value:

In [142…
```
my_list[-1] = 4
print(my_list)
```

```
[3, 'hello world', True, None, 3, 4]
```

We can select a sublist from the list:

In [143…
```
my_list = ['problems','worthy','of','attack','prove','their','worth','by','fighting
print(my_list[3:6])
```

```
['attack', 'prove', 'their']
```

Notice that the index 3 is inclusive, whereas the index 6 exclusive, so, as a result, we obtain a sublist containing elements at indices 3, 4, and 5 (but not 6).

We may also select sublists without the lower and/or upper bounds:

In [144…
```
print(my_list[3:])
print(my_list[:5])
print(my_list[:])
```

```
['attack', 'prove', 'their', 'worth', 'by', 'fighting', 'back']
['problems', 'worthy', 'of', 'attack', 'prove']
['problems', 'worthy', 'of', 'attack', 'prove', 'their', 'worth', 'by', 'fightin
g', 'back']
```

You can specify a step:

In [145…
```
my_list[::2]
```

Out[145]:
```
['problems', 'of', 'prove', 'worth', 'fighting']
```

Reverse the order by setting a negative step size:

In [146…
```
my_list[::-1]
```

Out[146]:
```
['back',
 'fighting',
 'by',
 'worth',
 'their',
 'prove',
 'attack',
 'of',
 'worthy',
 'problems']
```

And combine the step with lower and upper bounds:

In [147…
```
my_list[2:10:3]
```

Out[147]:
```
['of', 'their', 'fighting']
```

We can use Python's `range` function to generate a list of consecutive integers:

In [148…
```
list(range(10))
```

Out[148]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Conveniently, we can specify a step as well:

In [149…
```
list(range(1,10,2))
```

Out[149]:
```
[1, 3, 5, 7, 9]
```

We can add elements to the end of the list via the `append` method (a **method** is a function associated with a particular object, in our example, `my_list`):

In [150…
```
my_list = list(range(0,10))
my_list.append(25)
my_list.append(25)
print(my_list)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 25, 25]
```

We can remove specific elements by calling the method `remove` and supplying it with the value of an element that we would like to remove. Note that only the first instance of an element will be removed:

In [151…
```
my_list.remove(25)
my_list.remove(my_list[5])
print(my_list)
```

```
[0, 1, 2, 3, 4, 6, 7, 8, 9, 25]
```

We can **filter** a list using something like

In [152…
```
list(filter(lambda x: x > 5, my_list))
```

Out[152]:
```
[6, 7, 8, 9, 25]
```

Here the **lambda** or **anonymous function** `lambda x: x > 5` is equivalent to

In [153…
```
def my_func(x): return x > 5
```

but shorter and avoids giving the function a name — it is not needed, as we don't intent to call this function in the future.

We can **map** or apply a function (or lambda) to each element of a list:

In [154…
```
list(map(lambda x: x*2, my_list))
```

Out[154]:
```
[0, 2, 4, 6, 8, 12, 14, 16, 18, 50]
```

The function `sorted` sorts a list without modifying it — it returns a new, sorted list, while keeping the original one intact:

In [155…
```
sorted(my_list, reverse=True)
```

Out[155]:
```
[25, 9, 8, 7, 6, 4, 3, 2, 1, 0]
```

In [156…
```
my_list
```

Out[156]:
```
[0, 1, 2, 3, 4, 6, 7, 8, 9, 25]
```

On the other hand, the method `sort` modifies the list — it sorts it **in place**:

In [157…
```
my_list.sort(reverse=True)
```

In [158…
```
my_list
```

Out[158]:
```
[25, 9, 8, 7, 6, 4, 3, 2, 1, 0]
```

Tuples

Let us consider an example.

In [159…
```
a = [3, "hello world", True, None, 3, math.pi]
a = ['some', 'other', 'list']
a
```

Out[159]:
```
['some', 'other', 'list']
```

The variable `a` was first assigned to the list `[3, "hello world", True, None, 3, math.pi]`, but was then reassigned to another list, `['some', 'other', 'list']`. Variables can be thought of as pointers (**references**) to objects in memory, such as lists. Two variables can reference the same object in memory, e.g.

In [160…
```
a = [3, "hello world", True, None, 3, math.pi]
b = a
```

Now,

In [161…
```
a
```

Out[161]:
```
[3, 'hello world', True, None, 3, 3.141592653589793]
```

In [162…
```
b
```

Out[162]:
```
[3, 'hello world', True, None, 3, 3.141592653589793]
```

Since lists are **mutable** objects, they can be modified after construction. Notice that we are not reassigning a variable so it references a new object in memory, we are modifying the object that it is already pointing to:

In [163…
```
a[2] = False
a
```

Out[163]:
```
[3, 'hello world', False, None, 3, 3.141592653589793]
```

Notice that, since the variable `b` is referencing the same object, its value has also changed:

In [164…
```
b
```

Out[164]:
```
[3, 'hello world', False, None, 3, 3.141592653589793]
```

In this sense, mutable objects are somewhat dangerous. Consider the following code:

In [165…
```
def my_mean(arg):
    # This could be a long function, which, perhaps by mistake,
    # modifies arg:
    # ...
    arg[3] = 11.7
    # ...
    return sum(arg) / len(arg)
```

Let's apply this function to

In [166…
```
a = [4.25, 18.5, 22.5, 13.7, 25.4]
```

The result of the (broken) `my_mean` looks roughly correct...

In [167…
```
my_mean(a)
```

Out[167]:
```
16.47
```

...although it's not. But what's worse, the user of `my_mean`, who never expected that function to modify its argument, is in for a surprise:

In [168…
```
a
```

Out[168]:
```
[4.25, 18.5, 22.5, 11.7, 25.4]
```

When we doubt the validity of some code, we may defensively copy the arguments like so:

In [169…
```
a = [4.25, 18.5, 22.5, 13.7, 25.4]
print(my_mean(a.copy()))
a
```

Out[169]:
```
16.47
[4.25, 18.5, 22.5, 13.7, 25.4]
```

Notice that a copy is equal to...

In [170…
```
a = [4.25, 18.5, 22.5, 13.7, 25.4]
b = [4.25, 18.5, 22.5, 13.7, 25.4]
a == b
```

Out[170]:
```
True
```

...but not identical to (does not correspond to the same object in memory as) the original:

In [171…
```
a is b
```

Out[171]:
```
False
```

Whereas if both variables point to the same object in memory we get both **equality** and **identity**:

In [172…
```
a = [4.25, 18.5, 22.5, 13.7, 25.4]
b = a
print(a == b)
print(a is b)
```

```
True
True
```

We can also check this by examining the `id` of the object, which in CPython is equal to its address in memory:

In [173…
```
id(a)
```

Out[173]:
```
2363859066816
```

In [174…
```
id(b)
```

Out[174]:
```
2363859066816
```

Mutable objects, such as lists, may therefore be a source of subtle and difficult to track bugs. They are less safe than **immutable** objects, which cannot be modified after construction. Mutable objects are particularly dangerous in multi-threaded environments where code runs in parallel.

Fortunately, Python has a built-in data structure, which is very similar to a list, but immutable: a **tuple**. We create a tuple instead of a list by using round brackets instead of square brackets:

In [175…
```
a = (4.25, 18.5, 22.5, 13.7, 25.4)
type(a)
```

Out[175]:
```
tuple
```

Alternatively, we may cast a list to a tuple:

In [176…
```
a = tuple([4.25, 18.5, 22.5, 13.7, 25.4])
type(a)
```

Out[176]:
```
tuple
```

Once a tuple has been created, it cannot be modified: `a[0] = 3.57` will raise an error and the tuple doesn't have methods such as `a.append(3.57)`.

Notice that

In [177…
```
(3)
```

Out[177]:
```
3
```

is interpreted as the number 3, whereas

In [178…
```
(3,)
```

Out[178]:
```
(3,)
```

is interpreted as a tuple containing a single element — number 3.

Exercise

Write a single function that will return the two roots of a given quadratic equation as a tuple. Test your function on the quadratic equation $2x^2 - 3x + \frac{1}{2} = 0$.

Solution

In [179…
```
def quadratic(x, a, b=0., c=0.,):
    return a * x * x + b * x + c
```

In [180…
```
def discriminant(a, b=0., c=0.):
    return b * b - 4. * a * c
```

In [181…
```
def solve_quadratic(a, b=0., c=0.):
    d = discriminant(a=a, b=b, c=c)
    if d > 0.:
        return (-b - math.sqrt(d)) / (2. * a), (-b + math.sqrt(d)) / (2. * a)
    elif d == 0.:
        return (-b / (2. * a),)
    else:
        return tuple()
```

In [182…
```
sols = solve_quadratic(2., -3., .5)
sols
```

Out[182]:
```
(0.19098300562505255, 1.3090169943749475)
```

In [183…
```
quadratic(sols[0], 2., -3., .5)
```

Out[183]:
```
5.551115123125783e-17
```

In [184…
```
quadratic(sols[1], 2., -3., .5)
```

Out[184]:
```
0.0
```

In [185…
```
sols = solve_quadratic(1., 2., 1.)
sols
```

Out[185]:
```
(-1.0,)
```

In [186…
```
quadratic(sols[0], 1., 2., 1.)
```

Out[186]:
```
0.0
```

In [187…
```
sols = solve_quadratic(2., 0., 0.5)
sols
```

Out[187]:
```
()
```

Exercise

Set the variable `a` to `15`, the variable `b` to `7`, then, without typing in any digits, *without using arithmetics, and without introducing any new variables*, swap the values of the two variables, so the variable `a` becomes equal to `7` and the variable `b` to `15`. Hint: use tuples.

Solution

In [188…
```
a = 15
b = 7
a, b = b, a
print('a:', a)
print('b:', b)
```

```
a: 7
b: 15
```

Shallow versus deep copy

As we have mentioned the copying of objects, we should point out that there are the **shallow** and **deep** variants of copy in Python.

The shallow variant copies the object but not its elements; elements of the original data structure are still referenced. For example:

In [189…
```
a = (['one', 'two', 'three'], [0, 1, 2, 3, 4, 5])
```

`a` is a tuple of lists, i.e. a tuple whose elements are both lists.

Let us take a closer look at this object:

In [190…
```
id(a)
```
Out[190]:
```
2363858216704
```

In [191…
```
a[0]
```
Out[191]:
```
['one', 'two', 'three']
```

In [192…
```
id(a[0])
```
Out[192]:
```
2363859061504
```

In [193…
```
a[1]
```
Out[193]:
```
[0, 1, 2, 3, 4, 5]
```

In [194…
```
id(a[1])
```
Out[194]:
```
2363859294592
```

Let us now take a (shallow) copy of `a`, `a_copy`:

In [195…
```
import copy
a_copy = copy.copy(a)
```

In [196…
```
a_copy
```
Out[196]:
```
(['one', 'two', 'three'], [0, 1, 2, 3, 4, 5])
```

Notice that

In [197…
```
id(a_copy)
```
Out[197]:
```
2363858216704
```

is different from

In [198…
```
id(a)
```
Out[198]:
```
2363858216704
```

but, because we took only a shallow copy,

In [199…
```
id(a[0]) == id(a_copy[0])
```
Out[199]:
```
True
```

and

In [200…
```
id(a[1]) == id(a_copy[1])
```
Out[200]:
```
True
```

We didn't copy the lists; only the tuple that contains them. Thus `a[0]` and `a_copy[0]` are referencing the same object. In other words,

In [201…
```
a[0] is a_copy[0]
```
Out[201]:
```
True
```

Let us now try to modify an element of `a_copy`:

In [202…
```
a_copy[0].append('four')
```

While we cannot change the tuple itself since the tuple is immutable, we can change the tuple's elements, which in this particular case are mutable. `a_copy`'s zeroth element has changed:

In [203…
```
a_copy
```
Out[203]:
```
(['one', 'two', 'three', 'four'], [0, 1, 2, 3, 4, 5])
```

And, because `a_copy` is a shallow copy of `a`, the zeroth element of `a` has also changed:

In [204…
```
a
```
Out[204]:
```
(['one', 'two', 'three', 'four'], [0, 1, 2, 3, 4, 5])
```

This isn't the case for the deep copy:

In [205…
```
a_deep_copy = copy.deepcopy(a)
```

In [206…
```
a
```
Out[206]:
```
(['one', 'two', 'three', 'four'], [0, 1, 2, 3, 4, 5])
```

In [207…
```
a_deep_copy
```
Out[207]:
```
(['one', 'two', 'three', 'four'], [0, 1, 2, 3, 4, 5])
```

In [208…
```
a_deep_copy[0].append('five')
```

In [209…
```
a_deep_copy
```
Out[209]:
```
(['one', 'two', 'three', 'four', 'five'], [0, 1, 2, 3, 4, 5])
```

Notice that the zeroth element of the original `a` has not changed:

In [210…
```
a
```

Out[210]:
```
(['one', 'two', 'three', 'four'], [0, 1, 2, 3, 4, 5])
```

Since we took a deep copy of `a` to produce `a_deep_copy` from `a`, `a_deep_copy[0]` and `a[0]` are distinct objects:

In [211…
```
id(a[0])
```

Out[211]:
```
2363859061504
```

In [212…
```
id(a_deep_copy[0])
```

Out[212]:
```
2363859248768
```

Dictionaries

Python **dictionaries** are powerful abstractions that let us define **key-value pairs**. In other programming languages, such abstractions are also known as **maps**. We define dictionaries by using the following notation:

In [213…
```
book = {
    'authors': 'Michael Berthold',
    'title': 'Intelligent Data Analysis',
    'publisher': 'Springer',
    'year': 2003
}
```

In this dictionary, the **keys** `'authors'`, `'title'`, `'publisher'`, and `'year'` correspond to the **values** `'Michael Berthold'`, `'Intelligent Data Analysis'`, `'Springer'`, and `2003`, respectively.

Data structures can be **nested**. For example, the value in a dictionary may itself be a data structure, such as a list:

In [214…
```
book = {
    'authors': ['Michael Berthold', 'David J. Hand'],
    'title': 'Intelligent Data Analysis',
    'publisher': 'Springer',
    'year': 2003
}
```

We can index the dictionary using the `[]` notation:

In [215…
```
book['authors']
```

Out[215]:
```
['Michael Berthold', 'David J. Hand']
```

Notice that in this instance the indices are strings, not integers. (But it's possible to use integers as dictionary keys.)

Dictionaries are mutable:

In [216…
```
my_dict = {1:'one',2:'two',3:'three'}
print(my_dict[1])
my_dict[4] = 'four'
print(my_dict)
```

```
one
{1: 'one', 2: 'two', 3: 'three', 4: 'four'}
```

Let's see how we could define a toy dataset of financial time-series. The keys of our dictionary are going to be tickers, the values lists of stock prices.

In [217…
```
my_dict = {
    'AAPL':[200,201,200.1,205],
    'GOOG':[700,750,640,720],
    'AMZN':[900,850,920,910]
}
```

In [218…
```
my_dict
```

Out[218]:
```
{'AAPL': [200, 201, 200.1, 205],
 'GOOG': [700, 750, 640, 720],
 'AMZN': [900, 850, 920, 910]}
```

Here, each value is a list of asset prices, e.g.

In [219…
```
my_dict['AMZN']
```

Out[219]:
```
[900, 850, 920, 910]
```

Sets

Sets are defined using the syntax

In [220…
```
s = {'red', 'green', 'blue', 'red', 'red', 'green', 'blue'}
```

Alternatively, a set can be constructed from another collection (such as the list in the following example) using the `set` constructor:

In [221…
```
s = set(['red', 'green', 'blue', 'red', 'red', 'green', 'blue'])
```

Unlike lists, repeated elements in sets count as one:

In [222…
```
s
```

Out[222]:
```
{'blue', 'green', 'red'}
```

In [223…
```
len(s)
```

Out[223]:
```
3
```

It doesn't make sense to talk about the indices of the elements of the set. The element is either present in or absent from the set:

In [224…
```
'green' in s
```

Out[224]:
```
True
```

In [225…
```
'purple' in s
```
Out[225]:
```
False
```

In [226…
```
'purple' not in s
```
Out[226]:
```
True
```

Sets are mutable:

In [227…
```
s.add('cyan')
s
```
Out[227]:
```
{'blue', 'cyan', 'green', 'red'}
```

We can consider **unions** of sets...

In [228…
```
{'red', 'green', 'blue', 'red', 'red', 'green', 'blue'}.union({'purple', 'green', '
```
Out[228]:
```
{'blue', 'green', 'purple', 'red', 'yellow'}
```

...**intersections** of sets...

In [229…
```
{'red', 'green', 'blue', 'red', 'red', 'green', 'blue'}.intersection({'purple', 'gr
```
Out[229]:
```
{'green'}
```

...as well as set **differences**:

In [230…
```
{'red', 'green', 'blue', 'red', 'red', 'green', 'blue'}.difference({'purple', 'gree
```
Out[230]:
```
{'blue', 'red'}
```

A union of the sets `a` and `b` is a set whose elemenents are the objects that are either elements of the set `a` or elements of the set `b` (or both). An intersection of the sets `a` and `b` is a set whose elements are the objects that are elements of the set `a` and also elements of the set `b`. A difference of the sets `a` and `b` is a set whose elements are elements of the set `a` but not elements of the set `b`.

Iteration

Iteration is the process of going through the elements of a collection. To facilitate iteration, we rely on **loops**, which allow us to evaluate blocks of code multiple times.

The `while` loop

The `while` loop is a basic loop that will execute if some condition is `True` and stops executing when the condition is `False`:

In [231…
```
a = True
while a:
    print('inside while loop')
    a = False
```

```
inside while loop
```

In this example the body of the loop was evaluated exactly once, because the variable `a`, initially `True`, was set to `False` on the very first iteration.

Here is another example:

In [232…
```
x = 0
while x < 10:
    print(x)
    x = x + 1
```

```
0
1
2
3
4
5
6
7
8
9
```

We remark that `x = x + 1` can also be written as `x += 1`.

We start by setting the variable `x` to 0. We then repeat the indented block, consisting of the lines `print(x)` and `x = x + 1`, while the condition `x < 10` holds. The second line in this block, `x = x + 1`, keeps incrementing the variable `x` by 1, so eventually the condition `x < 10` will end up being false. Thus we get ten **iterations** of the loop. If we check the value of the variable `x` after we have left the loop, we find that it is

In [233…
```
x
```

Out[233]:
```
10
```

We can escape the loop early by issuing a `break` command. This is particularly useful for infinite loops:

In [234…
```
a = 0
while True:
    a += 1
    print(a)
    if a == 10:
        break
```

```
1
2
3
4
5
6
7
8
9
10
```

The `for` loop

We use `for` loops to iterate through lists, dictionaries, ranges and other data structures. The `for` loop will go through every element in the collection and perform a given task on that element. Here is an example — let us add up all elements of a range using a `for` loop:

In [235…
```
a = 0
for i in range(10):
    a += i
print(a)
```

```
45
```

Exercise

Define

In [236…
```
list_of_words = ['problems', 'worthy', 'of', 'attack', 'prove', 'their', 'worth', '
```

Use a `for` loop to concatenate these words into a single string, separating them with spaces.

In [237…
```
list(enumerate(list_of_words))
```

Out[237]:
```
[(0, 'problems'),
 (1, 'worthy'),
 (2, 'of'),
 (3, 'attack'),
 (4, 'prove'),
 (5, 'their'),
 (6, 'worth'),
 (7, 'by'),
 (8, 'fighting'),
 (9, 'back')]
```

Solution

In [238…
```
s = ''
for w in list_of_words:
    s += w + ' '
s
```

Out[238]:
```
'problems worthy of attack prove their worth by fighting back '
```

Notice that our solution produces a spurious space at the end of `s`. We could resolve this issue by using a `for` loop with `enumerate`. `enumerate` "labels" each of the items in an iterable with its index:

In [239…
```
for i, w in enumerate(list_of_words):
    print(i, w)
```

```
0 problems
1 worthy
2 of
3 attack
4 prove
5 their
6 worth
7 by
8 fighting
9 back
```

Here is a solution without the spurious space at the end:

In [240…
```
s = ''
for i, w in enumerate(list_of_words):
    if i > 0: s += ' '
    s += w
s
```

Out[240]:
```
'problems worthy of attack prove their worth by fighting back'
```

We didn't have to use a `for` loop in this case, since `string` has the `join` method:

In [241…
```
result = ' '.join(list_of_words)
result
```

Out[241]:
```
'problems worthy of attack prove their worth by fighting back'
```

Its "opposite" (inverse) is the string's method `split`:

In [242…
```
result.split(' ')
```

Out[242]:
```
['problems',
 'worthy',
 'of',
 'attack',
 'prove',
 'their',
 'worth',
 'by',
 'fighting',
 'back']
```

There is more than one way to iterate through dictionaries:

In [243…
```
my_dict = {
    'AAPL':[200,201,200.1,205],
    'GOOG':[700,750,640,720],
    'AMZN':[900,850,920,910]
}
```

By default, we shall be iterating only through the dictionary's keys:

In [244…
```
for k in my_dict:
    print(k)
```

```
AAPL
GOOG
AMZN
```

The dictionary's `items` method gives access to keys and values:

In [245…
```
for key, value in my_dict.items():
    print(key, value[0])
```

```
AAPL 200
GOOG 700
AMZN 900
```

Exercise

Compute the `n` th term of the Fibonacci sequence using loops instead of recursion.

Solution

In [246…
```
def fibonacci(n):
    a, b = 0, 1
    if n == 0: return a
    if n == 1: return b
    for i in range(n):
        a, b = b, a + b
    return a
```

In [247…
```
fibonacci(9)
```

Out[247]:
```
34
```

Exercise

Consider the following list of postcodes:

In [248…
```
postcodes = ['E14 4AD', 'NW1 6AA', 'EC2N 2DB', 'EC4A 2BE', 'E14 5AB', 'EH1 3AB']
```

Use a loop to count how many of them occur in East London.

East London postcodes start with letter E followed by a number. Thus `'E14 4AD'` is an East London postcode, whereas `'NW1 6AA'` and `'EC2N 2DB'` are not.

Solution

In [249…
```
east_london_count = 0
for postcode in postcodes:
    if postcode[0] == 'E' and postcode[1] >= '0' and postcode[1] <= '9':
        east_london_count += 1
east_london_count
```

Out[249]:
```
2
```

Note that strings are compared lexicographically, thus

In [250…
```
'abc' < 'abd'
```

Out[250]:
```
True
```

Thus the following comparison checks whether a string represents a digit:

In [251…
```
'abc' >= '0' and 'abc' <= '9'
```
Out[251]:
```
False
```

In [252…
```
'3' >= '0' and '3' <= '9'
```
Out[252]:
```
True
```

Notice that the comparison `type(postcode[1]) == int` would not work. Indeed, while in some cases the character at index 1 may represent a digit, its type is still `str`. Note the ampersands in the output of the following

In [253…
```
'E14 4AD'[1]
```
Out[253]:
```
'1'
```

In [254…
```
type('E14 4AD'[1])
```
Out[254]:
```
str
```

Exercise

Consider the following list of emails:

In [255…
```
emails = [
    'isaac.newton@morganstanley.com',
    'gottfried.leibniz@gs.com',
    'andrey.kolmogorov@jpmorgan.com',
    'joseph.fourier@bnpparibas.com',
    'henri.lebesgue@bnpparibas.com',
    'norbert.wiener@jpmorgan.com',
    'paul.levy@jpmorgan.com'
]
```

Count how often each domain (`morganstanley.com`, `gs.com`, etc.) appears in this list.

Solution

In [256…
```
counts = {}
for email in emails:
    split_email = email.split('@')
    name = split_email[0]
    domain = split_email[1]
    if domain in counts:
        counts[domain] += 1
    else:
        counts[domain] = 1
counts
```
Out[256]:
```
{'morganstanley.com': 1, 'gs.com': 1, 'jpmorgan.com': 3, 'bnpparibas.com': 2}
```

Exercise

Consider the following list of emails:

In [257…
```
emails = [
    'isaac.newton@morganstanley.com',
    'gottfried.leibniz@gs.com',
    'isaac.newton@morganstanley.com',
    'isaac.newton@morganstanley.com',
    'andrey.kolmogorov@jpmorgan.com',
    'gottfried.leibniz@gs.com',
    'joseph.fourier@bnpparibas.com',
    'henri.lebesgue@bnpparibas.com',
    'andrey.kolmogorov@jpmorgan.com',
    'norbert.wiener@jpmorgan.com',
    'andrey.kolmogorov@jpmorgan.com',
    'paul.levy@jpmorgan.com'
]
```

Count how many times each email is repeated in this list. Which email or emails occur the maximum number of times?

Solution

In [258…
```
repeat_counts = {}
for email in emails:
    if email in repeat_counts:
        repeat_counts[email] = repeat_counts[email] + 1
    else:
        repeat_counts[email] = 1
```

In [259…
```
repeat_counts
```

Out[259]:
```
{'isaac.newton@morganstanley.com': 3,
 'gottfried.leibniz@gs.com': 2,
 'andrey.kolmogorov@jpmorgan.com': 3,
 'joseph.fourier@bnpparibas.com': 1,
 'henri.lebesgue@bnpparibas.com': 1,
 'norbert.wiener@jpmorgan.com': 1,
 'paul.levy@jpmorgan.com': 1}
```

In [260…
```
max_repeats = max(map(lambda x: x[1], repeat_counts.items()))
max_repeats
```

Out[260]:
```
3
```

In [261…
```
list(filter(lambda x: x[1] == max_repeats, repeat_counts.items()))
```

Out[261]:
```
[('isaac.newton@morganstanley.com', 3), ('andrey.kolmogorov@jpmorgan.com', 3)]
```

Exercise

Consider the following lists that represent a table of assignment marks:

In [262…
```
names = [
    'Isaac Newton',
    'Gottfried Leibniz',
    'Isaac Newton',
    'Isaac Newton',
    'Andrey Kolmogorov',
    'Gottfried Leibniz',
    'Joseph Fourier',
    'Henri Lebesgue',
```

```
    'Andrey Kolmogorov',
    'Norbert Wiener',
    'Andrey Kolmogorov',
    'Paul Levy'
]
marks = [
    8.,
    9.,
    8.5,
    10.,
    9.,
    7.,
    8.5,
    10.,
    8.5,
    9.,
    10.,
    10.
]
```

Find the best assignment mark for each person. You may choose to use the `zip` function — look it up.

Solution

In [263…
```
top_marks = {}
for name, mark in zip(names, marks):
    if name in top_marks:
        if mark > top_marks[name]:
            top_marks[name] = mark
    else:
        top_marks[name] = mark
top_marks
```

Out[263]:
```
{'Isaac Newton': 10.0,
 'Gottfried Leibniz': 9.0,
 'Andrey Kolmogorov': 10.0,
 'Joseph Fourier': 8.5,
 'Henri Lebesgue': 10.0,
 'Norbert Wiener': 9.0,
 'Paul Levy': 10.0}
```

We see that dictionaries are often used as supporting, auxiliary structures for processing lists.

Exercise

Consider the following list of daily temperatures:

In [264…
```
temperatures = [25., 25.5, 27., None, 28., 28.5, None, None, 30., 29.5, 29.5, None,
```

Interpolate the missing values using the previous non-`None` value. In other words, the first `None` should be replaced with `27.`, the next two with `28.5`, and the last with `29.5`.

Solution

In [265…
```
interpolated_temperatures = []
non_none_temperature = None
for temperature in temperatures:
```

```
    if temperature is not None:
        interpolated_temperatures.append(temperature)
        non_none_temperature = temperature
    else:
        interpolated_temperatures.append(non_none_temperature)
```

In [266…
```
interpolated_temperatures
```

Out[266]:
```
[25.0, 25.5, 27.0, 27.0, 28.0, 28.5, 28.5, 28.5, 30.0, 29.5, 29.5, 29.5, 28.0]
```

Exercise

Consider the function

In [267…
```
def func(x): return -2. * x * x + 7. * x + 3.
```

Find, correct to the nearest integer, the value of `x` at which this function attains its maximum (also specify what that maximum value is).

Solution

In [268…
```
xs = range(-10, 10, 1)
ys = list(map(func, xs))
# So we better understand the geometry of the graph (a parabola with the two branch
print(ys)
x_at_max_y = None
max_y = None
for x, y in zip(xs, ys):
    if max_y is None or y > max_y:
        max_y = y
        x_at_max_y = x
print('x:', x_at_max_y, ', y:', max_y)
```

```
[-267.0, -222.0, -181.0, -144.0, -111.0, -82.0, -57.0, -36.0, -19.0, -6.0, 3.0, 8.
0, 9.0, 6.0, -1.0, -12.0, -27.0, -46.0, -69.0, -96.0]
x: 2 , y: 9.0
```

Exercise

Consider the following function of two variables:

In [269…
```
def func(x, y):
    return -x * x - y * y + x * y - 3. * x + 5. * y + 5.
```

On the grid $-10 \leq x \leq 10$, $-10 \leq y \leq 10$, find the point $(x_{\max}, y_{\max})$ at which this function achieves its maximum and that maximum value.

Solution

In [270…
```
max_f, max_f_x, max_f_y = None, None, None
for x in range(-10, 10):
    for y in range(-10, 10):
        f = func(x, y)
        if max_f is None or f > max_f:
            max_f, max_f_x, max_f_y = f, x, y
print('x:', max_f_x)
```

```
print('y:', max_f_y)
print('max f:', max_f)
```

```
x: -1
y: 2
max f: 11.0
```

Notice that the above is not the only maximum for this integer grid:

In [271…
```
func(0, 2)
```

Out[271]:
```
11.0
```

In [272…
```
func(0, 3)
```

Out[272]:
```
11.0
```

To find all points on the grid at which the maximum is attained:

In [273…
```
max_f_points = []
for x in range(-10, 10):
    for y in range(-10, 10):
        if func(x, y) == max_f: max_f_points.append((x, y))
max_f_points
```

Out[273]:
```
[(-1, 2), (0, 2), (0, 3)]
```

We have restricted ourselves to whole numbers. Looks like the maximum of the function lies somewhere in the triangle defined by the above three points:

In [274…
```
func(-.5, 2.5)
```

Out[274]:
```
11.25
```

Temporal types

In data science we often have to deal with temporal, time series data. Python's standard `datetime` module

In [275…
```
import datetime as dt
```

comes in useful in these cases. This module implements dates:

In [276…
```
my_date = dt.date(2019, 8, 7)
my_date
```

Out[276]:
```
datetime.date(2019, 8, 7)
```

times:

In [277…
```
my_time = dt.time(14, 3, 8, 357123)
my_time
```

Out[277]:
```
datetime.time(14, 3, 8, 357123)
```

datetimes:

In [278…
```
my_datetime = dt.datetime(2019, 8, 7, 14, 3, 8, 357123)
my_datetime
```

Out[278]:
```
datetime.datetime(2019, 8, 7, 14, 3, 8, 357123)
```

and timedeltas (differences, time intervals):

In [279…
```
my_timedelta = dt.timedelta(seconds=5)
my_timedelta
```

Out[279]:
```
datetime.timedelta(seconds=5)
```

Dates, times, and datetimes can be queried for their individual parts:

In [280…
```
my_datetime.date()
```

Out[280]:
```
datetime.date(2019, 8, 7)
```

In [281…
```
my_datetime.time()
```

Out[281]:
```
datetime.time(14, 3, 8, 357123)
```

In [282…
```
my_datetime.year, my_datetime.month, my_datetime.day, \
my_datetime.hour, my_datetime.minute, my_datetime.second, my_datetime.microsecond
```

Out[282]:
```
(2019, 8, 7, 14, 3, 8, 357123)
```

In [283…
```
my_date.year, my_date.month, my_date.day
```

Out[283]:
```
(2019, 8, 7)
```

In [284…
```
my_time.hour, my_time.minute, my_time.second
```

Out[284]:
```
(14, 3, 8)
```

To get the current datetime, we can use

In [285…
```
dt.datetime.now()
```

Out[285]:
```
datetime.datetime(2023, 11, 17, 12, 47, 21, 279258)
```

Moreover, Python supports temporal arithmetics:

In [286…
```
my_timedelta = dt.datetime(2019, 8, 7, 14, 9, 53, 357123) - dt.datetime(2019, 8, 7,
my_timedelta
```

Out[286]:
```
datetime.timedelta(seconds=7793, microseconds=357123)
```

In [287…
```
my_timedelta.days, my_timedelta.seconds, my_timedelta.microseconds
```

Out[287]:
```
(0, 7793, 357123)
```

In [288…
```
my_timedelta.total_seconds()
```

Out[288]:
```
7793.357123
```

In [289…
```
my_timedelta = dt.date(2019, 8, 8) - dt.date(2019, 8, 7)
my_timedelta
```

Out[289]:
```
datetime.timedelta(days=1)
```

In [290…
```
my_timedelta.days, my_timedelta.seconds, my_timedelta.microseconds
```

Out[290]:
```
(1, 0, 0)
```

In [291…
```
my_timedelta.total_seconds()
```

Out[291]:
```
86400.0
```

Parsing and formatting temporal data

Temporal data often occurs as strings. One of the most common tasks is **parsing** — the conversion of that textual data to an appropriate, in this case temporal, data type.

In Python, temporal data can be parsed using `strptime`:

In [292…
```
dt.datetime.strptime('2019.09.01', '%Y.%m.%d').date()
```

Out[292]:
```
datetime.date(2019, 9, 1)
```

Note that we had to specify the format for the string to be parsed, in the above example `'%Y.%m.%d'`.

In [293…
```
dt.datetime.strptime('01-09-2019', '%d-%m-%Y').date()
```

Out[293]:
```
datetime.date(2019, 9, 1)
```

In [294…
```
dt.datetime.strptime('20:46:35', "%H:%M:%S").time()
```

Out[294]:
```
datetime.time(20, 46, 35)
```

In [295…
```
dt.datetime.strptime('2019.09.01T20:46:35.123', "%Y.%m.%dT%H:%M:%S.%f")
```

Out[295]:
```
datetime.datetime(2019, 9, 1, 20, 46, 35, 123000)
```

It is also often useful to **format** temporal data as strings of an appropriate format. This is achieved using `strftime`. (In `strptime`, `p` stands for "parse"; in `strftime`, `f` stands for "format".)

In [296…
```
dt.datetime.strftime(dt.datetime(2019, 9, 1), '%Y.%m.%d')
```

Out[296]:
```
'2019.09.01'
```

In [297…
```
dt.datetime.strftime(dt.datetime(2019, 9, 1), '%d-%m-%Y')
```

Out[297]:
```
'01-09-2019'
```

In [298…
```
dt.datetime.strftime(dt.datetime(2019, 9, 1, 20, 46, 35, 123000), "%Y.%m.%dT%H:%M:%
```

Out[298]:
```
'2019.09.01T20:46:35.123000'
```

Conclusion

We have barely scratched the surface of Python programming. We have introduced the basic building blocks, but the programmer's skill comes from masterfully using their combinations. Metaphorically speaking, we have introduced the valid chess moves, but haven't considered the openings, tactics, and strategy. To become a good Programmer or Data Scientist, you need to further advance your programming skills.

Bibliography

To learn more about Python, you may wish to consult the official Python documentation: https://docs.python.org/3/

You may also wish to look at the following books:

- Wes McKinney. *Python for Data Analysis*, 2nd Edition. O'Reilly Media, Inc., 2017. This book goes through the foundations of the Python programming language, the Jupyter environment, industry-standard Python libraries, especially Pandas, and applications to time series data.
- Yves Hilpisch. *Python for Finance*, 2nd Edition. O'Reilly Media, Inc., 2019. This book goes through the foundations of the Python programming language, industry-standard Python libraries, including Pandas, an introduction to object-oriented programming, and applications to automated trading, derivatives, and portfolio valuation.
- Francois Chollet. *Deep Learning with Python*. Manning Publications, 2018. This book focusses on Deep Learning (DL), rather than Python, but considers how Python libraries could be used to facilitate DL. It does not cover the fundamentals of Python in great depth.
- David Beazley, Brian K. Jones. *Python Cookbook*, 3rd Edition. O'Reilly Media, Inc., 2013. As the title suggests, this book acts as a cookbook for Python programming.
- Luciano Ramalho. *Fluent Python: Clear, Concise, and Effective Programming*, O'Reilly Media, Inc., 2015. As the subtitle suggests, this book explains how to use Python in a clear, concise, and effective manner.

If you have a question check whether it is already answered on StackOverflow: https://stackoverflow.com/

Very often, if you google your Python programming question, you end up on StackOverflow.

If you are using Python for data science, check out

- https://www.kaggle.com/ (great for data science in general)
- https://www.quandl.com/ (a great source of financial and alternative data).