FIRST INTRODUCTION TO PYTHON

Installation, `Jupyter notebook`, arithmetic and logical expressions, data types, variables, functions, conditions, loops, `NumPy` arrays

Roman Gerasimov

1 Installation and first launch

Being one of the most used programming languages in the world, Python comes in a huge variety of distributions, each having its own unique flaws, virtues and compatibility issues. We recommend using the development environment called `Jupyter Notebook`[1]. The environment provides means to write, execute, organize, store and annotate your Python code and is widely employed in many fields including astronomy.

Google Colaboratory is an online `Jupyter Notebook` service provided for free by Google at https://colab.research.google.com/. Once accessed and logged in through your Google account, the website will display an example project titled "Welcome to Colaboratory". A new blank project can be started by clicking on File ⟩ New notebook in the main menu (Fig. 1). The project will be saved in the *Google Drive* associated with the account. You may prefer *Google Colaboratory* as your main notebook environment as it requires no installation and executes code on external servers without exerting any load on the local computer. It is however limited in available configuration options which is why we recommend installing a local environment as well.

`Jupyter Notebook` can be installed locally on any major operating system (Windows, Linux and Mac OS included) as a part of the `Anaconda` distribution. `Anaconda` can be downloaded for free from its official website: https://www.anaconda.com/download/.

Once installed, open the terminal (on Mac OS, search for `terminal`; on Windows, search for `cmd.exe`; on most Linux distributions ctrl + ⇧ + T will do), change your directory to where you would like your Python code to be saved (use the `cd` command regardless of your operating system) and launch `Jupyter Notebook` by typing `jupyter notebook` (Fig. 2).

[1] The name `Jupyter` is an acronym of three programming languages supported by the environment: **Ju**lia, **Py**thon and **R**

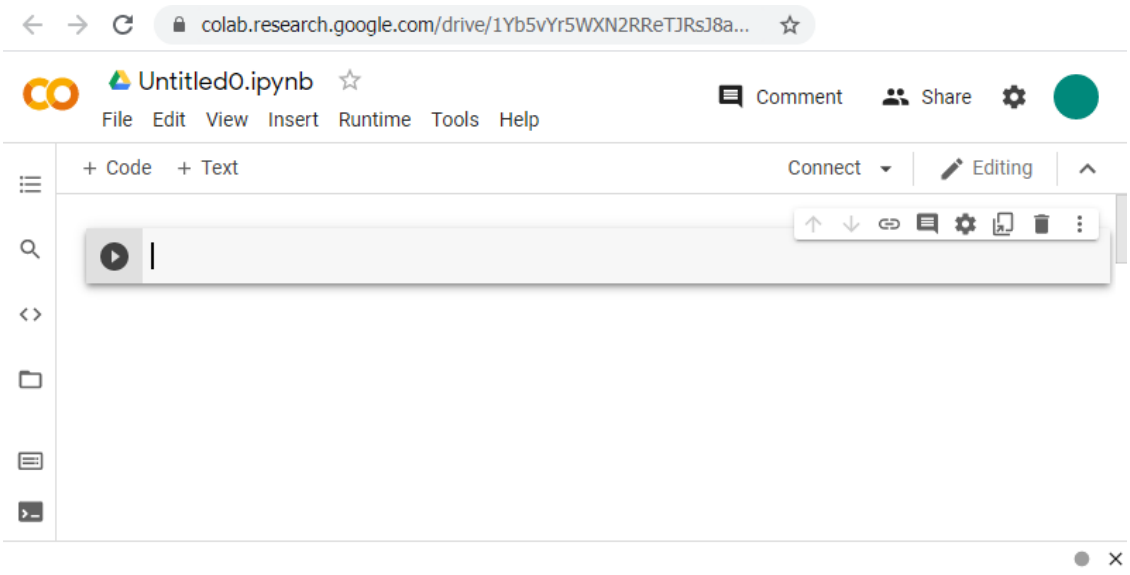

Figure 1: A blank `Jupiter Notebook` loaded through *Google Colaboratory*

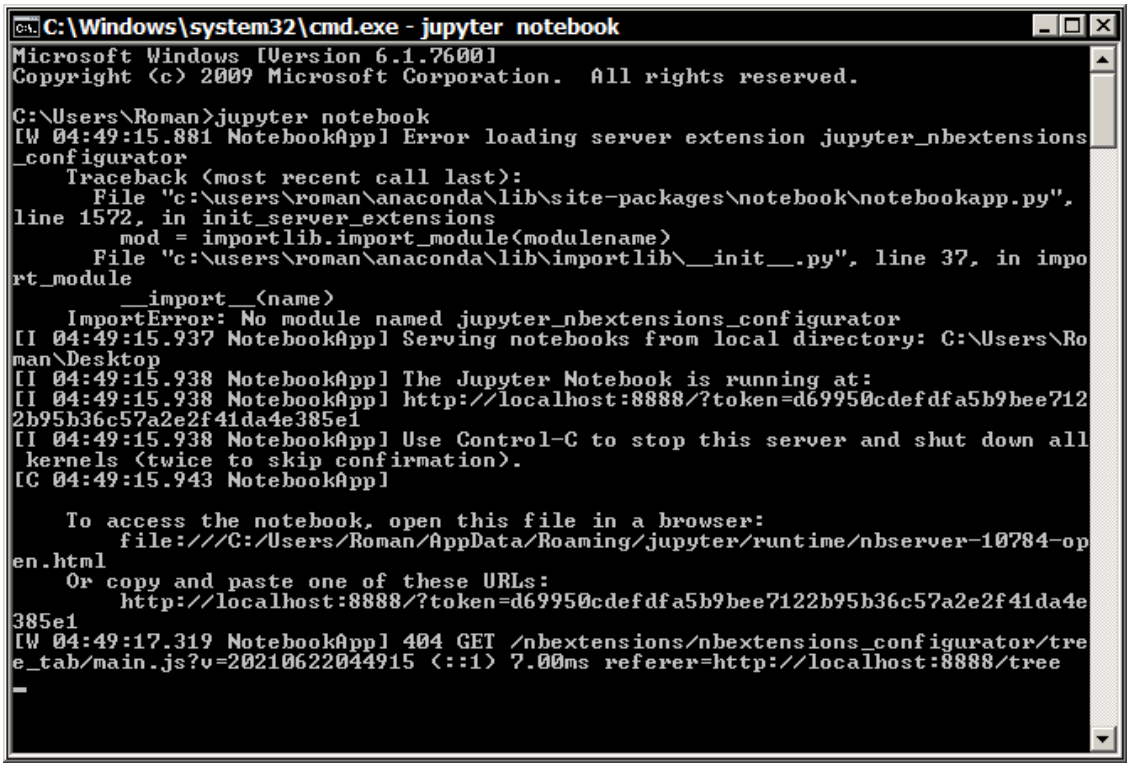

Figure 2: Terminal view of a running `Jupyter notebook` under Windows.

It may take a few seconds for the environment to start up. Eventually, it should open a new tab in your browser, listing the files in your chosen folder. Keep the terminal session window open for as long as you are working on your code. When finished, save your changes, close the browser tab, switch back to the terminal window and press ctrl+C to stop the notebook server. Alternatively, you can simply close the terminal window. In some cases, it may take the environment a few seconds or minutes to end your session.

2 Jupyter notebook

`Jupyter Notebook` is an environment where you can write, organize and execute your Python code as well as add formatted comments that may help you or somebody else understand what is going on. A new notebook will receive a default name (usually, `Untitled.ipynb` or `Untitled0.ipynb`). You can (*and should*) rename your notebook to something more meaningful by clicking on its title at the top of the page (however, the suffix `.ipynb` should be kept). Your work will be saved automatically every now and then; however, it is a good idea to save everything manually with File>Save or ctrl+S before closing the browser tab at the end.

A notebook is composed of cells. Originally, there will only be one, but you can create more with the + button in the toolbar (locally) or the +Code and +Text buttons in *Google Colaboratory*. You can also copy, delete and reorganize your cells as you wish, using the other buttons. Every cell will be one of two types: a code cell or a markdown (text) cell. Locally, you can change the type of any cell by activating it with a click and using the type dropdown in the toolbar (it is set to "`Code`" by default). As the name suggests, a code cell contains Python code that can be executed. A markdown (text) cell contains arbitrary comments that can be formatted using a markup language called Markdown. Have a look at the Markdown Cheatsheet, https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet, to learn more about this language. Should you happen to be familiar with other markup languages, such as HTML or LaTeX, you may find it useful to know that both are partially supported in markdown cells as well. Try creating a new cell of markdown (text) type and entering in the following lines:

```
# Large header
## Smaller header

Some text. **Some bold text**. *Some italic text*. Here is a bullet list:

* Item 1
* Item 2

<table>
  <tr>
    <td>This is a table...</td>
    <td>...created with HTML</td>
  </tr>
</table>

And this is a LaTeX equation: $L=4\pi\sigma R^{2} T^{4}$.
```

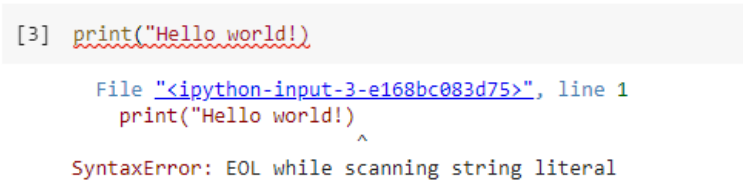

Figure 3: Error issued by the environment in response to invalid Python code. In this case, the closing quotation mark after `Hello world!` is missing. The explanatory text underneath the cell identifies the line where the error occurred and provides a line of description explaining exactly what went wrong. In this case, the problem is that the Python interpreter ran into the end of line (EOL) before the string literal (i.e. `Hello world!`) ended.

When finished, press ⇧+⏎. Hopefully, you should see some basic formatting written in Markdown, followed by a table written in HTML and an equation written in LaTeX. At this stage, do not worry if you never encountered those markup languages before and the above makes little sense. Nonetheless, do try to annotate any Python code that you write with markdown cells and feel free to experiment with basic formatting either by using the Cheatsheet or by asking your mentors for help!

Of course, the primary purpose of a notebook is to store and execute Python code. Create a new cell and make sure that it is a code cell. Type in the following line and execute the cell, as before, by pressing ⇧+⏎:

```
print("Hello world!")
```

This is a line of code written in Python that outputs ("*prints*") "Hello world!" onto the screen. On *Google Colaboratory* the cell might take a few seconds to run, but the output should eventually appear immediately under the cell.

If the above worked successfully, try running invalid Python code to see how the environment reacts. For example, remove one of the quotation marks in the code and execute the cell again (Fig. 3).

The environment should say that a syntax error was found, give you a short description of the error (in my case, "EOL while scanning string literal", where EOL means "end of line") and state where the error is (in my case, it is "line 1"). No doubt, you will be running into a lot of errors and this information may help finding and fixing them. Your mentor will also need to know this information should you require assistance troubleshooting the error.

3 Basic expressions and data types

Try running the following lines of code, placing only **one** of them in your code cell at a time and executing the cell:

```
2                       # Outputs 2
2 + 2                   # Outputs 4
2 * (3 - 1)             # Outputs 4
5 ** 2                  # Outputs 25
5 / 2                   # Outputs 2.5
5 // 2                  # Outputs 2
```

```
'Hello'                 # Outputs 'Hello'
'Hello' + 'World'       # Outputs 'HelloWorld'
'Hello' * 3             # Outputs 'HelloHelloHello'
```

The ultimate purpose of programming is to take some initial data and transform it into some final output. The data comes in many different *types*: numbers, strings of text, vectors, matrices and so forth. The type of data determines what operations can be performed on it (e.g. numbers can be added or multiplied, strings can be concatenated, matrices can be transposed etc.) The first line of code gives Python an item of data, which is the number 2. We do not ask Python to perform any processing of the data just yet, so it returns the input (2) back as is.

Note that `#` in Python means that everything that follows until the end of the line is a comment. Comments are ignored by the interpreter and you do not need to type those in to make the code work. However, good commenting is necessary to make your code easier to organize and understand. While markdown cells are there to store a general description of what you are doing, `#` can be used to annotate a specific line or block of the code.

FYI: Good and bad commenting

As opposed to some of the early programming languages like Fortran, Python was developed with user-friendliness in mind. More often than not, Python code reads as plain English and can be understood with no prior knowledge of the language whatsoever. However, overreliance on default behavior (e.g. by using the standard order of operations in arithmetic expressions instead of brackets), failure to give functions and variables (covered later) meaningful names and allowing excessive redundancy (e.g. through failing to utilize loops and functions where necessary) can easily render your Python code incomprehensible, defying its underlying philosophy.

Your commenting style is often a good indicator of the code quality. Well written code rarely needs comments to explain how it works. Instead, one should be able to infer that by looking at the code itself. The primary purpose of comments is not to explain how the code works, but why it is there. If you ever find yourself writing comments that are longer than the code they are aiming to explain or comments that answer the "how" question instead of the "why" question, there is almost certainly a better way to rewrite your code that would not require this. Our aim is not just to write code that "works", but also one that can be maintained and expanded by others in the future.

Lines 2 through 5 attempt to perform basic arithmetic operations. In Python, `+` means addition, `-` means subtraction, `*` means multiplication, `/` means division and `**` (not `^`!) means exponentiation. Spaces are ignored by the interpreter in most cases and can be used freely to organize the code.

Note that lines 1 through 4 output numbers without a decimal separator (e.g. `2` instead of `2.0`) as opposed to line 5. In Python, the presence of a decimal separator determines whether the data is of type `int` (stores integers) or `float`[2] (stores real numbers). As will be demonstrated in a moment, those data types are treated differently when operations are applied to them. In general, Python will strive to produce output of the same type as the input. Since all the input data in lines 1 through 4 are of type `int` (no decimal separators),

[2]The word `float` is an acronym for *Floating-point arithmetic* and represents the mechanism that Python (and many other languages) use to store real numbers in computer memory. As opposed to *Fixed-point arithmetic*, the amount of memory allocated for the whole and decimal parts of the number is not fixed and can be chosen by the interpreter to suit the situation at hand.

the output is provided as `int` as well. However, line 5 stands out because conventional division (`/`) is **not supported** by type `int`. As such, Python automatically converts the output to type `float` as indicated by the decimal separator.

Python follows the conventional order of operations (multiplication and division are carried out before addition and subtraction), which can be altered with parentheses as illustrated in line 3.

There exists another arithmetic operation in Python denoted with `//` that corresponds to *integer division*, i.e. division where the fractional part of the result is dropped. As opposed to conventional division, integer division is supported by type `int`, as demonstrated in line 6.

FYI: Integer division in older versions

Earlier versions of Python (`2.x`) applied integer division automatically whenever the input data are of type `int` instead of converting to `float` as modern versions do. For example, Python 2 would evaluate `5 / 2` as `2` instead of `2.5`. This somewhat counter-intuitive feature is responsible for countless mistakes among novices and experienced developers alike, resulting in its removal in version 3. It is however important to have this in mind when working with older code as it may produce unexpected results due to this subtle but important difference. When dealing with code written by somebody else, always check which version of Python it was intended to be ran with. Fortunately, the use of Python 2 has been almost completely phased out over the last few years.

The last three lines demonstrate a few operations that can be performed on strings (type `str`). Just like the decimal separator tells Python that the value should be interpreted as `float`, the quotation marks around the value tell Python that the value should be interpreted as `str` (both single and double quotation marks are allowed). When working with strings, `+` means concatenation and `*` means duplication. Operations like `/` are not supported and will produce errors when applied to strings (try it!).

Let us have a look at a few more advanced examples. As before, run the below lines one at a time in your code cell:

```
2 + 2                       # Outputs 4
str(2) + str(2)             # Outputs '22'
'2' + '2'                   # Outputs '22'
int(2.6) + float('3.5')     # Outputs 5.5
3 ** (2 + 2)                # Outputs 81
str(int(1e20 + 2))          # Outputs '100000000000000000000'
str(int(1e20) + 2)          # Outputs '100000000000000000002'
1+5j * 3                    # Outputs (1+15j)
(1+5j).imag                 # Outputs 5.0
```

Line 2 here demonstrates *typecasting*, i.e. the process of forcing Python to convert data into a particular type. In this case, number 2 is provided as `int`, but `str(2)` around it typecasts the value into a string which results in concatenation rather than addition. In line 4, a `float`, 2.6, is typecast into an `int` which results in the loss of the fractional part (i.e. 2.6 becomes 2). Then, the string `'3.5'` is typecast into a `float` and added to 2 for the total of 5.5. Python will always return a `float` when combining `int` and `float` together unless we explicitly typecast the result into something else.

Line 6 introduces scientific notation in Python, using `e`, meaning $\times 10^{\cdots}$. E.g. `1e20` means 1×10^{20}. Scientific notation turns the number into `float` similarly to the decimal separator even if the decimal separator itself is absent (`1.0e20` is however valid as well).

Lines 6 and 7 draw an important difference between `int` and `float`. When integer arithmetic is performed, it will be carried out **exactly**, regardless of the number of significant figures as long as the computational power of the system allows that. On the other hand, `float` numbers are trimmed at a certain number of significant figures so that they always take the same amount of memory. This is why in line 6, $10^{20} + 2$: the added 2 is simply beyond the maximum level of `float` precision and will be dropped. On the other hand, in line 7, 10^{20} is converted into an `int` first, forcing integer arithmetic instead, where the full precision is maintained.

Line 8 introduces complex numbers (type `complex`). In Python, the imaginary unit is denoted with `j`. For example, `1+15j` means $1 + 15i$. Note how the order of operations (multiplication is done first) applies to line 8.

Finally, line 9 emphasizes a very important feature of Python: it is an object-oriented language. This means that every item of data (be it `str`, `int`, `float` or anything else) is treated as an object. Objects can have *attributes* that can be accessed by typing out their name after a period (`.`). `imag` is an attribute possessed by any object of type `complex` and will store its imaginary part. Analogously, `real` will store the real part. Note that these two specific attributes are only possessed by complex numbers. An attempt to access them in, say, a `str` (e.g. by typing `'x'.real`) will result in an error.

4 Variables

So far we only considered isolated lines of code. Most tasks will need to be broken down into multiple steps, requiring a convenient way to save intermediate results and refer to them. With most languages, including Python, this is accomplished through *variables*. Consider the following snippet of code. This time, run every line of the snippet at the same time in a single code cell:

```
a = 2
b = 6
c = a * b + a
print(c, float(c), complex(c))
```

A variable is defined with `=` (the assignment operator). The first line defines a variable called `a` that refers to the integer 2. From now on, we can refer to this integer using the name of the variable. Equivalently, line 2 defines a variable called `b` that stores a different integer. Line 3 performs some basic operations on `a` and `b` and defines a third variable, `c`, to store the result.

The last line calls the `print()` function (more about functions later). It simply outputs to the screen one or more expressions separated by commas. In this case, it will output `c` as is, then typecast it into a `float` and a complex number and output those as well.

FYI: Automatic output

At this point you maybe wondering why the code snippet above uses the `print()` function to display the output on the screen, while previous examples did not require this and the output showed up on the

screen just fine. In fact, the `print()` function in this example is not necessary because `Jupiter notebook` automatically displays the evaluation result of the last line of code in the cell. The code above would work just as well if the last line were replaced with `c, float(c), complex(c)` instead of `print(c, float(c), complex(c))`.

The `print()` function may be necessary to display values on the screen from lines in the middle of the cell among other applications. It may also be necessary in an environment other than `Jupiter notebook` that does not display output automatically. It is also worth noting that automatic output may be suppressed by typing a semi-colon (`;`) after the last line.

Once a variable is defined, it can be redefined an unlimited number of times. A common task in many applications is swapping the values of two variables around. For example, in order to swap the values of `a` and `b`, one may intuitively (and incorrectly) attempt the following:

```
a = 2
b = 6

# Now swap the values
a = b
b = a

# Check the result
print(a, b)  # WRONG!
```

The code above leaves both `a` and `b` storing the same value 6 instead of the expected 6 and 2 respectively, because the original value of `a` was lost once we assigned it to `b` in `a=b`. A better approach to the task is to introduce a third variable to store the original value of `a` such as below:

```
a = 2
b = 6

# Now swap the values
original_a = a
a = b
b = original_a

# Check the result
print(a, b)
```

In this case, the original value of `a` is stored in a new variable which we called `original_a`. In Python, the developer is free to name their variables however they like as long as they restrict themselves to English letters (`a` through `z` and `A` through `Z` case-sensitive), digits (`0` through `9`) and underscores (`_`); however, variable names cannot start with a digit. It is considered a good coding practice to give variables meaningful names to clearly communicate their purpose.

FYI: Variable naming conventions

While both upper and lower case letters can be used in variable names, it is generally recommended to adhere to the official Python Style Guide available at https://www.python.org/dev/peps/pep-0008/. Following the Style Guide significantly improves the readability of your code in the eyes of other Python developers. Among

other things, the Style Guide recommends limiting all lines to at most 79 characters, adding single spaces around assignment operators and using lower case letters with underscores in function names.
Adhering to the Style Guide is by no means mandatory and, in fact, this document itself probably violates many of its more subtle clauses. Commercial software development companies often require their employees to follow the Style Guide of the language they use and regularly check that they do so as I happen to know from personal experience. However, I have never encountered such practice in the scientific community for better or for worse.

The current values of the variables will be preserved either until the end of your `Jupyter notebook` session or until you restart the kernel by going to Kernel > Restart or Runtime > Restart runtime. It is a good idea to restart the kernel regularly when testing your code to make sure that its successful execution does not rely on leftover values of variables that will get lost once the notebook is closed.

5 Functions

Often, the same calculation needs to be performed multiple times in the same program. Rather than repeating the same code multiple times, most languages provide means to define *functions*, or blocks of code that are given a special name, which can be used to "call" them from other parts of the code. We have already encountered a few *built-in* functions before when typecasting data (e.g. `int()`[3] is a function and so is `print()`). Functions effectively act as *subprograms*, i.e. small programs within the code of the main program that may have their own input and output. When a function is called in the code, the input is provided in parentheses immediately after the name of the function (e.g. `print(1)`). In fact, parentheses are necessary even when no input is needed. For example, another built-in function called `help` can be used to display Python documentation on the screen. The function can be called with no input as `help()` (it can also be called **with** input, in which case the specific page of documentation corresponding to the type of the object provided will be displayed. For instance, use `help(1)` in order to learn more about type `int`).

FYI: Function terminology
A plethora of terms for functions and similar objects exist in the programming jargon including *function*, *procedure*, *subroutine*, *subprogram*, *method* and more. Typically, *subprogram* is a generic term for a block of code that can be given a name and called from elsewhere in the code as a "mini-program" within the main program. *Subroutine* and *procedure* are equivalent terms describing a subprogram that has input but no output data, while a *function* is a subprogram that has both input and output. For example, one can define a *function* to calculate the square root of a number (in which case it will have both input – the number – and the output – its square root) or a *procedure* to save a given number into a file on the disk without outputting any result. While older languages such as Fortran drew explicit distinction between the two, both have equivalent syntax in Python and, in general, Python programmers refer to both as *functions* regardless of whether output is produced or not (in fact, it is impossible to have a proper *procedure* in Python as default output of type `NoneType` is automatically provided when none is available).
A *method* is a *subprogram* that is possessed by some object as an attribute. For example, in Python the

[3] To emphasize that the object being referred to is a function, in this document as well as most Python documentation a pair of empty parentheses after the function name will be used. This is merely a documenting convention and does not always imply that the function can be called like that in your code.

type `complex` has a method called `conjugate()` that calculates the conjugate value of the number, such that `(1+15j).conjugate()` returns `1-15j`.

The code snippet below serves as an example of defining a new function:

```
def square_root(x):
    result = x ** 0.5
    return result

print(square_root(1), square_root(2), square_root(3))
```

Here, we use the `def` keyword to define a new function called `square_root()`. As with variables, function names are arbitrary, but giving them meaningful names is considered a good coding practice. Right after the name of the function, we need to list every input that the function requires by name. Each input in this list is called the *argument* of the function and the argument names obey the same rules as variable names. When the function is called, Python will create variables within the function of the same name as arguments, through which the input values can be accessed. In this case, only one argument, `x`, is listed. It will be the number, whose square root needs to be computed.

Lines 2 and 3 are the *body* of the function. Each line is indented with the same number of spaces to tell Python where the body of the function begins and ends. The number of spaces is arbitrary, but the common convention is to use four. More often than not, `Jupiter notebook` will take care of indentation automatically. In the body, `x` is raised to the power of 0.5, thereby yielding its square root. The result is saved in a variable called `result`. Finally, a `return` keyword is used that serves two purposes. First, it terminates the function handing the execution back to the main program and, second, it sets the variable name that follows it, `result`, as the output (the *returned value*) of the function.

In the final line, we call the function three times and print the results, outputting the numerical values of $\sqrt{1}$, $\sqrt{2}$ and $\sqrt{3}$. Now that the code cell containing the definition of the function has ran, the function will be accessible in any other code cell until the notebook session is closed or the kernel is restarted.

Functions can have more than one argument and return more than one value. Consider the following example of a function, written to solve quadratic equations:

```
def solve_quadratic(a, b, c):
    """Solves a*x**2 + b*x + c = 0 and returns both roots"""
    d = b**2.0 - 4 * a * c
    d = complex(d)
    x1 = (-b + d**0.5) / (2*a)
    x2 = (-b - d**0.5) / (2*a)
    return x1, x2

print(solve_quadratic(1, 1, 0), solve_quadratic(1, 0, 2))
```

This function has three arguments: `a`, `b` and `c` and returns two outputs: `x1` and `x2`. Line 2 of the snippet is the so-called *docstring*. It is made of arbitrary text, enclosed in three quotation marks, that is usually pasted in the first line of the body and describes what the function does. The docstring is purely a matter of convenience and will not affect the execution of the function. It is considered good practice to provide a docstring for every

function in the code (in fact, it is the docstring that would be displayed if somebody ran `help(solve_quadratic)` to see the documentation describing the function). Docstrings are also used by software, automatically compiling documentation for the code. In line 4, `d` is typecast to a complex number. This is done on the off chance that `d` ends up being `float` and negative after line 3, as Python does not allow raising negative numbers of type `float` to fractional powers. Such feature is only supported by complex numbers; hence the typecasting call. Try running the snippet in a new code cell. On my machine, the output looks as follows:

```
(0j, (-1+0j)) ((8.659560562354934e-17+1.4142135623730951j),
(-8.659560562354934e-17-1.4142135623730951j))
```

Note that the real part of the second pair of roots is not exactly equal to zero. Instead, it evaluated to a very small non-zero number due to the finite precision of real number arithmetic in Python. Every numerical calculation performed by a computer is bound to have a small error due to limited machine precision. Sometimes Python will be clever enough to identify that and perform appropriate rounding and sometimes it will not be, resulting in the output as above.

An interesting quirk of Python that is becoming more and more common in modern programming languages is that its functions are actually **variables** of the special type `function`. This is clearly illustrated in the snippet below (must be ran after the previous snippet):

```
a = solve_quadratic
print(a(1,1,0))
```

Here, our quadratic function is assigned to another variable called `a` that is subsequently called as a function resulting in the same output as `solve_quadratic(1, 1, 0)`. Note that in the first line, no parentheses are placed after the name of the function, because the function is *not being called*. Instead, it is treated as a variable and assigned to another variable. If we ran the following code instead,

```
a = solve_quadratic(1,1,0)
print(a)
```

`a` would store the returned value of the function instead of becoming the function itself. An attempt to call it as a function would result in the "not callable" error (try it!) Since functions are treated as variables by Python, the same naming restrictions apply (i.e. underscores, letters and digits only, case-sensitive and cannot start with a digit).

6 Conditions and logical expressions

Every program considered up to now had a linear execution flow. This means that every instruction was carried out in a specific order that remained the same every time the code cell was ran. Very often, it is necessary to only execute a block of code when a certain condition holds true or to repeat executing the same block of code until a certain condition is met. Have a careful look at the following snippet, illustrating non-linear flow:

```
a = -2+1j
```

```
# Check if a is purely real
a = complex(a)
if a.imag == 0:
    print("a is purely real")
a = a.real

# Check if a is between 5 (inclusive) and 10 (exclusive)
if a >= 5 and a < 10:
    print("a is between 5 and 10")
else:
    print("a is not between 5 and 10")

# Check if a is positive, negative or 0
if a > 0:
    print("a is positive")
elif a < 0:
    print("a is negative")
else:
    print("a is neither positive nor negative")
    print("So it is probably zero")
```

Run the cell multiple times, altering the first line to assign different values to `a`. The code checks three conditions and prints different strings of text depending on whether these conditions hold or not. The general structure of a condition is given below (this is not proper code, it will not run!):

```
if <conditional expression to check>:
    code
    that
    runs
    when
    condition holds
elif <some_other_conditional_expression_to_check>:
    code
    that
    runs when
    this other condition
    holds
else:
    code
    that
    runs
    when
    none of the above conditions
    hold
```

Note that only the first `if` statement and the block of code that follows are mandatory. `elif` can be used to check a second condition when the first one does not hold (if the first condition holds, Python will simply skip the rest of the structure). The `elif` statement and the block of code that follows can be omitted or used multiple times to check three or more conditions (as before, Python will ignore the rest of the structure once it finds a condition that holds). If none of the conditional expressions hold, Python will execute the block of code that follows the `else` statement, if provided.

Returning back to our snippet, the first conditional expression, `a.imag == 0`, holds when `a.imag` (the imaginary part of `a`) is equal to 0. Note the double equals sign, `==`. The repeated character is very important as it is used to differentiate the equality operator, `==`, from the assignment operator, `=`, that we used before when defining variables. Using the assignment operator, `=`, instead of the equality operator, `==`, in a conditional expression will trigger an error (this is usually considered a virtue of Python, as many other languages would accept assignment operators in conditions, but evaluate them in unobvious ways producing unexpected results that are hard to track down). In Python, conditional expressions have the special type `bool` (short for Boolean algebra named after its pioneer George Boole).

Other operators that can be used in conditional expressions include `!=` (not equal), `>=` (bigger than or equal to), `<=` (less than or equal to), `>` (bigger than) and `<` (less than). Multiple conditional expressions can be combined together with `and`, `or` and `not`. To illustrate the concept, try running the following lines of code **one at a time**:

```
1 == 2                          # False
1 != 2                          # True
not (1 == 2)                    # True
(1 != 2) and (1 != 3)           # True
(1 == 2) and (1 != 3)           # False
(1 == 2) or  (1 != 3)           # True
(not (1 == 2)) and (1 != 3)     # True
```

Line 1 evaluates to `False`, because 1 is not equal to 2. Line 2 evaluates to `True` for the exact same reason. `not` is a unitary operator, meaning that it acts on a single expression that follows it. Its role is to turn `True` into `False` and `False` into `True` (i.e. it inverts the value). In line 3, `1==2` on its own would evaluate to `False`. With the `not` operator in front of it, it evaluates to `True` instead. Both `and` and `or` are binary operators, as they join two different conditional expressions into one. `or` evaluates to `True` when at least one of the expressions is `True`, while `and` evaluates to `True` only when both expressions are `True`. Finally, multiple operators can be combined together and the order of operations can be altered with brackets as demonstrated by line 7.

It is interesting that even non-conditional expressions can be used in `if` and `elif` statements. Consider the following example:

```
if 42:
    print("Python thinks that 42 is True")

if 'UFOs':
    print("Python thinks that UFOs are True")

if not 0:
    print("Python thinks that 0 is False")

if not "":
    print("Python thinks that an empty string is False")
```

When an expression of type that is not `bool` is encountered in an `if` or `elif` statement, it will be typecast into one (this kind of automatic typecasting is sometimes called *type-juggling*, emphasizing its unobvious nature and wide room for confusion and mistakes). The rules are that all non-zero numbers evaluate to `True` and so do all non-empty strings. Any data

type can also be typecast into a `True` or `False` value manually with the `bool()` typecasting function.

7 While-loops

Loops allow running the same block of code multiple times until a certain condition is satisfied. This differentiates them from functions, which run only when called. Python supports two different types of loops: *while-loops* and *for-loops* (the third common type of loops in programming languages, *until-loops*, is not supported). Consider an example while-loop below:

```
1 i = 1
2 while i <= 10:
3     print(i)
4     i += 1
```

The block of code after the `while` statement (lines 3 and 4) will run again and again until the conditional expression after `while` is satisfied. Every run of the loop is known as *iteration*. In every iteration, line 3 will print the current value of `i` and line 4 will increase the value of `i` by 1 (`i+=1` is a shorthand for `i=i+1`; `i-=1`, `i*=1` and `i/=1` will also work, but structures like `i--` and `i++` are not supported, despite being very common in many other languages). Ultimately, the loop will run 10 times, printing numbers from 1 to 10.

If the condition of the loop is never satisfied, the loop will continue iterating indefinitely and the program will "hang". This behavior can be recreated if line 4 is removed (try it!). When this happens, a locally ran notebook may become unresponsive, consuming arbitrary amounts of memory and processing power. The evaluation of the cell can be stopped by choosing Kernel » Interrupt or Runtime » Interrupt execution in the main menu (assuming you still have control of the interface).

Within the body of the loop, two additional statements are available: `continue` terminates the current iteration of the loop and skips to the next one, `break` exists the loop right away. An example application of these statements is in the code snippet below. Try running the code and explaining any output it produces.

```
i = 0
while i <= 10:
    print("Iteration started")
    i += 1
    print("i =", i)

    if i == 5:
        continue

    if i == 7:
        break

    print("Iteration finished")

print("End of loop")
```

8 Lists and for-loops

A list is a special data type (type `list`) that can store multiple values of any data type (including other lists!) at the same time. Consider the example below:

```
my_first_list = [1, 2.5, "Area 51", 1+5j]
my_second_list = [1 == 3, 2 != 4]

print(my_first_list)                    #  [1, 2.5, 'Area 51', (1+5j)]
print(my_first_list + my_second_list) #  [1, 2.5, 'Area 51', (1+5j), False
                                       , True]
print(my_first_list[2])                #  Area 51

i = 0
while i < len(my_first_list):
    print(i, ':', my_first_list[i])
    i += 1
```

Lines 1 and 2 define two different lists and save them in two variables called `my_first_list` and `my_second_list`. Lists are defined by comma-separating their values and enclosing all of them in `[]`. The first list contains 4 elements: an `int`, a `float`, a string and a complex number. The second list is made of two conditional expressions, one of which evaluates to `False` and the other one to `True`.

Lines 4 prints the first list, while line 5 adds the two lists together (for lists, `+` results in concatenation) and prints the result. Line 6 demonstrates how an individual element of the list can be accessed, where `2` is the index of the element. Indexing begins with 0, so `[2]` refers to the third element of the list. An attempt to access a non-existent element of the list (e.g. `my_first_list[4]`) will trigger an error. Negative indices can be used to refer to elements in a reversed order. For example, `my_first_list[-1]` will access the last element of the list, `my_first_list[-2]` will access the second last element etc.)

Finally, lines 8 through 11 iterate over every element of the list in a while-loop and print all of them as well as their indices. `len()` is a built-in function that returns the total number of elements in the list (in our case, 4).

Although it is perfectly valid to use while-loops to iterate over lists, Python offers a different type of loops that are designed specifically for this purpose. They are called *for-loops*. Consider the same program as above rewritten as a for-loop instead of a while-loop:

```
my_first_list = [1, 2.5, "Area 51", 1+5j]

for element in my_first_list:
    print(element)
```

A `for...in...:` statement expects a list to be iterated over after `in` and a variable (previously defined or not) where the iterated element of the loop will be placed in each iteration (this variable is called iterant) after `for`. `continue` and `break` can be used in for-loops just as well as in while-loops.

In the example above, we do not have access to the index of the element being iterated. If this is necessary, refer to the example below instead:

```
my_first_list = [1, 2.5, "Area 51", 1+5j]
```

```
for i, element in enumerate(my_first_list):
    if element == "Area 51":
        my_first_list[i] = "Aliens"

print(my_first_list)
```

The snipped above searches the element `"Area 51"` and replaces it with `"Aliens"`. It is imperative that `my_first_list[i]` is being overridden and not `element`, as `element` is a copy of the iterated element and not the element itself.

9 Dictionaries

Lists are not the only data type in Python capable of storing multiple elements at once. In general, such variables are called *iterable*. In addition to lists, they most importantly include *tuples*, *dictionaries*, *sets* and `NumPy` *arrays*. The difference between tuples and lists is somewhat subtle and has to do with mutable and immutable data types that you can read more about in the official documentation.

Dictionaries (type `dict`) are similar to lists; however, instead of numeric indices the elements of a dictionary are strings. A dictionary may be defined as follows:

```
my_first_dict = {
    'my_first_element': True,
    'my_second_element': 5,
    'my_third_element': ['This', 'Is', 'A', 'List'],
}

# Print one of the elements
print(my_first_dict['my_third_element'])
```

The ability to give names to individual elements allows for much better organization of your data. Just like lists, dictionaries can be iterated over with a for-loop:

```
for key in my_first_dict:
    print(key, my_first_dict[key])
```

The iterant variable (in this case, `key`) will be receiving the **index** of the element on each iteration and not the value. The value may be obtained from the original dictionary using that index.

Dictionaries are **not** ordered, which means that the order in which the elements are iterated may differ between sessions, computers and executions and must not be relied upon[4].

Sets are an iterable data type that cannot contain duplicating elements. Typecasting a list into a set results in their removal:

```
my_list = [1, 2, 3, 3, 3]
my_set = set(my_list)
print(my_set)  # {1, 2, 3}
```

[4]The default Python installation comes with the `collections` library that may be loaded. It provides a few additional iterable data types including `OrderedDict` that is equivalent to the default `dict` but maintains a fixed consistent order. See the official documentation at https://docs.python.org/3/library/collections.html

NumPy arrays are discussed in the following section.

10 NumPy

So far, we have only considered native built-in features offered by vanilla Python. The true power of this language in the field of scientific computing lies in external libraries, containing predefined functions for performing specific tasks. Of those, NumPy is perhaps the most famous one, encompassing a suite of tools for linear algebra, statistical analysis, differential equations and so much more. The module comes with the standard distribution of Anaconda and needs not be installed separately. However, we have to tell Python that we want to use this library in our code with the import keyword.

```
import numpy as np

print(np.pi)          # The circle constant
print(np.e)           # Euler's constant

# Trigonometric functions
x = 5
print(np.sin(5), np.cos(5), np.tan(5))

# Statistics
my_list = [1, 5, 25, 50]
print(np.mean(my_list), np.std(my_list))
```

Line 1 tells Python to load the NumPy library and make its content accessible in our code. Here, numpy is the name of the library[5] and np is an arbitrary alias that can be used instead of numpy throughout the code to make it shorter. Any alias can be chosen, but np is the standard convention. Once imported, np acts as a variable. It has attributes (such as pi and e), similarly to how complex numbers have imag and real. It also has functions (technically, methods), such as sin(), cos() and many-many more. Both attributes and functions are accessible through typing np followed by a period (.), followed by the name of the desired method or attribute. Methods must be followed by a pair of parentheses with arguments to tell Python that we want them called.

NumPy has extensive online documentation, covering most of its features. If the purpose of any of the functions in the snippet above is not clear, look up their exact specification in the official documentation at https://numpy.org/doc/stable/.

The central concept of NumPy is NumPy arrays. They are a data type that can hold multiple values at the same time, similarly to Python's native lists. The snippet below summarizes a few ways in which a NumPy array can be created:

```
print(np.ones(5))                    # Create an array of five "ones"
print(np.zeros(6))                   # Create an array of six "zeros"
print(np.random.uniform(0, 1, 10))   # Create an array of ten random
                                     numbers between 0 and 1
```

[5]The official name of the library is spelled NumPy with capitalization as indicated. However, the *machine name* of the library, i.e. the name by which it is referred to in code, is spelled in lower case only. This is in fact a standard convention across most Python packages because loaded variables are treated as variables as well and follow the same naming conventions

```
print(np.array([1, 3, 6]))              # Create an array from a list
print(np.linspace(1, 10, 19))           # Create an array of 19 evenly spaced
                                          numbers between 1 and 10

# Arrays can be saved in files for future use and loaded from them
a = np.random.uniform(0, 1, 10)         # Generate some random numbers
np.savetxt('random.dat', a)             # Save them in the file random.dat
b = np.loadtxt('random.dat')            # Load them back from the file into b
print(b)                                # Print them
```

The advantage of NumPy arrays over lists is in automatic iteration of any applied operations. For example, when a number is added to a NumPy array (using +), it will be automatically added to every element of that array. When two arrays of equal sizes are multiplied together, every element of the first array will be automatically multiplied by the corresponding element of the second array and so forth. With lists, similar manipulations must be performed in a for-loop or a while-loop for every element individually. The code snippet below exploits this feature of NumPy arrays to integrate $\sin(x)$ from $x = 0$ to $x = \pi$:

```
x = np.linspace(0, np.pi, 1000)          # Array of 1000 evenly spaced
                                         numbers between 0 and pi

# sin(x) will be "sampled" at these points

y = np.sin(x)                            # Compute sin(x) for all 1000
                                         numbers (no loops necessary)
dx = x[1] - x[0]                         # Difference between adjacent x
                                         values
print("Integral:", np.sum(dx * y))
```

In fact, NumPy already has a function to carry out such integration called `np.trapz()`:

```
x = np.linspace(0, np.pi, 1000)
y = np.sin(x)

print("Integral:", np.trapz(y, x))
```