Python for a non-technical audience

Foreword

This book started out as a series of lecture notes for ESS345H1 Computational Geology, but gradually morphed into a textbook for python. The aim of this text is not to provide a comprehensive python introduction but rather to provide a guided approach to coding. The pdf version is meant to be used as a reference while working on the interactive chapters.

I am indebted to Tianshi Liu who proofread the text and provided invaluable commentary.

Contents

1 Introduction

> Coding is a fun thing to do. It's captivating, maddening, powerful, and deeply weird.
>
> Coding teaches - and also rewards - dogged persistence. It shows us that errors are a normal part of life and that careful, patient work can repair them.
>
> Clive Thompson

Most programming books are written for people who already know at least one programming language, and typically have considerable formal training in mathematics, physics or engineering. As such, they introduce the grammar of their respective language in the context of math, physics, or another computing language.

In my experience, the biggest hurdle in teaching programming is not to learn a computer language but to use the language to solve a given problem. In other words, how do I decompose a problem into smaller parts which can be solved by iteration and comparison. This text aims to first introduce the basics of the python programming language, and then uses simple exercises to build the problem solving skills necessary to create a more complex program.

The goal is to enable students to:

- write and read simple python programs
- provide some familiarity with error messages and how to debug them
- know where to get help (built-in help system, stackoverflow)
- know how to ask for help
- provide some guidance on how to solve problems
- provide some cookbook recipes for solving typical workflows (i.e., create and annotate a graph, import data from excel, do some statistical analysis.

This course is built around the Jupyter Notebook system, and all chapters are available as fully interactive notebooks which can be used to experiment with. Programming is a skill that is best acquired by doing.

> Just remember that your programming education is your responsibility, even when you take a class. A course will provide a framework for acquiring a grade and credit at the end of the term, but that framework doesn't limit you in your learning. Think of your time in the class as a great opportunity to learn as much about the subject as possible beyond any objectives listed

in the course syllabus. —- V.A. Spraul, Thinking like a programmer

1.1 Getting the tools

In order to run a python program, you will need two things: A) A text editor so that you can write python code, and B) a python-interpreter that can execute your python code. Often these components are packaged into a so-called integrated development environment (IDE). For this course, we will use the Jupyter Notebook Environment. These notebooks allow us to mix explanatory text and live computer code (and its results); they also provide us with a way to save our work in PDF format; Most commercial data analysis is now done in notebook format, and last but not least, we can use a cloud service which runs in the web browser, so there is no need to install software locally.

1.1.1 Using the Syzygy cloud

Syzygy is a strange word, isn't it? Look it up it does have meaning. The notebooks and assignments will be made available in a just-in-time fashion. If you use the following link, updates will automatically applied upon login. https://utoronto.syzygy.ca/jupyter/hub/user-redirect/git-pull?repo=https://github.com/uliw/PNTA-Notebooks&urlpath=tree/PNTA-Notebooks/&branch=main

This automated process sometimes has issues. See the "Interacting with Jupyter Chapter" for more information, and how to login without updating.

2 Of Folders and Directories

This module is available as interactive Jupyter notebook in your project home directory as

- `./PNTA-Notebooks/file_system/File_System_Exercise`
- `./PNTA-Notebooks/File_System_Exercise_assignment`

3 Getting to know your Jupyter Notebook

This module is available as three interactive Jupyter notebooks in your project home directory as:

- `./PNTA-Notebooks/Interacting_with_Jupyter/Interacting_with_jupyter-1`
- `./PNTA-Notebooks/Interacting_with_Jupyter/Interacting_with_jupyter-2`
- `./PNTA-Notebooks/Interacting_with_Jupyter/Interacting_with_jupyter-assignment`

To create a computer program, we need the following ingredients:

1. A text editor that allows us to write computer code.
2. A program that executes our code
3. And a way to see the results of our code

There are many ways to achieve the above, but typically most people use what is commonly referred to as an "Integrated Development Environment" or IDE. There is an enormous variety of IDE's available, but most require a local installation. For this course, we will use a web-based IDE, called `Jupyter`, widely used in the sciences and business environments.

In this unit, you will learn:

1. How to access a Jupyter server
2. How to open a Jupyter notebook

Jupyter servers can run locally on your computer, are available commercially for a fee, and some are even available for free (e.g., googles Colab). For this course, we will use a Jupyter server which is provided by Compute Canada and is accessible with your UTorID.

There are different ways to access your Jupyter account. In this course, we will use this somewhat lengthy URL: `https://utoronto.syzygy.ca/jupyter/hub/user-redirect/git-pull?repo=https://github.com/uliw/PNTA-Notebooks&urlpath=tree/PNTA-Notebooks/&branch=main`

If you look carefully, you see that the first part specifies the Toronto jupyter server, which is followed by a redirect to a program called git-pull with reference to a GitHub repository I use for the course materials. This line will first authenticate you, then download or update the course materials, and then start your jupyter interface.

So if you click on the above link, it should open a browser window with the link (see Fig. 3.1) (BTW, Sometimes this process fails with an error message. See below how to resolve them.)

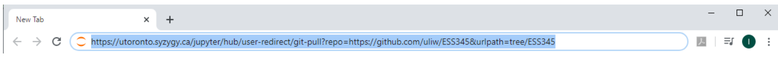


Figure 3.1: This is what you will briefly see in your browser status line

and this will then take you to your UtorId login screen (Fig. 3.2)

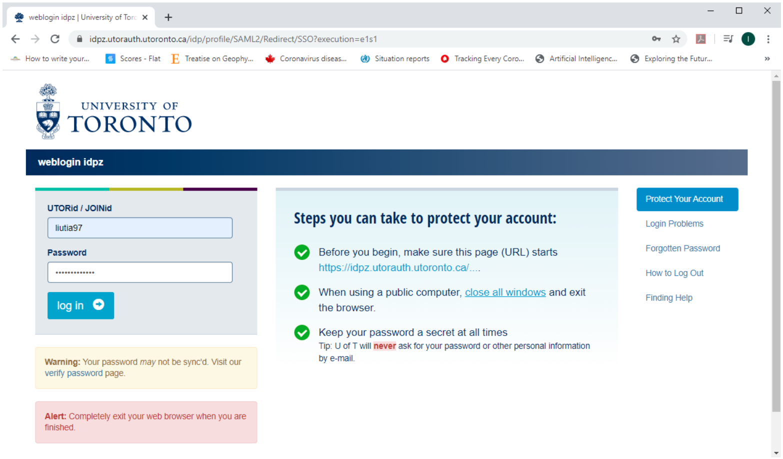


Figure 3.2: The UTorID login screen

and finally you should see a screen like the one shown in Fig. 3.3 (but without the `My_Stuff` directory).

For now, don't click on anything, but let's have a look around. If you are unable to see any of the following elements, ask your neighbor, or speak up!

You can see the Jupyter label in the top right corner, the `Logout` and `Control Panel` buttons in the top right corner. Next comes a line with tabs for `Files`, `Running`, `Clusters`, followed by a line with buttons for `Upload`, `New`, and `Reload`. This is followed by a table.

The table header starts with a button that lets you select all files or only specific file types. Next comes the blue breadcrumb, which indicates your position in the file system tree. It should read `PNTA-Notebooks`. The next button lets you sort the table by `Name`, the date when the file was `Last Modified`, or the `File Size`.

The table rows contain a variety of entries, where each entry type is signified by a different icon.

1. The first one is a folder symbol with no name but two dots. This symbol represents the folder that contains the `PNTA-Notebooks` folder (i.e., one up). If you click on it, you will go to your home folder. You can return to the `PNTA-Notebooks` folder by clicking on the `PNTA-Notebooks` folder symbol.

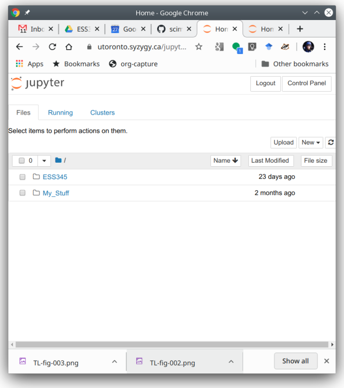


Figure 3.3: The home screen of your Jupyter session. Note that yours might look slightly different.

2. Once you are back in the `PNTA-Notebooks` folder, the following file you see is the syllabus. This file shows the notebook icon and also has the `.ipynb` extension, which denotes all jupyter notebooks. If the notebook is open and active, this icon will be green. Otherwise, it will be gray.
3. The next two icons show a paper symbol, which denotes a text file. You can click on these files and read them. If you are done reading, click on the jupyter icon in the upper left corner to return to the folder view.

Let's make some changes:

1. go to your home directory (i.e., one up from PNTA-Notebooks) and in the upper right corner, select `New` and `Folder` see Fig. 3.4
2. This will create a new folder called `Untitled Folder`. Select the check-mark left of the folder name, then click rename in the upper left corner, and rename the new folder `My_Stuff`. Your screen should now look like Fig. 3.5 This new folder will hold all your own jupyter notebooks.

3.1 Jupyter Connection Problems

- If this is the first time you trying to download the course materials
  - Sometimes, the process which downloads the course data from the github repository fails. In this case, do the following
  - Close all windows which point to https://utoronto.syzygy.ca/
  - Log in to https://utoronto.syzygy.ca/ using this link (not the one in the textbook)

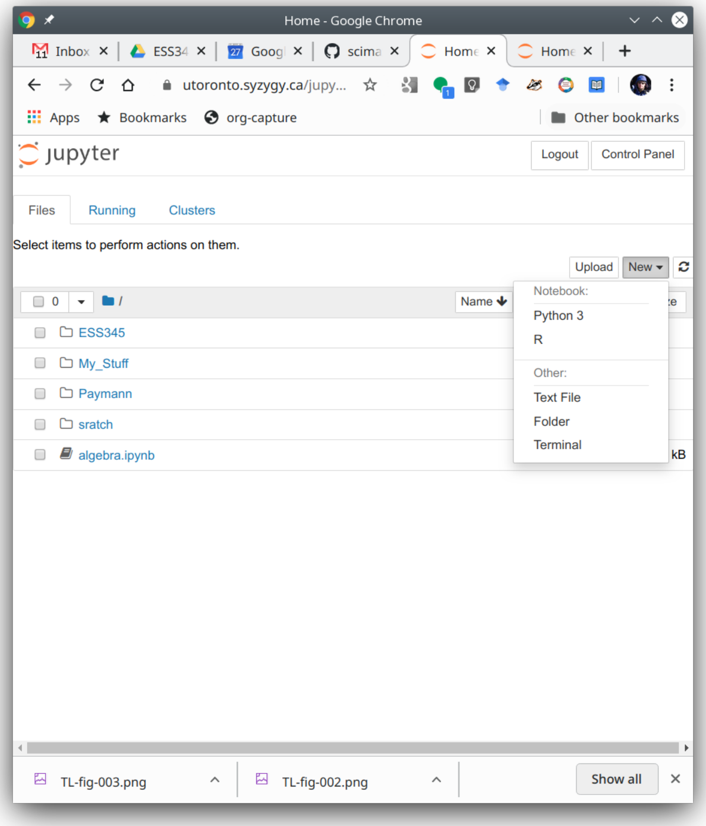


Figure 3.4: Creating a new folder. Note that we do this from the home directory, i.e., one up from `PNTA-Notebooks`

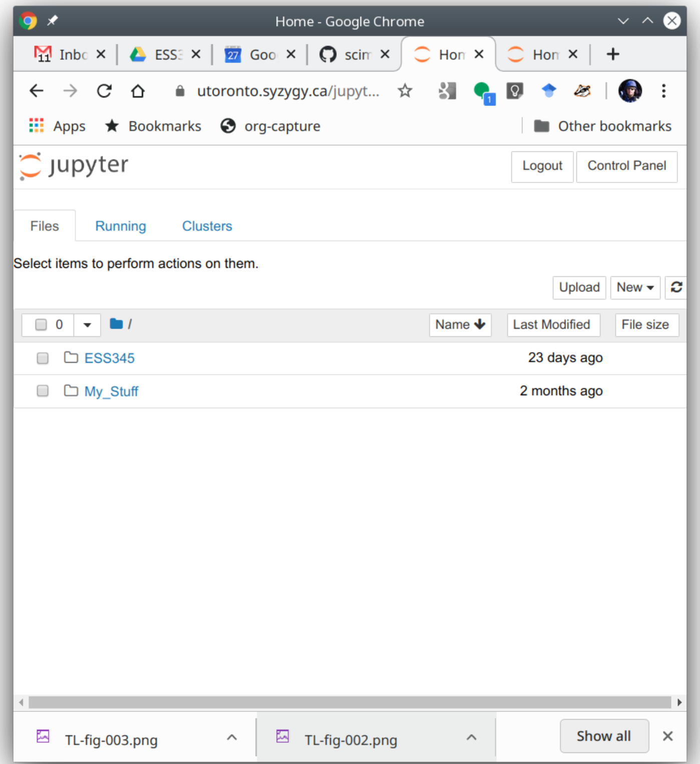


Figure 3.5: Your home directory should now look like this.

  - If you see the `PNTA_Notebooks` directory, open it and open the `reset_-git.ipynb`. Select the code cell, and hit `sift+enter`. This should take care of any connection issues.
  - If the `PNTA_Notebooks` directory is missing:
    * This will open a command line terminal
    * Select "New" -> "Terminal" on the right hand side
    * Enter the following command at the prompt:

git clone https://github.com/uliw/PNTA-Notebooks

- Hit enter
- This will download all the PNTA-Notebooks course materials
- Now close your syzygy session
- Once you logged out of everything, you should be able to use the link in the textbook
- Once you logged out of everything, you should be able to use the link in the textbook

3.2 Recap

In this module, you should have learned:

1. How to access the Jupyter server
2. How to create and name folders (directories) on the Jupyter server
3. How to resolve Jupyter connection and github update problems

Since Jupyter notebooks will be the primary tool we use in this course, and the dominant format for your assignment submissions, we will explore their use in greater detail in the following.

A Jupyter notebook has three principal elements see Fig.3.6

1. Text areas (often referred to as cells). You can edit these text2 cells.
2. Code blocks (or code cells). These contain computer code that can be executed. You can edit and execute these cells.
3. Result Blocks, which contain the output produced by a code block. Results blocks cannot be edited. However, you can select and copy them.

If you are still reading this as a pdf file, please switch to the notebook version of this very text by following this link https://utoronto.syzygy.ca/jupyter/hub/user-redirect/git-pull?repo=https://github.com/uliw/PNTA-Notebooks&urlpath=tree/PNTA-Notebooks//Interacting_with_Jupyter/Interacting_with_jupyter-2.ipynb&branch=main

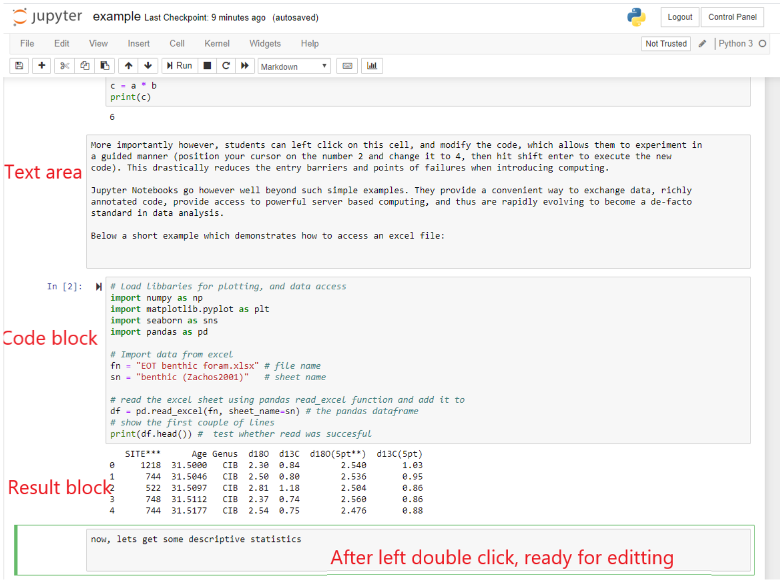


Figure 3.6: An example of a Jupyter notebook (.ipynb file) including text areas, code blocks and result blocks. After double left click on the cell, the cell will be ready for editing (and the bar will change to green)

Now, you can follow the instructions below:

If you do a single left click with your mouse on this text, you will see that this cell is now drawn with a blue border. This indicates that the cell has been selected. (see Fig.3.7).

If you double-click on this cell, you will see that the cell background changes to gray, rendering the text in a typewriter font. This indicates that you are now in editing mode. You can leave the editing mode any time by hitting `Shift-Enter`.

Also, most of the usual keyboard shortcuts will work:

- Ctrl-z = undo
- Ctrl-c = copy
- Ctrl-x = cut
- Ctrl-v = insert

3.3 Potential Problems

Sometimes, you may lose the connection to the Jupyter server. This is indicated by the red `Not Connected` icon in the status bar (see Fig.3.9). This is usually caused by a firewall rule. Please speak to a TA or the instructor to resolve this issue.

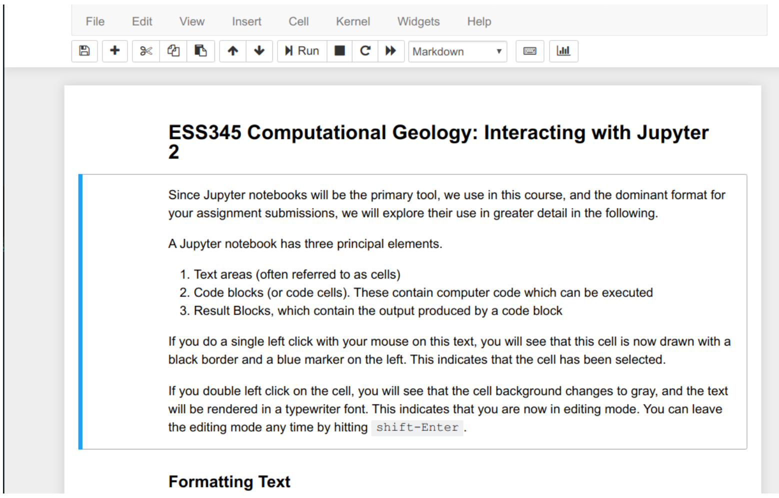


Figure 3.7: A single click will select a notebook cell, which is indicated by the blue border to the left.

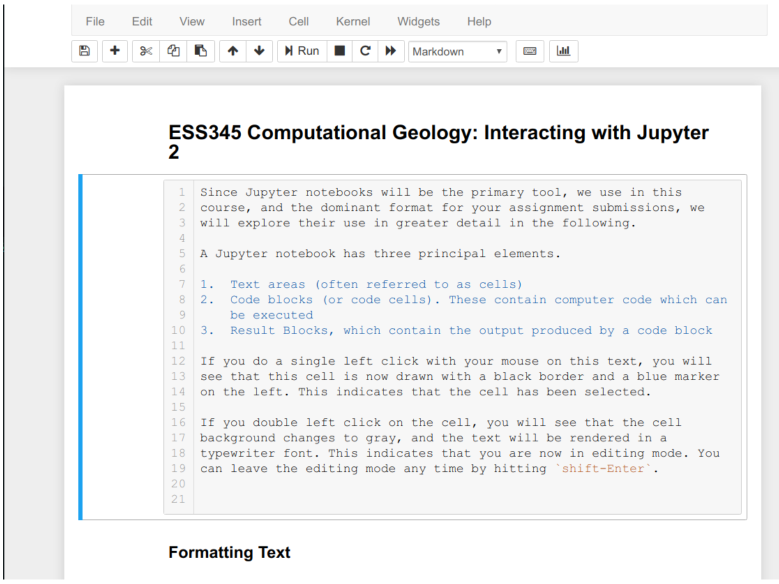


Figure 3.8: A double click on any cell it will activate the editing mode. You can leave the editing mode any time by hitting `shift-Enter`.

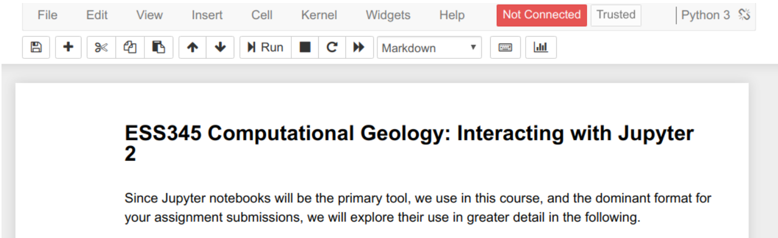


Figure 3.9: The red `not connected` icon indicates that you lost connection to your Jupyter Server. This is usually caused by a firewall rule. Please speak to a TA or the instructor to resolve this issue.

3.4 Formatting Text

Before exploring how-to-use Jupyter notebooks to run python code, let's explore how to format the text in the text cells. Create a new notebook in your `My_stuff` folder. You can do this via the drop-down list called `New` in the upper right corner, and then select `Python 3`, and rename the new file. These steps should now pose no problem to you. If they do, please speak up!

Once your new notebook is open, it will look like Fig. 3.10. Notice the text `In [ ]:` on the left side. This tells you that this is a code cell that expects python code. So before we start playing with text, we need to change the cell type (see Fig. 3.11)

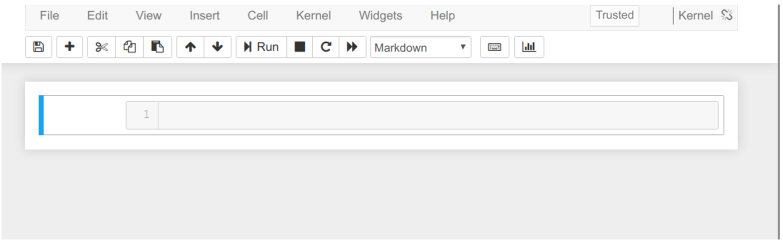


Figure 3.10: The default view when opening a new notebook. Note the text `In [ ]:` on the left side. This tells you that this is a code cell that expects python code.

Use the dropdown menu to change the cell type to `markdown`.

Once you have changed the cell to markdown, you will note that the `In [ ]:` text on the left is now gone.

Next, copy the following lines into your new notebook cell:

3.5 This is a second-level heading

This is **bold** text

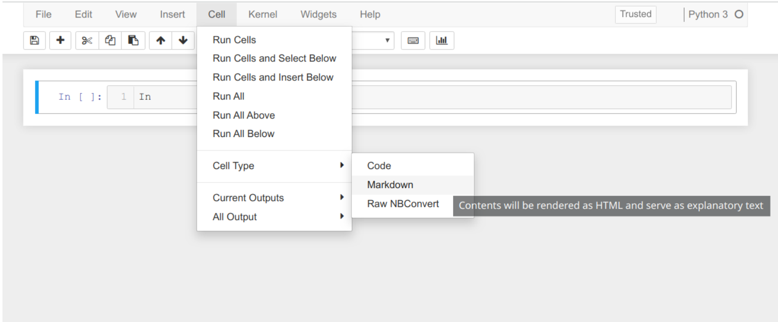


Figure 3.11: Use the dropdown menu to change the cell type to `markdown`

1. This is a simple list
    - First list entry
    - Second list entry
2. This is the 3rd entry in the enumerated list and has no other purpose than just being the 3rd entry.

You will notice that the formatting has been lost. Go back to these instructions, activate editing mode, and explore how to create the formatting you see (see Fig.3.12) . Now go to your new notebook, and apply the `mark down` syntax in your own notebook. . Remember that you can leave the edit mode any time by hitting `Shift+Enter`.

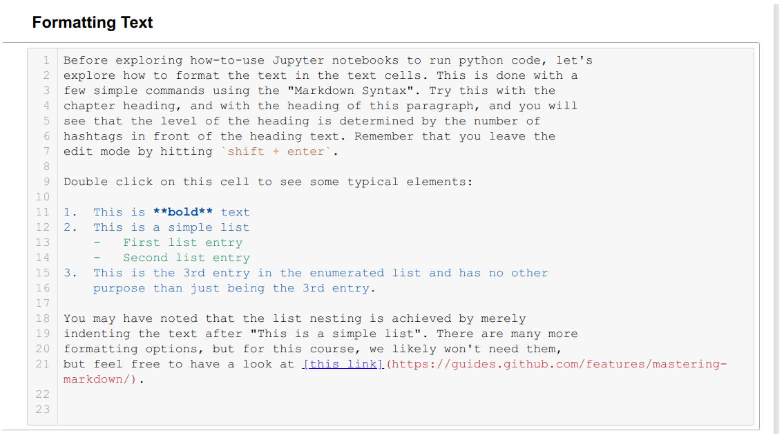


Figure 3.12: Here, you can see how to format your text with `mark down` syntax

You may have noted that the list nesting is achieved by indenting the text after "This is a simple list". There are many more formatting options, but for this course, we likely won't need them, but feel free to have a look at this link.

Last but not least save your edits by clicking on the floppy disk icon in the upper left corner. This will create a checkpoint so that you can return to this version at any time!

3.6 Adding Code cells

Now let's try adding a code cell below the text cell and enter a trivial statement like `1 + 1` and hit `Shift+Enter` and you should see the result displayed below the code cell (hopefully, it is 2). Go ahead and edit your code cell (e.g., 1+3) and hit `Shift+Enter` again. The result should change accordingly (see Fig.3.13)

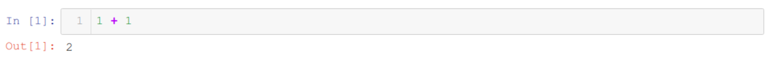


Figure 3.13: Example of a trivial python statement

Note that your edits are not auto-saved! You need to explicitly use the floppy disk icon (leftmost, directly under `File`) to save your work!

3.7 Downloading your notebook

All of your assignments will have to be submitted on Quercus. In order to mark your assignments, you need to submit a pdf copy, as well as the actual notebook file. This is easily done via the `File` dialogue (see Fig. 3.14)

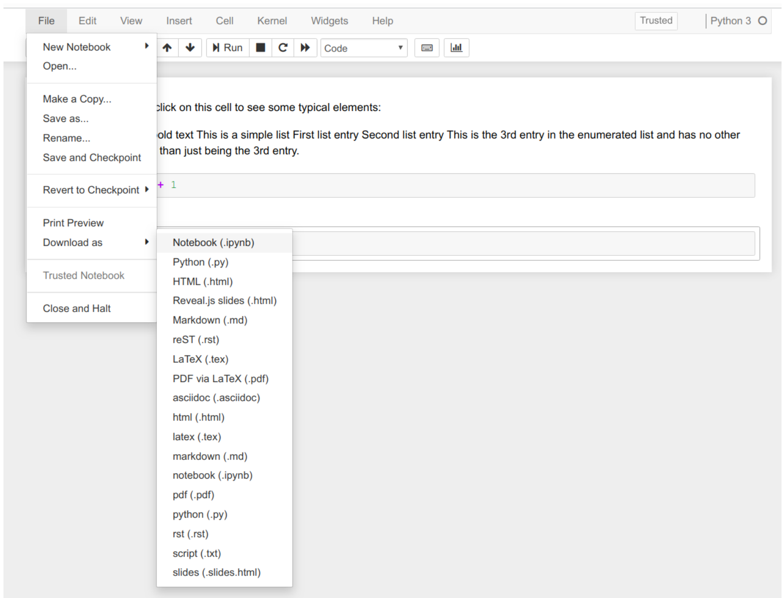


Figure 3.14: You can download the notebook in a variety of formats.

3.8 Recap

In this module, you learned how-to:

1. Create a notebook
2. Save a notebook
3. Download a notebook
4. Create text cells
5. Edit and format text cells
6. Create code cells
7. Execute code in a code cell
8. Add a new code cell

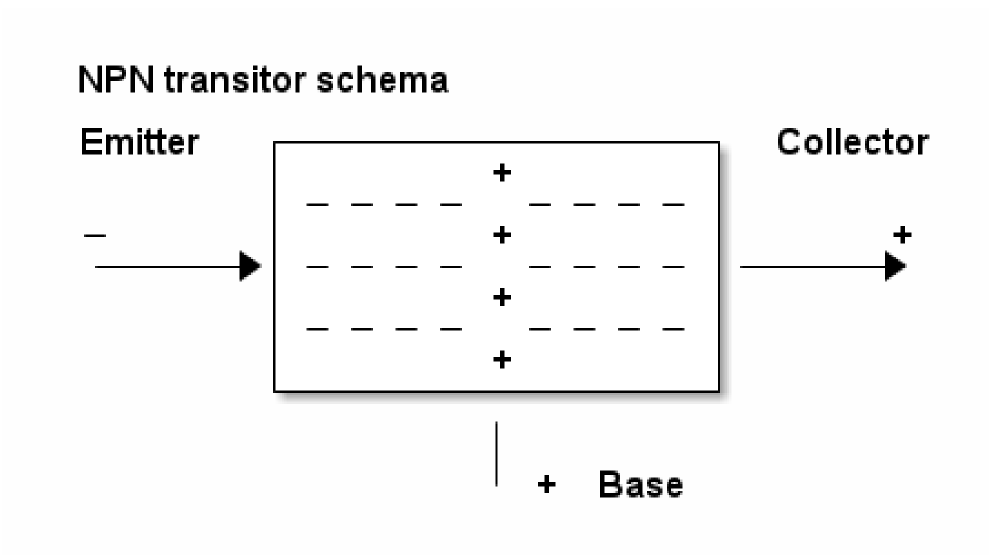
4 Basic elements of a program

4.1 How-to store a number in a computer

This module is available as interactive Jupyter notebook in your project home directory as

- `./PNTA-Notebooks/Storing_Numbers/storing_numbers`

On its most basic level, computers are made of simple transistors. Transistors are nifty devices that allow us to control the flow of a large current with a small current. At their heart, they consist of two layers of thin silicon with additional electrons in the crystal lattice and a thin layer of silicon missing some electrons (or vice versa). This thin layer acts as an insulator and prevents any current flow (see the sketch below). However, if we add a small current to this layer, we will move some electrons into the thin layer, which will take the space of the missing electrons in the crystal lattice, and now the current can flow between the two outer layers until we remove the control current again - hence the term semiconductor. See this youtube link for a detailed explanation.



For our purposes, a transistor can be thought of as a simple switch that either lets an electrical current pass or not. As such, the transistor can reflect the numbers 0 (nothing passes) or 1 (everything passes). So with a single transistor, we can count two numbers

(0, 1). Now, if we add another transistor, we should be able to count to four. However, since each transistor is either on or off, we need to express numbers in a binary system (i.e., a number system, which only knows two numbers, zero and one).

Let's consider how we count. We have 10 fingers that we can use to count from 0 to 9. However, what happens if we need to count past 9?

100-999	10-99	0-9	Name
		.	.
		.	.
		8	eight
		9	nine
	1	0	ten
	.	.	.
	9	9	ninty nine
1	0	0	one hundred

So each time we run out of fingers (counting apples, e.g.), we add a number to the left, which tells us that we already have ten (or twenty, thirty) apples, which we have to add to the current number of fingers. So we are counting in batches of 10. A more scientific way to say this is counting to the base of 10. We can rewrite the above table as

10^2	10^1	10^0	Name
		.	.
		.	.
		8	eight
		9	nine
	1	0	ten
	.	.	.
	9	9	ninty nine
1	0	0	one hundred

where you would have to multiply the table header with the column value, e.g.,

```
10^0 x 8 = 1 x 8 = 8
```

let's try this with row 7

```
9 * 10^1 + 9 * 10^0 = 90 + 9 = 99
```

4.1.1 The binary system

So how would we count if we only had one finger? Zero is easy (no finger), one is one finger, but what about 2 and 3, etc.?

Similar to the above, we can do a table, but this time to the base of 2/

2^2	2^1	2^0	Name
		0	zero
		1	one
	1	0	two
	1	1	three
1	0	0	four
1	0	1	five
1	1	0	six
1	1	1	seven

let's look at the last line in this table, which is similar to the decimal system, can be written as

```
1 * 2^2 + 1 * 2^1 + 1 * 2^0 which is equal to
1 * 4 + 1 * 2 + 1 * 1 which is equal to
4 + 2 + 1 = 7
```

so with three transistor memory cell , we can store numbers from 0 to 7. If we had eight transistors, we could store numbers from zero to $2^8 = 255$. Rather than saying a memory cell is eight transistors wide, it is common to say 8-bit (or 1 byte) wide.

Modern computers have memory cells that contain many more cells, typically 64 bits (or 8 byte), which allows them to store numbers from 0 to 2^{64}.

The above works well for positive numbers with no fractional part (i.e., 1, 2, 3, 4, etc.). These numbers are called unsigned integers. However, what about storing negative numbers?

To store a negative integer number, we have to sacrifice one bit, which will now indicate whether the number is positive or negative. These numbers are called signed integers; with those, we can count from -2^{63} to +2^{63}.

Numbers that have a fractional component are called floating-point numbers (e.g., 1.2345). To store these numbers, we need a way to express them in our binary system. The trick is to decompose them into two integers

```
1.2345 = 12345 x 10^{-4}
```

You can immediately see that such a number needs double the memory of an integer number.

Some computing languages are very particular about these things, e.g., you need to declare that a given memory area will only hold signed integer values. Any attempt to store another number type will result in an error message.

Python is pretty easygoing about this. I.e., if you write `a=1.23` python will know that you are storing a floating-point value. So why do we bother with this in the first place? Imagine you are dealing with very small numbers (say environmental pollution in the ppb range). Then it is important to understand that this number may not be accurately reflected in your code. This is one of the more common problems when you work with computer models.

It is possible to change the type of a number explicitly; try the following example.

```
a = 20/3  # the result of this division is a floating-point number
print(a)  # print the value of a
int(a)    # convert a into an integer number
```

There are two things to note here: A) the use of comments. Everything behind the hashtag is ignored but helps to explain what is going on; B) casting a float to an integer differs from rounding the value! (If you round 6.666 to the nearest integer, you will get 7!)

Summary

In this section, you should have developed some sense of

- how numbers are stored electronically
- how to count if you only have one finger
- how to convert from binary notation into decimal notation
- that it matters what kind of numbers you are storing
- the difference between casting and rounding

4.2 Variables which store a single value

This module is available as two interactive Jupyter notebooks in your project home directory as:

- `./PNTA-Notebooks/Data_types/Variables/variables`
- `./PNTA-Notebooks/Data_types/Variables/variables_assignment`

A computer is a device that can store and manipulate data (information). Before we learn the basics of data manipulation, we need to understand how data is stored.

On its most basic level, a computer resembles a library (you have been to the library, right?). There are vast amounts of storage space, a catalog where we can look up where a specific book is stored, a librarian who will move books around and do other maintenance tasks, and a user who manipulates the information in those books, and who might writes a new book that eventually goes back into the library.

Books can be found using a rather specific way of writing their address in the library storage which, e.g., could read `QA76.17.C4672012`.

Similarly, the information in a computer can be retrieved by using equally cryptic schemes, e.g., `0x5560b128f480`. The `0x` at the beginning tells us that this is a number to the base of 16, which after conversion into our 10-based system indicates that the respective data is stored in the 93873777472640$^{\text{th}}$ memory cell. It is cumbersome to keep these alphanumeric sequences in mind. This is where variables come into play. Rather then saying store the number 4 at location `0x5560b128f480` we can simply say "assign the value of 4 to the variable a". Let's try this and execute the following statement doing a left-click in the following cell, and hit **`shift+enter`**

```
a = 4
```

This will produce no output, but you will have assigned the value of 4 to the variable `a`. Let's verify that this assignment did succeed. We can do so by asking the python interpreter to print the value of a. Execute this cell, and you should see the result printed below this cell

```
print(a)
```

Go ahead and try this

```
a=4
b=5
c=10
(a/b)*c
```

So far, this seems straightforward, and it likely reminds you of symbolic math you've learned in your school years. While there are computing environments that can do symbolic math , the above example is an entirely different beast. In symbolic math, the expression `a = b` is equal to the expression `b = a`. Not so in coding (try e.g., `a = 4` vs `4 = a`). Remember that writing `a = 4` means take the value of 4, store it into memory, and give this memory location the name "a". A better way to write this would be `a <- 4`.

Likewise the expression `a+a` means retrieve the value stored at location "a" and do an addition with the value stored at location "a". So the "a" is merely a reference to a memory location. Note that there is nothing specific about calling this "a", You could equally write `myfirstvariablename = 4` but it is a lot of typing.

4.2.1 Operators

The `+` sign is called an operator. There are quite a few operators, but we will likely get away with the basics ones

```
1 + 1 # Addition
8 / 4 # division
2 * 3 # multiplication
3 ** 2 # 3 to the power of 2
```

```
9 ** 0.5 # the square root of 9 (you remember that from school do you?)
```

So by now, you have learned three very fundamental skills:

1. Assigning a label and value to a memory location
2. Retrieving value from a memory location using the label as reference
3. Doing some simple operations on variables and numbers
4. And if you paid attention, how to add a comment into your python code

4.2.2 Variable names

In principle, python variables can have any name you like. However, all variables:

1. Must start with a letter
2. can only contain letters, numbers, and underscores (i.e., no blank spaces or special symbols)
3. You cannot use the minus sign (or dash) to make a compound name. Compound names should always be connected by an underscore. i.e., `student_id` rather than `student-id` which would be interpreted as `student` minus `id`

In addition, I recommend adhering to the following guidelines:

1. Avoid variable names that are too general or too wordy. E.g., rather than `my_first_variable` call it `length` or any other descriptive term.
2. Avoid letters like "I" or "O" where it is hard to know whether you mean lowercase "i", upper case "I", or the number "1", similarly, "O" is easily confused with zero.
3. Variables names should always be lower case
4. And it is always a good idea to add some comments which explain what your variable means (more on this later).
5. If in doubt, google "Python Naming Conventions".

4.3 Compound Variables

This module is available as in your project home directory as:

- `PNTA-Notebooks/Data_types/Lists/lists-slicing`
- `PNTA-Notebooks/Data_types/Lists/slicing-assignment`
- `PNTA-Notebooks/Data_types/Working_with_lists/working_with_lists`
- `PNTA-Notebooks/Data_types/Working_with_lists/working_with_lists_assignment`
- `PNTA-Notebooks/Data_types/Strings/strings`

- PNTA-Notebooks/Data_types/Strings/strings_assignment
- PNTA-Notebooks/Data_types/Other_data_types/other_data_types

4.3.1 Lists

Note 1: This notebook contains several code blocks. The idea is that you execute these code blocks as you read through the text. So treat this notebook as an interactive tutorial rather than a static test. This mode of reading will be the default from now on.

Note 2: If you accidentally modify notebook content, go to `File -> Revert to Checkpoint`, and select the last checkpoint. The system should create a checkpoint the first time you open a notebook. If you accidentally delete a notebook, use the link posted in the assignment, and restore the notebook.

Note 3: Checkpoints are a powerful feature. When you work on your assignments, create them every so often so that you do not lose your work. It is good coding practice to create a new checkpoint whenever you solve a minor milestone and before you tackle the following problem.

So far, we only considered variables which hold a single value. However, the true power of programming stems from the fact that you apply a given procedure over and over. As such, python provides various data-types that can hold more than one value.

Lists hold a sequence of numbers. However, lists are not restricted to numbers; they can contain numbers, letters, words, and even other lists. You can also mix and match these things in the same list. For the sake of simplicity, we will stick to numbers here, though.

Let's create a list:

```
my_list = [1 2 3]
```

Please execute this block.

Ugh, this fails! You need to separate list items with commas!

```
my_list = [1, 2, 3]
```

Why is there no output?

The above statements assign values to a variable. However, the assignment itself is not an operation that yields a result. All it does is create a list. If you want to see the list, you need to use the print function (we will explore the print function in greater detail later on).

```
my_list = [1, 2, 3]
print(my_list)
```

You may have noticed that in the previous modules, most variable names were single letters, whereas here, I use a compound word. Personally, I try to use a single letter or words for simple variables that only contain a single value, whereas compound variables use a two-letter name. You may also notice that I prefixed the variable with the word `my`. This avoids potential overlap with builtin python functions (e.g., `list()` which is a builtin function).

Accessing list elements

Single list elements Consider the following list:

```
my_list=[1.12, 3.2, 1.45, 12.01, 1.12, 3.2, 1.45, 12.01]
```

In real life, lists are often rather long, and we only want to see certain elements of it. Say you only want to see the second element of this list, so we can write

```
print(my_list[1])
```

The number in the square bracket is called the index and refers to the position of the list element inside the list. Why do we see the 2nd element of this list, even so, the index is `1`?

This is because python starts counting at zero. So to see the first list element, you would need to write `my_list[0]` (try this in the above code cell for yourself). Most programming languages behave this way (the notable exception is Matlab).

Accessing a range of elements (slicing) If we want to see several elements of a list (a so-called slice), we can specify a range.

```
print(my_list[2:4])
```

So rather than specifying a list index, we provide a range expression `2:4` which you can read as `start:stop`. Compare the result in the above statement which the actual list. Which index positions are retrieved by this operation? Do you understand why there are only two values in the result? If you need to think about this for more than 5 minutes, start talking to your peers, and if you still can't figure it out, talk to TA or the instructor.

Accessing the last element(s) in a list Sometimes, we have a-priory knowledge of how many list elements there are, sometimes we don't. In this case, we could use a python function to query the length of a list (i.e., `len(my_list)`). However, we can do this more elegantly with range expressions:

```
print(my_list[-1])    # the last element in the list
print(my_list[-2])    # the second last
print(my_list[-3:-1]) # the third and second last!
print(my_list[-3:])   # the last 3
```

If you are really on the ball, you noticed that `-3:-1` and `-3:` give different results and, you now why (it is the same answer as in the previous section). If this is still confusing, please find out from your peers, and if this does not help, flag down a TA or the instructor.

Other list slices In the previous example, we have seen that the colon separates the beginning and end values of a list range, and that if we omit the number, it will default to the max index. The same is true for the starting index

```
print(my_list[0:3]) # will print the first 3 values
print(my_list[:3])  # this is the same
print(my_list[:])   # will print the whole list
```

Last but not least, we can add a third argument to the range expression that tells python to only access every n-th element (`start:stop:step-size`)

```
print(my_list[0:-1:2]) # every second element  until the second last starting with the first
print(my_list[1::2])   # every second element including the last, starting with the second
```

We can extend this syntax to display the list in reverse. I.e., we specify the higher index value as the start value, and the lower index value as a stop value, and then we use a negative step size (this is important!)

```
print(my_list[-3:0:-1]) # start with the 3rd last element going  backwards to the 2n element
print(my_list[-3::-1])  # start with the 3rd last element going  backwards to the 1st element
print(my_list[-1::-1])  # Show the entire list backwards
print(my_list[::-1])    # Same but sthorter
```

If you do not understand why the first results omit the 1st list element, please speak up now.

Take me home: A summary of essential concepts in the above chapter

- You create lists by enclosing a sequence of list elements into square brackets
- A comma must separate individual list entries
- Avoid the letters O or l in variable names as they are easily confused with the numbers zero and one.
- Avoid overwriting built-in python functions by prepending `my_` to your variable names. With time you will get the hang of it which names are being used by python(e.g., print, list, dict, float, etc.) and which names are safe to use (e.g., `i`, `class_list` etc.).
- We can access list elements by providing a single index number
- We can access multiple list elements by providing a range expression [start:stop:step]
- The index of the first list element is zero. The index of the last list element is -1

4.3.2 Working with Lists

Slicing lists is all good fun, but to work with lists, we need meaningful ways to modify it. The following code is straightforward to understand

```
my_list = [1, 2, 3]
print(my_list)
my_list[1] = 44
print(my_list)
```

I also assume that it is clear what I am trying to do in this example

```
my_list = [1, 2, 3]
my_second_list = my_list
```

but explain the following:

```
my_list = [1, 2, 3]
my_second_list = my_list
my_list[1] = 44
print(my_second_list)
```

To understand this result, we need to make a detour into what lists are, and how python handles them.

A quick detour into the world of object-oriented programming (OOP)

There are several ideas on how to map a real-world problem in terms of a program. The two most prominent approaches are functional programming and object-oriented programming.

In a functional program, we have self-contained program code which expects input data manipulates the input data, and returns the output data. The python function `len()` is a good example. In the following, I pass the list `my_list` as an argument to the function=len()= which then determines the length of the list `my_list`, and returns the length of `my_list` as a numeric value that we store in `l`

```
my_list = [6,2,1]
l = len(my_list)
print(l)
```

This approach is straightforward and clean.

The other programming paradigm is called object-oriented programming. In this case, a set of self-contained code not only contains the instructions on how to modify data, but it also contains the data. . While python allows for both programming styles, the actual python language itself is written as object-oriented code. So far, we considered `my_list` just a special type of variable. It is, however, much more. `my_list` is a list object that contains the list data and also knows several methods to manipulate this data.

So, if you want to sort the data using the functional approach, we could write

```
my_list = [12, 1, 14, 6]
my_sorted_list = sorted(my_list)
print(my_sorted_list)
# compare against my_list
print(my_list)
```

Again, we pass `my_list` as argument to the `sort()` function, which will return a sorted list that we store in `my_sorted_list`

Now let's do this in an object-oriented way.

```
my_list.sort()
print(my_list)
```

Here, we call the `sort()` method of the list-object, which will then sort the list. Nice and compact, but now, we have modified the actual list, whereas in the functional example, the original data was left untouched.

We can explore what methods are known to a given object. Try the following:

```
dir(my_list)
```

```
# Out [11]:
# text/plain
: ['__add__',
:  '__class__',
:  '__contains__',
:  '__delattr__',
:  '__delitem__',
:  '__dir__',
:  '__doc__',
:  '__eq__',
:  '__format__',
:  '__ge__',
:  '__getattribute__',
:  '__getitem__',
:  '__gt__',
:  '__hash__',
:  '__iadd__',
:  '__imul__',
:  '__init__',
:  '__init_subclass__',
```

```
:  '__iter__',
:  '__le__',
:  '__len__',
:  '__lt__',
:  '__mul__',
:  '__ne__',
:  '__new__',
:  '__reduce__',
:  '__reduce_ex__',
:  '__repr__',
:  '__reversed__',
:  '__rmul__',
:  '__setattr__',
:  '__setitem__',
:  '__sizeof__',
:  '__str__',
:  '__subclasshook__',
:  'append',
:  'clear',
:  'copy',
:  'count',
:  'extend',
:  'index',
:  'insert',
:  'pop',
:  'remove',
:  'reverse',
:  'sort']
```

This will print a lengthy list of methods available with this object. All the methods which start with a double underscore are meant for internal use only (so-called "dunders"). All others are referred to as "user visible" . In our examples above, when you call the list without anything, the `__str__` method is executed and prints a string with the list data. And that is all you need to know about double underscore methods (dunders).

How do we use these methods? Try this

```
my_list  # this will activate the __str__ method

# now call my_list with the __str__() method and test if you get the same output
# note that methods are called by adding a dot to the object name, followed by the
# method name and a pair of brackets, i.e., my_list.__str__()
```

The results of both expressions are identical. Note, however, the use of the brackets. Without this, the second example will fail (methods like functions always require brackets!)

More interesting (to us) are the methods that are meant to be used by a user of this object (i.e., without underscores). If you check the above, you will find a method called reverse. So let's try this. You already noticed that we could call an object method by appending the method name to the object. Also, for the sake of readability, I prefer to explicitly call the print function as this makes it evident what you are trying to do.

```
print(my_list)
my_list.reverse()
print(my_list)
```

Notice that the reverse method does not return the list values in reversed order; rather, it reverses the list in place!

```
print(my_list)
my_list.sort()
print(my_list)
```

Since you may lose the original list, sorting a list in place may not be what you want! You may need to create a copy to preserve the list ordering.

How to find out what those methods do But how do I know what all of these methods do? Thankfully, there is a simple help system available: Let's try this with the sort method

```
help(my_list.sort)
```

```
# Out [10]:
# output
Help on built-in function sort:

sort(*, key=None, reverse=False) method of builtins.list instance
    Stable sort *IN PLACE*.
```

Some gibberish here, but the key info is `Stable sort *IN PLACE*.` which tells you that it will modify the actual list in place and not return a sorted copy. Contrast this with the output of

```
help(sorted)
```

```
# Out [12]:
# output
```

```
Help on built-in function sorted in module builtins:

sorted(iterable, /, *, key=None, reverse=False)
    Return a new list containing all items from the iterable in ascending order.

    A custom key function can be supplied to customize the sort order, and the
    reverse flag can be set to request the result in descending order.
```

which tells you that this function expects some sort of list (i.e., iterable) and will return a new list that is sorted. If you are still lost, use Google, and search for `python list sort`, which likely directs you to `programiz` where you will find a clear explanation and examples! And if this does not help, pipe up and get in touch with your TA or instructor!

Take me home:

- python objects consist of data and methods to manipulate the data
- you can list the available methods of an object using the `dir()` function
- methods with a double underscore are not meant for external use
- object methods are called by appending the method name with a dot to the object name (i.e., `my_list.sort()`).
- Most object methods do not generate return values, rather they modify data in place.
- You can query what a method does by calling the help function (`help(object.method_name)`)
- functions are called by typing the function name and providing the argument to the function in brackets (i.e., `sorted(my_list)`)
- most functions return a modified copy of the data that then needs to be stored in a new variable.

Referencing objects

So most things python, are actually objects which we can reference by name. The name in turn, is simply a reference to a memory location where this object is stored. Thus, `my_list` is merely an object handle, not the actual variable. This is why the following code does not produce the expected results:

```python
my_list = [1, 2, 3]        # create list object
my_second_list = my_list   # copy object handle
my_list[1] = 44            # use the copied object handle to modify a list element
print(my_second_list)
```

So the second line does not produce a copy of the data in `my_list`, rather, it copies the reference (i.e., the memory location of the list object) to `my_list`. Let's verify this by querying python for the memory address of `my_list` and `my_second_list`

```
print(id(my_list))
print(id(my_second_list))
```

as you can see, they are identical. So if we modify the content of `my_second_list`, and then ask python to print the data at the memory location `my_list` points to, we get the very same data as in `my_second_list`. Confused? You are in good company!

But if you want an independent copy of `my_list`? Python provides several methods around this problem, and as long as you deal with simple lists that do not contain other lists, we can use the `copy` method of the list object. This kind of copy is known as shallow copy . There is also a deep copy function, but deep copies involve some interesting problems which are beyond the scope of this course.

```
my_list = [1, 2, 3]
my_second_list = my_list.copy()
my_list[1] = 44
print(my_list)
print(my_second_list)
```

and just when you think you finally got it, consider this:

```
1 my_list = [1, 2, 3]
2 my_second_list = my_list
3 print(id(my_list))
4 print(id(my_second_list))
5 my_list = 12
6 print(my_list)
7 print(my_second_list)
```

Surprisingly, `my_second_list` is not affected by the assignment in line 3! What happens here? Line 3 is an assignment where you ask python to assign the value of 12 to `my_list`. So this assignment deletes the previous definition of `my_list` and replaces it with the value of 12. This is fundamentally different from `my_list[1] = 44`, where you ask to change the first element of the list. So the assignment expression is not for the list but for the list element!

But what happens to `my_second_list` in the above case? Python checks whether another object references an object. If so, the new `my_list` object is created in a different memory location so that you do not loose the data in `my_second_list`. Let's Verify this:

```
my_list = [1, 2, 3]
my_second_list = my_list
print(id(my_list))
print(id(my_second_list))
```

```
my_list = 12
print(my_list)
print(my_second_list)
print(id(my_list))
print(id(my_second_list))
```

It is actually a bit more complicated than this, but this simple explanation will do for us.

Take me home

- most things python are objects
- objects are programming constructs that contain data and methods to manipulate the data.
- you can query the object methods via `dir(object_name)`
- you can call object methods via `object_name.method_name()`
- you can use the help system to get information on specific method `help(object_name.method_name)`
- object names are a handle to their memory location
- copying the object handle does not copy the data!
- functions expect data as an argument and will return a copy of the processed data (aka result).

Manipulating lists

Back to our main task. You have a list, and you want to append a value

```
my_list = [ 4, 2, 3]
my_list.append(1)
print(my_list)
```

Lets, insert a new number at index position 2

```
my_list.insert(1,44)
print(my_list)
```

Let us remove the last item on the list

```
my_list.pop()
print(my_list)
```

We can also be specific and remove the item at a given index

```
my_list = [6,3,4,6,9]
my_list.pop(1)
```

```
print(my_list)
```

We can remove a value. Unlike `list.pop()` this will remove the first occurrence of the number 6.

```
print(my_list)
my_list.remove(6)
print(my_list)
```

rather than adding a single value, we can add a list of values

```
my_list = [6,3,4,6,9]
my_list.extend([1,2,3])
print(my_list)
```

A variation of the above is when we have two lists that we add together in the following way:

```
second_list = [12, 16, 3, 0]
new_list = my_list + second_list
print(new_list)
```

Will the above also work for subtracting two lists from each other?

We can determine at which index position we will find a given value. For this, you can use the `list.index()` method that will return the first occurrence of a given value. Note, it will **only return the first match!** We will explore how to find all matches in a later chapter.

```
print(my_list)
my_list.index(3)
```

we can count how many times a value occurs in the list

```
print(my_list)
my_list.count(6)
```

and we can remove a value. **Note that this will only remove the first occurrence!**

```
print(my_list)
my_list.remove(6)
print(my_list)
```

and reverse a list (which is different than sorting!)

```
print(my_list)
my_list.reverse()
print(my_list)
```

Sometimes, knowing how many elements are in a list is handy. We can use the python function `len()`. Note how this is a function and not a list method. I.e., we call it by passing the list as an argument to the function rather than calling the list method (i.e., `my_list.len()`)

```
len(my_list)
```

and for good measure, we can delete all list items

```
print(my_list)
my_list.clear()
print(my_list)
```

Take me home

- There are numerous ways to manipulate lists:
  - You can join lists
  - You can add elements to a list at arbitrary positions
  - You can remove elements from a list
  - You can find out where elements are located in a list
  - You can count how often a value occurs in a list
  - You can count how many elements are in a list
  - You have practiced using methods that belong to the list objects (e.g., `list.pop()`, `list.index()` etc.), and using functions like `len(list)`
  - Many of the above methods will only work with the first occurrence of a value. In a later chapter, we will learn how we can use repetition to, e.g., get the index values of all occurrences.

4.3.3 Strings

Strings

Previously, we explored how numbers are stored in memory. This was achieved by changing a number from its representation in the decimal system to the binary system.

There is no simple way to do the same for letters and symbols . However, nothing prevents us from mapping letters to numbers and then storing the number. The only extra step required is to tell the program that this number is meant to be a letter. Python will then use a lookup table to figure out which letter to print as output. Obviously, this will only work if everyone agrees to use the same lookup table...

We can use the `chr()` function to show the character associated with a given number

number to ASCII}

```
print(chr(65))
```

But how do we tell python that a given value should be interpreted as a character? Using `chr()` each time we want to see a character would be tedious.

In our last module, we discovered that each variable is assigned a type (int, float, etc.). These assignments are done when you assign a value to the variable. E.g., if you type `a=12` python will assign an integer-type. If you type `a=12.1` python will assign an float-type.

However, we can't simply write `a=b` to assign the letter `b` to the variable `a` because `b` could also be the name of a variable. Most programming languages thus use quotation marks to indicate that you are assigning a character to a variable. Consider the following

```
b = 12      # the value of the variable b is 12
print(b)
a = b       # the value of the variable a is 12
print(a)
a = "b"     # the value of the variable a is the letter b
print(a)
print("a") # print the character "a"
```

you can use either single quotation marks or double ones

```
a = 'b'   # OK
print(a)
a = "b"   # OK
print(a)
```

but this will not work

```
a = "a'
```

Lastly, We can use the `ord()` function to show which number belongs to a given letter ASCII to number}

```
print(ord("a"))
```

The ASCII table ASCII stands for "American Standard Code for Information Interchange" and defines a way to map characters to numbers (see the below table for a short excerpt).

Dec	Hex	Char	Description
65	41	A	Capital A
66	42	B	Capital B
67	43	C	Capital C
68	44	D	Capital D
69	45	E	Capital E
70	46	F	Capital F
71	47	G	Capital G
72	48	H	Capital H
73	49	I	Capital I

The complete table is available at https://en.wikipedia.org/wiki/ASCII The ASCII standard is the oldest and most widely accepted way to map characters to numbers. However, due to its age and country of origin, it comes with considerable limitations. It was initially designed to store text characters with a bit-width of 1 byte. I.e., with numbers between 0 and 255. This implies that there are insufficient mappings to consider special characters, like umlauts, let alone other alphabets.

It is only recently that a globally accepted mapping between numbers and text has become available (https://en.wikipedia.org/wiki/Unicode), but even there, different variants exist, and not every operating system supports them similarly. This is one of the reasons why we will only use letters that are defined in the original ASCII-table.

Fun fact: The name of the artificial intelligence in the movie "2001, A Space Odyssey" was HAL. This name was derived by subtracting 1 from the ASCII values of IBM.

Strings Working with single letters is not convenient. Thus, every computer language knows about sequences of letters, which are called strings. We can think of strings simply as a special type of list. It should, therefore, be no problem for you to print, e.g., the 3rd letter of this string.

```
a = "This is an example of a string"
print(a)
# now print the 3rd letter of this string
```

Unlike lists, you cannot modify elements of a string (i.e., they are immutable):

```
a = "Joe"
a = "Jessie" # this is ok
a[2] = "x"   # this will not work
```

Similarly to lists, you can work with ranges and query a string object to see which methods it provides. We can do this with the `dir()` function as follows:

```
a = "Joe"
dir(a)
```

Remember that any names that start with an underscore are best ignored, unless you want to change the internals of the python language.

You can get further information on these methods with the `help()` function by combining the variable name with the method name, e.g.,

```
a = "Joe"
help(a.upper)
```

Other Data Types

Python knows various data types, many of which we may never use. However, the most important ones should at least be mentioned:

Vectors versus Lists If you have used matlab before, you will be familiar with vectors. Unlike matlab, python does not have a native vector type. Lists can contain almost anything (e.g., other lists, strings, and any of the types below). Vectors, however, can only contain numbers and support mathematical methods (i.e., multiplying two vectors will give you the cross product). We will learn how to use vectors in a later module.

Tuples Python tuples are a special version of a list that won't allow you to modify the value of a list element. We define a tuple by declaring the list with regular brackets rather than square brackets.

```
my_list = [1, 2, 4]       # a regular list
my_tuple = (1, 2, 3) # a tuple list
```

We access elements of a tuple via index or range operations similar to regular list

```
print(my_tuple[1]) # note the square brackets for the index expression!
```

however, unlike lists, **you cannot change the values in a tuple!**

```
my_tuple[1] = 3
```

While you cannot change the value of a tuple, it is possible to join two tuples to create a **new** tuple.

```
my_first_tuple = (1, 2, 3)
my_second_tuple = (3, 8, 7)
new_tuple = my_first_tuple + my_second_tuple
print(new_tuple)

# or replace a existing tuple
```

```
my_first_tuple = my_first_tuple + my_second_tuple
print(my_first_tuple)
```

Tuples are an immutable type (similar to strings). So you can create them, you can delete them, and you can re-assign them. But you cannot change the values of the tuple elements.

Sets Sets are declared with curly braces. Unlike lists, you can only add unique elements. If there are duplicates, they will simply be ignored.

```
my_set = {1, 2, 3, 3}
print(my_set)
```

Another critical difference is that sets are not indexed. I.e., you cannot write `my_set[1]`. Set elements can be added, removed, and changed. Furthermore, sets provide all sorts of exciting operations (i.e., union, difference, intersection, subset, etc.). A practical example of how to use sets would be the following case. You have the list of students enrolled in ESS224H1 Introduction To Mineralogy And Petrology, and the list of students who are enrolled in ESS262H1 Earth System Processes. If both are a set, you can use a single command to find out which students are enrolled in both courses:

```
ESS224 = {"Maria", "Stuart", "Andy", "Drew"}
ESS262 = {"James", "Mark", "Alex", "Maria", "Silvy", "Stuart"}
ESS224.intersection(ESS262)
```

or, imagine that you have two e-mail lists, and you want to join both list and you ant to make sure that people do not receive the e-mail in duplicate

```
ESS224 = {"Maria", "Stuart", "Andy", "Drew"}
ESS262 = {"James", "Mark", "Alex", "Maria", "Silvy", "Stuart"}
print(ESS224.union(ESS262))
```

Dictionaries Dictionaries are a datatype that enables you to look up values based on a key rather than an index. Each dictionary entry consists of a key-value pair. The key can be a number, a string, or tuple, and the value can be pretty much anything (e.g., a value, a string, a list, another dictionary, etc.). Let's consider this simple example:

```
in_class_quiz = {"Domenica" : 72,
                 "Brian" : 77,
                 "George" : 95,
                 "Liz" : 81}
```

Note that dictionaries use curly braces similar to sets, but the internal structure is different.

We can query the dictionary to see how individual students performed in the quiz.

However, rather than referring to the result by a numeric index, we use the key

```
print(in_class_quiz["Brian"])
```

Note that the key needs to be unique otherwise, you will override the earlier definition.

```
in_class_quiz = {"Domenica" : 72,
                 "Brian" : 77,
                 "George" : 95,
                 "Brian" : 50,  # Brian appears twice, so this entry overwrites the earlier one
                 "Liz" : 81}
print(in_class_quiz["Brian"])
```

similar to lists, we can change/update individual values by referring to their key.

```
in_class_quiz["Brian"] = 70
print(in_class_quiz["Brian"])
```

Random bits and pieces All of these data types are ultimately ways of storing list-type elements. One of the most often used operations on a list is determining the number of list elements. The python function `len()` will do exactly this

```
s = "Test"
l = [1, 2, 3]
s = {1 ,2}
print(len(s))
print(len(l))
print(len(s))
```

You may have stumbled upon this before, but it is possible to convert many of these datatypes into another data type:

```
t = (1, 2 , 3, 3)
l = list(t)
s = set(l)
print(t)
print(l)
print(s)
```

4.4 The print statement

This module is available as in your project home directory as:

- `PNTA-Notebooks/The_print_statement/the_print_statement`

- `PNTA-Notebooks/The_print_statement/the_print_statement_assignment`

Note, the terms "statement" and "function" are used interchangeably.

4.4.1 Recap

Over the course of the last couple of modules, we implicitly learned (hopefully) a couple of essential things about python. Here is a summary. If any of these concepts or terms are unclear, please go back to the previous modules and discuss them with your TA/Prof where necessary.

1. While python understands numbers without help, we have to mark strings with quotation marks (otherwise, python would interpret it a variable or function name): `s = "Hello World"`
2. It matters how we use brackets:
   - Square brackets to the right of an equal sign indicate that you declare a list: `a = [1, 2, 4]`. Square brackets to the right of a name or letter indicate an index expression: `a[1] = 22`. Index expressions can also be used to address ranges and stepsize: `a[1:2:1]`.
   - Round brackets to the right of an equal sign are used to declare a tuple: `t=(1, 2, 3)`, whereas round brackets to the right of a name indicate that this name is a function: `print()`. To access the 2^{nd} element of the tuple, you will still have to use square brackets since you are doing an index operation: `t[1]`. Tuples are immutable.
   - Curly braces to the right are used to declare a set: `u = {1,2,3}`. Set members cannot be retrieved by an index operation, and set values are immutable.
   - Curly braces are also used to declare dictionaries. In this case, the declaration contains key:value pairs that are separated by a colon. `d = {"B":12, "A":14, "C":7}`. Dictionary entries can be retrieved by using an index expression using the dictionary key: `d["B"]`
3. A name followed by regular brackets denotes a python function: `print()`. Functions exist independently of data (or objects like lists).
4. A name followed by a dot and another name with brackets denotes an object method: `a.sort()` which only exists as long as the object `a` exists.
   - Object methods can either modify the object-associated data in place, e.g., `a.sort()`
   - Or, they do not modify the object associated data, but instead return something, e.g., `s.split(",")`. So if you would like to change the original data set, you would have to write an explicit assignment: `s = s.split(",")`. If in doubt, use the `help()` function.
5. We can query the object type with the type function: `type(a)`, and we can list the

object-associated methods with the dir function: `dir(a)`

6. We can get help on python functions with the help command: `help(print)`
7. We can get help on object methods with the help function: `help(list.pop)`
8. The name of a compound variable (e.g., lists) refers to its object handle, not its value. Thus, the expression `my_new_list = my_old_list` will not copy the list values; it will only copy the memory address where those values are stored. In other words, if you modify `my_new_list`, you will also modify `my_old_list`. To create an independent copy you need to use the copy method: `my_new_list = my_old_list.copy()`

4.4.2 Using print()

By now, you probably noticed that the print statement is a function. Functions usually take one or more arguments (i.e., data or variables) that are enclosed by regular brackets and return a modified copy of that data. We use the So far, we have used the print function to echo the value of a variable back onto the screen. However, we can do much more with print function. This module will teach the basics of producing formatted output.

Most things python are straightforward, but the syntax of the print function is not. This is simply because every new python version provides new ways to format the print output. In this course, we will use the syntax provided by python 3.7, the so-called "f-strings". You will note that this is different from the syntax in many examples on the web, which often refer to python 2.7.

4.4.3 F-strings

The `print()` statement allows us to display the value of a variable.

```
a = 2
b = 4
c = a * b
print(c)
```

this is handy enough while you develop your code, but it would be nice if could print something more meaningful, e.g., "The square-root of 12 equals 4".

In other words, we need a way to mix strings with variable values. This is achieved with the so-called "f-strings" (python 3.7 and newer only).

```
a = 2
b = 4
c = a * b
```

```
4 message = f"Multiplying {a} with {b} yields {c}"
5 print(message)
```

Let's analyze the above example: we precede the string declaration in line 4 with an `f` which signals to python that this string will be an f-string. As usual, we enclose the string in quotation marks. After the opening quotation marks, we have some text, which will be output as is. However, in f-strings anything between the curly braces will be interpreted as python code. This can be a variable, a function, or any another valid python expression. Try the following to verify this claim

```
a = 12
print(f"Four times {a} equals {4*a}")
```

f-strings provide us with a powerful way to mix text and computational results.

4.4.4 Escape sequences

Imagine that you need to print more than one line, e.g., something like this:

```
----------------------------

       Attention!

----------------------------
```

This can be achieved in a variety of ways, e.g., we could create the above explicitly

```
print(" - - - - - - - - - - - - - - -")
print()
print("       Attention!")
print()
print(" - - - - - - - - - - - - - - -")
```

Not a bad way, but lots of typing. If we could tell python to insert a new line and tab, we could rewrite this statement more shortly. This is done with so-called escape characters.

```
a = " - - - - - - - - - - - - - - -"
message=f"{a} \n\n \t Attention! \n\n{a}"
print(message)
```

In python (and many other languages), the backslash marks a control character, i.e., the character after the backslash has a special meaning. So in the above example, the sequence of `\n` will be replaced with a linefeed command (i.e., an empty line), and the `\t` will be replaced by a tab. Furthermore, you can use the backslash to indicate that the following character should not be interpreted as a special command. Imagine a case where you want to print the backslash in your output. This is done by preceding the backslash with a backslash. This is also called escaping the backslash.

```
message="Print a newline here: \nprint the backslash here: \\"
print(message)
```

Here is a list of some frequently used escape sequences. Note, if you use these in your notebook text cells, you need to wrap them into a code block (triple backticks). Escape sequences are not part of the markdown syntax and cause all sorts of weird problems (among them missing pdf output)

Escape Sequence	Meaning
`\newline`	Ignored
`\`	Backslash (\)
`\'`	Single quote (')
`\"`	Double quote (")
`\a`	ASCII Bell (BEL)
`\b`	ASCII Backspace (BS)
`\f`	ASCII Formfeed (FF)
`\n`	ASCII Linefeed (LF)
`\r`	ASCII Carriage Return (CR)
`\t`	ASCII Horizontal Tab (TAB)
`\v`	ASCII Vertical Tab (VT)
`\xA1`	ASCII character with hexadecimal value A1
`\135`	ASCII character with octal value 135 (])

4.4.5 Multiline f-strings

Sometimes, your message string won't fit on a single line. In this case, we can group several f-strings together

```
message = (
    f"This is the first string"
    f"This is the second string"
)
print(message)
```

You will note that the above does not automatically include linefeeds. I.e., you will need to add the appropriate escape sequences into the f-string.

4.4.6 Format modifiers

The above gives us already a lot of control over the output from a python program. However, consider the case where you get measurements from an analytical instrument, which reports the data as 12.3456006423. However, the actual instrument precision is only around 0.2. There, it would be nice to restrict the output of the print statement to significant figures.

We use format modifiers to modify how we print numbers. Try this:

```
a =  12.3456006423
print(f"The value of variable a equals {a}")
# now we add a format modifier
print(f"The value of variable a equals {a:1.2f}")
```

Let's analyze this statement. We use a colon after the variable name to attach a format modifier (note this method is specific to f-strings). The first number states that our output must have a minimum length of 1, with no more than two digits after the decimal point. Modify this statement so that you print the result with 3 and 4 significant figures. Also, does this operation truncate the number, or will the number be rounded?

The first number in the format modifier specifies the minimum length of the statement. I.e., if this number is larger than the length of the actual output (in the above case, 4 characters), the result will be padded with leading spaces. This can be important when saving data in a format that requires that the decimal point is always at the 7th character position.

```
a =  12.3456006423
print(f"The value of variable a equals {a}")
# now we add a format modifier
print(f"The value of variable a equals {a:6.2f}")
```

Similarly, python knows the "s" format modifier for strings. This can be used to

- truncate long strings
- pad them with blanks on the left
- pad them with blanks on the right

```
message = "Hello World"
print(f"The content of the message string = {message:.4s} - so what?") # use only the first
print(f"The content of the message string = {message:>20s} - so what?") # add 20 spaces in
print(f"The content of the message string = {message:<20s} - so what?") # add 20 spaces aft
```

Likewise, we can use the "d" modifier to pad or force python to print the sign of the number (automatically done for negative values), but usually omitted for positive ones.

```
a = 12347687
print(f"a = {a:>20d} - so what?")
print(f"a = {a:<20d} - so what?")
print(f"a = ${a:+d} - so what?")
```

Note that you cannot use modifiers to display floating point numbers as integer or string.

Integer Modifiers

These will only work for integer numbers

'b' Binary. Outputs the number in base 2.

'c' Character. Converts the integer to the corresponding Unicode character before printing.

'd' Decimal Integer. Outputs the number in base 10.

'o' Octal format. Outputs the number in base 8.

'x' Hex format. Outputs the number in base 16, using lower-case letters for the digits above 9.

'X' Hex format. Outputs the number in base 16, using upper-case letters for the digits above 9.

'n' Number. This is the same as 'd', except that it uses the current locale setting to insert the appropriate number separator characters.

'' (None) - the same as 'd'

Floating point modifiers

These will only work for floating point numbers

'e' Exponent notation. Prints the number in scientific notation using the letter 'e' to indicate the exponent.

'E' Exponent notation. Same as 'e' except it converts the number to uppercase.

'f' Fixed point. Displays the number as a fixed-point number.

'F' Fixed point. Same as 'f' except it converts the number to uppercase.

'g' General format. This prints the number as a fixed-point number, unless the number is too large, in which case it switches to 'e' exponent notation.

'G' General format. Same as 'g' except switches to 'E' if the number gets to large.

'n' Number. This is the same as 'g', except that it uses the current locale setting to insert the appropriate number separator characters.

'%' Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign.

'' (None) - similar to 'g', except that it prints at least one digit after the decimal point.

For more details, see `https://peps.python.org/pep-3101/#standard-format-specifiers`

4.5 Comparisons and Logic Operators

This module is available as in your project home directory as:

- PNTA-Notebooks/comparisons_and_logic_operations/comparisons_and_logic
- PNTA-Notebooks/comparisons_and_logic_operations/comparisons_and_logic_-assignment

Program code, which only ever executes the same sequence of events, is somewhat limited. As such, we need a way to change program execution depending on certain circumstances (typically the value of a variable). We, therefore, need a way to compare the values of variables and decide, e.g., whether one variable is equal to another.

4.5.1 Comparison operators

If we want to compare two numbers, we obviously cannot use the equal sign

```
a = 12
b = 14
a = b  # this assigns the value of b to a
```

In most programming languages, it is thus customary to write a double equal sign to denote a comparison.

```
a = 12
b = 14
a == b  # this compares the value of b to a
```

Unlike numeric operators, comparisons do not result in a numeric value but a truth value (logic or boolean value). In other words, two numbers are either equal (so the statement is `True`) or are not equal. Thus the statement is `False`

Most languages signify this by reporting this with the reserved words of `True` or `False`. Since there are only two logical states, boolean values can be represented by a single bit (i.e., either 0 or 1). Compare this to the memory requirements of a regular integer value (64-bit).

The basic comparisons operators are as follows:

```
5 == 4  # test for equality
2 != 3  # test whether two values are not equal
5 > 4   # test for greater value
5 < 4   # test for smaller value
1 >= 1  # test for greater or equal value
2 <= 2  # test for smaller or equal value
```

We can use the above operators in one way or another on most python data types, although you have to be careful to understand what is being compared. If we are comparing strings, the equality operator will tell you if two strings are the same or not. Run the following code:

```
print(f'"Apples" == "Apples" yields {"Apples" == "Apples"}')
print(f'"Apples" == "Oranges" yields {"Apples" == "Oranges"}')
```

Note the use of single quotes in the above f-string, which allows me to use double quotes inside the f-string.

But what is the point of

```
print(f'"Apples" < "Oranges" yields {"Apples" < "Oranges"}')
```

In this case, python compares the ASCII value of the first letter in each word, so this comparison is not particularly meaningful.

Some functions can return truth values as well. The following example tests whether a number is, e.g., of type real or type integer

```
isinstance(12, int)
```

Another example is the rather useful `in` operator

```
s="Apples and Oranges"
print("Apples" in s")
```

this also works for any list type data including dictionaries

```
d = {"Apples":10, "Oranges":12}
r = "Apples" in d
print(r)
```

4.5.2 Logic Operators

Now that we have truth values (aka boolean values), we can use logic operators to create more complex statements. Think, e.g., a condition like this: The student will pass the class if he submits more than 70% of his assignments, and is never more than 10 minutes late to class. This requires that we combine two or more comparison operators. There is indeed a whole subset of mathematics dedicated to this (boolean algebra), but for our purpose, we get away with the 3 basic operators called `and`, `or`, and `not`.

```
A = True
B = False
A and B   # only true if both are true
A or  B   # true if at least one is true
```

```
A not B   # wrong syntax, see below
```

let's do an actual example

```python
x = 12

r = x > 0 # test if x is greater than 0
print(f"x > 0 = {r}")

r = x < 10 # test if x is smaller than 0
print(f"x < 10 = {r}")

r = x > 0 and x < 10 # test if x is bewteen 0 and 10
print(f"x > 0 and x < 10 = {r}")

r = x > 0 or x < 10 # test if x is either larger than 0 or smaller than 10
print(f"x > 0 or x < 10  = {r}")
```

You can see that the above can get messy pretty quickly. Things get even more complicated if we start using the `not` operator. Imagine we need a statement that tests whether `x` falls in between a specific range (3^{rd} expression above). We can write this using the `not` operator as

```python
x = 12
r = not(x > 0 and x < 10) # test if x is not between 0 and 10
print(f"not(x > 0 and x < 10) = {r}")
```

However, with a little bit of thinking, we can rewrite this as

```python
x = 12
r = x <= 0 and x >= 10 # test if x is smaller than 0 and larger than 10
print(f"x < 0 and x > 10 = {r}")
```

which is easier to read.

Note that it makes a difference whether you write `x > 4` or `not(x < 4)`. The former includes 5, whereas the latter statement also includes the number 4. So care must be taken when negating (or un-negating) statements. Use the following snippet to test whether both statements give the same result for various values of `x`.

```python
x = 12
print(f"x = {x}")
r = x > 0 or x < 10 # test if x is either larger than 0 or smaller than 10
print(f"x > 0 or x < 10  = {r}")

r = not(x >= 0 or x <= 10) # test if x is either larger than 0 or smaller than 10
print(f"not(x >= 0 or x <= 10)  = {r}")
```

This becomes even more tricky when you use the `or` statement. The first logic operation in the next cell is true

```
x = 12
print(f"x = {x}")

r = x > 0 or x < 10 # test if x is either larger than 0 or smaller than 10
print(f"x > 0 or x < 10  = {r}")

# If we negate the above in english, we would get:
# test if x is neither larger/equal than 0  or smaller/equal than 10
# however, that is not what this code does
r = not(x >= 0 or x <= 10)(f"not(x >= 0 or x <= 10)  = {r}")

# rather, it says :  not(x is either larger than 0 or smaller than 10).
# to get the English language version, you would have to write

r = not(x >= 0) or not(x <= 10)
print(f"not(x >= 0) or not(x <= 10)  = {r}")
```

The second statement is the not-version of the first (so `True` becomes `False` or vice versa), but it is not how we would negate a statement in English.

Only the third statement is equivalent to the English language meaning. Much confusion and incorrect code originates from the careless writing of logical expressions. To this end, I am a great fan of using brackets to clarify logic operations. Consider this example

```
r = not(x >= 0 or x <= 10)
# versus
r = not(
    x >= 0
    or
    x <= 10
)
# or
r = not(
    (x >= 0) or (x <= 10)
)
```

Space comes for free; deciphering code costs time. . .

Use the above snippet to test whether these expressions give the same results for various values of x. Also, note the use of the brackets to group statements and the absence of commas. Otherwise, you would create a tuple!

4.6 Controlling Program flow with block statements

The previous section taught you the various methods to store, access, manipulate, and display data. The next big step in learning how to code is to combine these methods to create a program that can achieve a purpose.

Creating a program can often be a daunting experience. Often, you won't know where to start, and once you begin, things can get messy pretty quickly. However, similar to hiking, where you may not see the entire way ahead of you, all that is needed is the ability to take the first step, and to have a plan.

Consider this case: You have been asked to write a program that will convert a binary number into a decimal number. We can divide this task into three sub-tasks:

- get user input (the binary number)
- convert the number to decimal
- provide user feedback by printing the results

You can write (and test) code for the above steps without knowing anything about the other two. In other words, programming is an exercise of dividing a problem into smaller problems that you can tackle step by step.

Programming languages thus provide a variety of methods to group statements. The three basic groups shared by almost all languages are:

1. "if groups" that allow you to group programming statements based on a given condition (e.g., if `A` is larger than 10 do this, else, do that)
2. "loop groups" that allow you to repeat programming statements a given number of times (or until a condition is met)
3. functions allow you to group code statements as well, but this time the statements inside the function are invisible to the code outside the function. A good example is the `print()` function. You, as a user, do not need to know the code of the print function, you only need to know how to pass a value to the print function. This shields you from the complexity of how to actually print characters on the screen. This keeps your code clean, and reduces complexity.

These three methods of grouping computer code allow you to solve pretty much any computing task.

4.6.1 If statements

These modules are available as Jupyter Notebooks in:

- `PNTA-Notebooks/block_operations/if_statements`
- `PNTA-Notebooks/block_operations/if_statements_assignment`

The previous section taught you the various methods to store, access, manipulate, and display data. The next big step in learning how to code is to combine these methods to create a program that can achieve a purpose.

Creating a program can often be a daunting experience. Often, you won't know where to start, and once you begin, things can get messy pretty quickly. However, similar to hiking, where you may not see the entire way ahead of you, all that is needed is the ability to take the first step, and to have a plan.

Consider this case: You have been asked to write a program that will convert a binary number into a decimal number. We can divide this task into three sub-tasks:

- get user input (the binary number)
- convert the number to decimal
- provide user feedback by printing the results

You can write (and test) code for the above steps without knowing anything about the other two. In other words, programming is an exercise of dividing a problem into smaller problems that you can tackle step by step.

Programming languages thus provide a variety of methods to group statements. The three basic groups shared by almost all languages are:

1. "if groups" that allow you to group programming statements based on a given condition (e.g., if `A` is larger than 10 do this, else, do that)
2. "loop groups" that allow you to repeat programming statements a given number of times (or until a condition is met)
3. functions allow you to group code statements as well, but this time the statements inside the function are invisible to the code outside the function. A good example is the `print()` function. You, as a user, do not need to know the code of the print function, you only need to know how to pass a value to the print function. This shields you from the complexity of how to actually print characters on the screen. This keeps your code clean, and reduces complexity.

These three methods of grouping computer code allow you to solve pretty much any computing task.

The last module explored the use of comparison operators that return truth values. To branch the sequence of execution in our code depending on a given circumstance, we will need a statement that takes a truth (boolean) value and allows us to create two groups of code. One gets executed if the statement is true, and another block gets executed when the statement is false.

Most programming languages know a variation of the so-called "if-then-else" clause. The `if` part expects a truth value and executes the code followed after the `then` in case the value is `True`. In the case of a `False`, it will execute the part after the `else`. This is pretty straightforward, and the only other ingredient needed is a way to group programming code.

In python, all block statements end with a colon, and the code group following this block statement has to be indented. You end the code block by returning to the previous indention level. See this example:

```
a = 10

# start a new code block
if a > 10:  # the result of the comparison is either True or False
    # if true, execute this block
    message = f"a = {a} which is larger than 10"

# since the next command is back to the left, it ends the if block
else:  # start the else block
    # if false, execute this block
    message = f"a = {a} which is euqal or smaller than 10"

print(message)
```

Since the `print(message)` statement is moved back to left, it is not part of the `else` block. Try moving it to the right, and run the above code for `a=10` and `a=12`. Do you understand why there is no result if `a=12`?

Multi-way branching

If statements can contain more than one condition. Imagine you get data from some analytical machine that outputs the measured voltage between 0 and 50 volts. You also know that values below 2 and above 20 are unreliable, and values larger than 50 indicate a malfunction.

You can combine several `if` statements using the `elif` keyword which stands for "else if". Check the results of this code against voltage values of 0.2, 4, 22, and 55.

```
voltage = 12

if voltage < 2:
    print(f"Your sample is to small")
elif voltage > 2 and voltage < 20:
    print(f"Voltage = {voltage}V")
elif voltage > 20 and voltage < 50:
    print(f"Your sample is to big")
elif voltage > 50:
    print("---------------------------------")
    print(f"\n\nAttention: Instrument malfunction!\n\n")
    print("---------------------------------")
else:
    print(f"You should never see this")
```

Note: Python 3.10 now supports the `match` statement, which is more elegant. However, at present, your notebooks run on python 3.9. See https://www.python.org/dev/peps/pep-0635/ for more details.

Take a pass

Sometimes it is useful to do nothing. The following code is perfectly fine, but we have seen that the `not` operator can be confusing (especially in more complex examples).

```python
a = 12

if not a == 12:
    print(a)
```

Using the `pass` statement, we can rewrite this as

```python
a = 12

if a == 12:
    pass
else:
    print(a)
```

A bit wordy but much easier to read.

Obviously, with a bit of thinking, we could have written this more succinctly as

```python
a = 12

if a != 12:
    print(a)
```

Nested blocks

Code blocks can be nested into each other. We could, e.g., rewrite the above voltage example as

```python
voltage = 0.1
if voltage < 50:
    if voltage <= 20:
        if voltage > 2:
            print(f"Voltage = {voltage}V")
        else:
            print(f"Your sample is to small")
    else:
        print(f"Your sample is to big")
else:
    print("---------------------------------")
```

```
    print(f"\n\nAttention: Instrument malfunction!\n\n")
    print("--------------------------------")
```

As you can see, the nested block is shifted to another indentation level to the right, which means that it will only be executed if the second if condition is true. Note that this code is functionally equivalent to the example before but that it is much harder to figure out what it does. Similar to joining comparison operators with logic operators, it pays to think about the required logic carefully and simplify it as much as possible.

Pitfalls

The execution of multi-way branching statements stops with the first true condition. As such, conditional statements that follow afterward are not tested. Care is needed in designing such statements. In the following example, the second condition is never met, so you would never know that `a` is bigger than 7.

```
a = 12
if a > 5:
    print("a is bigger than 5")
elif a > 7:
    print("a is also bigger than 7")
else:
    print(f"a={a}")
```

Ternary Statements

Python supports an abbreviated form of writing logic and conditional expressions. I think they are bad style because it is not immediately obvious what the code is trying to achieve. However, you should be at least aware that this kind of syntax exists:

```
a = ''
b = 'Some text'
c = "more text"
x = a or b or c or None
print(x)
```

This assigns the value of the first non-empty object to x. Similarly, you can write

```
a = 13
b = 5
x = 16 if a > 12 else 22
print(x)
```

explore this for various values of `a`. If you find this confusing, be my guest, but remember where to look it up if you come across a ternary statement.

BTW, why am I so obsessed with clarity versus brevity? As a programmer, you will spend 90% of your time looking at code, trying to figure out what is happening in front of you. Only a small fraction of your time is spent creating code. There are, in fact, powerful and elegant programming languages out there. But they never gained popularity because it is hard to understand what the program is doing. A good example is IMB's APL (Advanced Programming Language). The following is a complete APL program that implements Conway's Game of Life.

```
life←{↑1 ω∨.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂ω}
```

Pretty slick, but good luck understanding how this works.

4.6.2 Loop statements

These modules are available as Jupyter Notebooks in:

- `PNTA-Notebooks/block_operations/loop_statements`
- `PNTA-Notebooks/block_operations/loop_statements_assignments`

Coding would be pointless if we had to execute each piece of code manually. Applying some code to each element of a list is where the true power of computing arises. We do this with the so-called loop statements (another type of block statement). Consider the following list

```python
my_list = ["Garnet", "Apatite", "Quartz", "Pyrite"]
```

Your task would be to print out each list element on a line by itself. Yes, you can write four print statements, but this will be tedious if you start dealing with a longer list. It would be a lot simpler if we had a way to tell python to repeat a given action 4 times. It would be even better if we could tell python to repeat an activity for every list element.

The for-loop

Repeating a given set of instructions for a known number of repetitions is the domain of the `for loop`. You may recall that we can access list elements via their index, e.g., to access the second element of a list we can write

```python
my_list[1]
```

Rather than writing the number one, we can use the value of a variable as index (this is also known as indirection). Doing so has the advantage of changing this value during program execution.

```
a = 1
my_list[a]
```

In most programming languages, you can solve the task of printing each element of `my_list` by: 1) Establishing `n` as the number of elements in the list; 2) Creating a loop that is executed `n` times;

1. Adding

a counter `i` to the loop that increases by one for each iteration of the loop; Access each list element by index using `i` as the index. We can do this in python as well:

```
my_list = ["Garnet", "Apatite", "Quartz", "Pyrite"]
n = len(my_list)  # get the number of elements in my_list

# start the loop block
for i in range(0,n,1):  # range(start,stop,step)
    print(f"i = {i}, my_list[{i}] = {my_list[i]}")
```

However, python has a more elegant (aka pythonic way) of looping over a list. Since python knows the type of `my_list`, python knows that this is a datatype with many elements. So we can write:

```
my_list = ["Garnet", "Apatite", "Quartz", "Pyrite"]
for e in my_list:
    print(e)
```

In this case, python will go through the list, starting with the first element, then assign the value of the first element to the variable `e`, execute the statements inside the loop block, and then continue with the next list element.

Try what happens if you change `my_list` to a tuple or set. Also, try what happens if you set `my_list=12`, and `my_list = "Hello World"`

And we can use the usual slicing statements to limit which elements the loop acts upon.

```
my_string ="Hello World"

for e in my_string[0:-1:2]:
    print(e)
```

Enumerating loops

Assume we have two lists (and ignore that this would be easier with a dictionary). To get the corresponding entry in the second list, we will need a counter that keeps track of where in the list we are:

```
products = ["Cheese", "Apples", "Butter"]
price = [12.03, 4.60, 7.01]
i = 0
for p in products:
    print(f"{p} = ${price[i]}")
    i = i + 1  # increase the counter by 1
```

another option would be to use the range function

```
products = ["Cheese", "Apples", "Butter"]
price = [12.03, 4.60, 7.01]
n = len(products)
for i in range(n):
    print(f"{products[i]} = ${price[i]}")
```

python offers, however, a more elegant way

```
products = ["Cheese", "Apples", "Butter"]
price = [12.03, 4.60, 7.01]
for i, p in enumerate(products):
    print(f"{p} = ${price[i]}")
```

Dictionaries Looping through a dictionary is similar to lists, but we need a way to differentiate between keys and values. This can be achieved with the `items()` method of the dictionary class

```
d = {"Pyrite" : "FeS",
     "Barite" : "BaSO4"}

print(d.items())
```

The `items()` method returns a list of tuples, where each tuple consists of the key and the value of a dict entry.

We can use this to loop over this list and unpack each tuple into two variables:

```
d = {"Pyrite" : "FeS",
     "Barite" : "BaSO4"}

for k, v in d.items():
    print(k, v)
```

So for each entry, python assigns the first value to `k` and the second to `v`. This so-called unpacking is a powerful feature. Explore for yourself what happens here:

```
d = {"Pyrite" : "FeS",
     "Barite" : "BaSO4"}
```

```

a, b = d.items()
```

While-loops

The other important loop type is the while loop. This type of loop executes until it is switched off (and if your switch does not work, it will run forever). In other words, while-loops are most useful in combination with if statements.

If you get the following joke, you've understood the essence of while-loops. "A wife asks her programmer husband to get milk while he is out. This was the last she saw of him". No worries if this makes no sense to you. It soon will!

To give a practical example, any ATM is running a while loop that shows the greeting screen. This loop runs until you slide your bank-card into the machine. Then the loop is interrupted, and you are asked to enter your pin. The below example demonstrates this in a simple way.

```
1  # initialize a as True, otherwise the while loop will never execute
2  a = True
3  
4  # do the while loop until a becomes False
5  while a:
6      # print instructions
7      print("\n Stop this loop by hitting the s-key")
8  
9      # wait for user input
10     my_input = input("Hit any other key to continue:")
11 
12     # evaluate input
13     if my_input == "s":
14         print(f"\nYou pressed the '{my_input}' key")
15         a = False  # this will stop the while loop
16     else:
17         print(f"\nYou pressed the '{my_input}' key")
18 
19 print("\nGood bye")
```

Once our code reaches line 5, the `while` statement will test if `a` is True or not. If `a = False`, the `while` block is skipped, and the code jumps directly to `print("\nGood bye")`.

If `a = True`, python will enter the `while` block, and go through each statement. Once it reaches the last statement, it will jump back to line 5, test the value of `a` again, and begin the circle anew. This will continue until some code inside the `while` block changes the value of `a` to False. Can you now explain why the wife's husband never returned?

While loops can be tricky if you get your logic wrong. Carefully consider this example before you run it.

```python
s = True # initialize the switch and set it to no
n = 0     # a counter
my_numbers = list(range(4))
while s:
    if my_numbers[n] == 3:
        print(f"my_numbers[{n}] = {my_numbers[n]}")
        s = False
        n = n + 1
```

So here, the counter is inside the if statement. I.e., it will never increase because `n` will stay zero forever. This might have happened because you missed the flawed logic or your indention accidentally changed. If you execute the above cell, you will start an infinite loop that will run forever (read on to find out how to fix this).

Your notebook shows an asterisk while executing your code. So if you accidentally created an infinite loop, you will see something similar to the following screenshot:

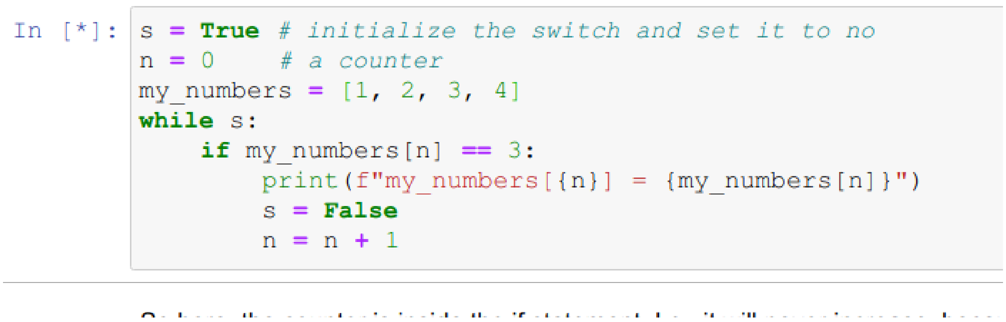


Figure 4.1: The asterisk to the left indicates that your program is busy

If this happens, you have to restart your notebook kernel via the kernel menu

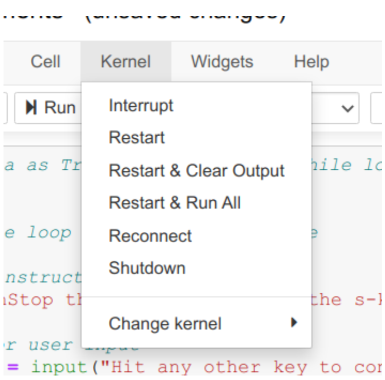


Figure 4.2: Use the kernel menu, if your code is stuck in an infinite loop

Advanced loop features Python loops support a couple of features that are not mentioned above. Most of these, you will not need for this course, but you should at least have heard about it.

- The `continue` statement will stop the execution at the current line and jump back to the header of the loop (i.e., execute the next iteration)
- The `break` statement will jump out of the loop. This is often used with nested loops
- The `else` statement is run only if the loop ends prematurely. Most useful in combination with the break statement

1. List comprehensions We already know that python makes it easy to iterate through the elements of a list. We can use this, e.g., to calculate the squares of a given sequence, and save the results into a new list:

```
my_list = [1, 2, 3, 4]
list_of_squares = []

for n in my_list:
  list_of_squares.append(n**2)

print(list_of_squares)
```

Python provides are a more concise way to do this, called list-comprehension

```
my_list = [1, 2, 3, 4]
list_of_squares = [n**2 for n in my_list]

print(list_of_squares)
```

In the above expression, the first entry is the function that should be executed for each element of `my_list`. I personally and many experienced programmers consider this bad style. It surely will add to your geek credentials but results in hard-to-read code, with no other benefit than saving two lines. But again, you will come across this, so you need to know about it.

Also, note that list comprehensions can be combined with conditionals

`[f(x) for x in sequence if condition]`

Using a loop to modify a list

Looping over list elements with python is straightforward. However, things can get messy if your loop modifies the elements you are looping over. Try this:

```
list1 = [ 1, 2, 3, 4]
for e in list1:
```

```
    list1.remove(e)

print(list1)
```

As you can see, removing elements from `list1` could really mess with the for statement. So you need to separate the list you are iterating over from the list you are modifying. You can use the list copy method to get a copy of the list.

A better way to do this is to create a copy first

```
list1 = [ 1, 2, 3, 4]
lc = list1.copy()

for e in lc:
    list1.remove(e)

print(list1)
```

4.6.3 Functions

These modules are available as Jupyter Notebooks in:

- `PNTA-Notebooks/Functions/Functions`
- `PNTA-Notebooks/Functions/functions_assignment`

In the previous module, we used blocks of programming statements so that we can execute them repeatedly (loop statements) or based on a given condition (if statements).

Functions are another way to group program statements. However, unlike ifs and loops, functions allow us the create named code blocks. This allows us to write code that we can use repeatedly.

example

Imagine that python would have no functions to do basic math. You could define a new function called `add_numbers` (let's ignore the details here for a moment. We will walk through this later).

```
def add_numbers(a, b):
    c = a + b
    return c
```

Next you could go on and use this new function `add_numbers` to define `multiply_numbers`

```
def multiply_numbers(a, b):
    c = 0
    for i in range(b):
```

```
        c = add_numbers(c,a)
    return c
```

For good measure, let us add a power function

```
def my_power(a,b):
    c = 0
    for i in range(b): # this will only work if b = int!
        c = multiply_numbers(c,a)

    return c
```

From the above, we can see that each definition relies on the previous one and that we create new language elements for python. This is most useful for blocks that are used more than once.

Once each element has been defined, we can use it in our code:

```
b=2
e=2

p = my_power(b,e)
print(f"{b}^{e} = {p}")

```

As you can see, the above result is wrong. This demonstrates the second use for functions - we can isolate code from each other. In the above case, we can test each function independently:

```
print(f"2 + 2 = {add_numbers(2,2)}")
print(f"2 * 3 = {multiply_numbers(2,3)}")
print(f"2 ^ 4 = {my_power(2,4)}")
```

This reveals quickly that our problem is with `my_power`. In other words, functions allow us to reduce complexity by breaking the code into isolated code fragments that can be tested individually.

Functions have the following characteristics:

- They allow us to group code sequences and refer to this group by name. This is useful to declutter your code, reduce complexity, and allows us to reuse code. Most of the python statements we have used so far are functions (e.g., the print statement).
- functions allow us to extend the capabilities of our program. We could, e.g., create a function called `bprint` which will only print in bold.
- Functions allow us to isolate code sections from each other. What happens inside the function stays inside the function, and what happens outside the function stays outside. If you use, e.g., the variable `a` in the main part of your code, it will be

kept different from a variable named `a` inside a function

```
# define function
def my_test():
    a = 13
    print(f"a = {a}, at address {id(a)}")

a = 12
my_test() # run function
print(f"a = {a}, at address {id(a)}")
```

- Since functions cannot see what happens outside, we need a way to pass information to and from functions. This is done with function arguments. In the below case, the function definition includes the variables `a` and `b`. This tells us that the function expects two values and that these values will be known as `a` and `b` inside the function. The line with the `return` statement indicates that the function will return the value of c.

```
def add_numbers(a, b):
    c = a + b
    print(f"a={a}, b={b}, c={c}")
    return c

r=add_numbers(2,3)
print(r)
print(c) # this will fail, since c is not known outside the function
```

- This helps with program design because we can

divide a program into functional parts that can be tested individually. In other words, we can reduce a complex problem into a series of less complex problems!

- the **value(s)** of a variable(s) can be passed into a function as arguments to the function call (see below)
- Function arguments and returns can be any python data type.
- The results of the computations inside the function can be returned to the calling code with the return statement.
- Functions must always be defined before you can use them. This is best done at the beginning of the code!
- Functions should always return a value

```
# bad use of a function
def add_numbers1(a, b):
    c = a + b
    print(c)
```

```

add_numbers1(2,3)

# clean use of a function
def add_numbers2(a, b):
    c = a + b
    return c

print(add_numbers2(2,3))
```

A note on the return statement

In the above code examples, we use the return statement to pass the results of the function back to the calling code. Using the above example, we can assign the result of `add_numbers2` to a new variable as

```
x = add_numbers(12,13)
print(x)
```

Note, however, that the return statement will exit the function immediately, and all code after the return statement will be ignored. Run the following example, and then explore what happens if you move the return statement inside the loop block:

```
def test_return():

    a = 0
    for e in range(3):
        print(f"e = {e}")
        a = a + e

    return a


result = test_return()
print(result)
```

A practical example

Think of the following problem: You want to write an application that converts a mineral name into its chemical formula. We can divide this problem into the following functional parts:

1. get user input
2. interpret the user input and find the chemical formula (or create an error message)
3. provide the result to the user

If we divide this problem with functions, we can write and test the first part even if we have no idea what to do about numbers 2 & 3. The same goes for #2. You can develop and test the code for #2; even so, you are entirely ignorant about #1 (#3).

Now, consider you are working in a team. Likely, you would distribute the tasks along the functional blocks. But this scenario also highlights an interesting problem. You need to agree on what kind of data team #1 will provide to team #2 and what team #2 will provide to team #3. So clearly, this requires a bit of planning and, more importantly, good documentation.

Let's do an actual example. We define a function with the `def` keyword, followed by the function name and a pair of brackets() with the usual colon symbol to denote the start of a block

```
def lookup_chemdata():
    # add your code here
    pass
```

the way this is written, this function would not know anything about the data which exists outside the function. So let's write it in a way that we pass on some data from the outside world. This is done by adding one (or more) function arguments

```
def lookup_chemdata(arg1):
    # add your code here
    pass
```

You can now call your function from a program, e.g.,

```
lookup_chemdata("Barite")
```

So the string "Barite" would become the argument to the function, and inside the function, this argument would be available through the variable `arg1`. Note the actual name does not matter. You could also write

```
def lookup_chemdata(n):
    # add your code here
    pass
```

and then use `n` inside the function. So far, our function does not much, and most importantly, it does not pass any value back to the calling program. So let's add a return statement

```
def lookup_chemdata(n):
    if n == "Barite":
        f = "BaSO4"
    else:
        f = "Mineral not found"
```

```

    return f
```

now you can call the function like this

```
r = lookup_chemdata("Barite")
print(r)
```

The result of the function will be stored in the variable to the left of the equation sign (i.e., `r`). So far, so good. However, what is missing is some documentation on what the function does and some documentation on what type of data the function expects and returns.

Docstrings We solve the first problem by adding a doc-string , . A docstring is a piece of explanatory text which we add at the beginning of the function definition. Unlike a regular comment, we enclose this text with three quotation marks

```
def lookup_chemdata(n):
    """This function takes a mineral name, and returns its respective
    chemical formula. If the mineral name cannot be found, the
    function will return an error message.

    Example:

          f = lookup_chemdata("Barite")
    """

    f = f"Unable to find data for {n}"

    if n == "Barite":
        f = "BaSO4"

    return f
```

and now we use the python help system to find out what this function does

```
help(lookup_chemdata)
```

Much better! Note that comments are meant for people who read your code, while docstrings are meant for people who use your code. To read all about docstrings, see this link:

https://realpython.com/documenting-python-code/

However, what is still missing is some information for our fellow coders on what type of data the function is expecting and what it will return to the calling code. We could (and some people do), explain all this in the doc-string. But python provides a more compact and elegant way.

Type hinting Unlike many other computing languages, python does not force you to declare that the variable is of type integer, or float. However, nothing prevents us from annotating our variables to clarify what we mean (this works for python 3.5 and higher). Line 1 shows how to annotate using a comment, but line 2 requires much less typing

```
1 a = 12         # an integer value
2 a: int = 12
```

We used this before, but let's review how type hints work. In the above example, we added a colon after the variable name and then used the keyword `int` to document that `a` should be an integer value. Python provides the following keywords for its basic data types

```
a: int
b: float
g: list
h: dict
k: set
m: tuple
t: str
```

We can use this syntax to annotate our function, and now reading the code, it is more obvious what you are trying to do.

```
def lookup_chemdata(n: str) -> str:
    """This function takes a mineral name, and returns its respective
    chemical formula. If the mineral name cannot be found, the
    function will return an error message.

    Arguments: lookup_chemdata(n: str) -> str:

    Example:

          f = lookup_chemdata("Barite")
    """

    f :str = f"Unable to find data for {n}"

    if n == "Barite":
        f = "BaSO4"

    return f
```

It was evident from the function definition that it expects a string and will return a string.

- **From now on, we will use docstrings for each and every function we create**

- **From now on, we will use type hinting in all of our scripts and programs**
- If you want to read more on why type hinting is essential, read the story on Dropbox (which is written in python)

Lastly, let's improve our function by using a dictionary to demonstrate a more complex type hinting example. Type hint support in the current python version is not yet fully implemented, so writing something `->dict[str, str]:` will fail.

However, we can enable features that will become available in the next python version by loading an additional library (see line 1 in the code below). We will learn how to work with libraries in a later module. For now, simply include the import statement at the beginning of your code.

```
1  from __future__ import annotations # enable full type hinting support
2
3
4  def lookup_chemdata(key: str) -> str:
5      """This function takes a mineral name, and returns its respective
6      chemical formula. If the mineral name cannot be found, the
7      function will return an error message.
8
9      Arguments: lookup_chemdata(n: str) -> str:
10
11     Example:
12
13           f = lookup_chemdata("Barite")
14     """
15
16     value: str = "This should not happen"
17
18     # define dictionary
19     database: dict[str, str] = {
20         "Barite": "BaSO4",
21         "Pyrite": "FeS2",
22     }
23
24     # test if key is known in database
25     if key in database:
26         value = database[key]
27     else:  # return an error message
28         value = f"{key} is not in the database. Typo?"
29     return value
30
31
32 print(lookup_chemdata("Barite"))
```

This is an excellent example of what you can do with a dictionary (test it out by calling `lookup_chemdata` with various values. Note that if your python version is below 3.9, you need to import the typing module to enable support for improved type hinting.

Functions and variable scope

Earlier, we talked about how functions allow us to isolate code. So let's explore this in more detail. Before running the following code, take a moment to predict the outcome.

```
1  # define function
2  """ test function. It returns nothing
3  """
4
5
6  def my_function() -> None:
7      a = a + 2
8
9
10 # now lets use the function in our own code
11 a: int = 12
12 my_function()
13 print(a)
```

Ugh, the dreaded "UnboundLocalError." This tells you that you used variable `a` in line 5 before you defined `a`. This happens because `a` was assigned the value of twelve, outside the function, and the function has no access to the values outside the function

Conversely, whatever you define inside the function is not available outside the function:

```
# define function
""" test function. It returns nothing
"""
def my_function()-> None:
    b: int = 12
    b = b + 2

# now lets use the function in our own code
my_function()
print(b)
```

There are ways around this by defining a variable as `global`. However, this is bad style and should be avoided like the plague.

So let's implement our function the correct way:

```
# define function
""" This function adds 2 to any number you pass to this function
"""
```

```python
def my_function(c) -> int:
    c = c + 2
    return c


# now lets use the function in our own code
a: int = my_function(5)
print(a)
```

Function calls can be nested The output of a function can be used as the input to another function. You already know the print function. So we can use the output of `my_function` as input to the print function:

```python
a: int = 5
print(my_function(a))
print(f"adding 2 to {a} results in {my_function(a)}")
```

Do's and do not's

As with all things code, there are better ways, and there are ways to shoot yourself in the foot. Here is a perfectly ok way to create a function that capitalizes a string:

```python
def my_cap(s: str) -> None:
    """
    This function takes s:str and converts all characters to capitals
    """
    print(s.upper())


lc: str = "This is important"
my_cap(lc)
```

However, in almost all cases, a function should take one or more value(s) and return one or more value(s). So a better way of doing this would be

```python
def my_cap(s: str) -> str:
    """
    This function takes a str and converts all characters to capitals
    """
    return s.upper()


lc: str = "This is important"
print(my_cap(lc))
```

This solution is better because it separates the printing from the conversion, and thus keeps `my_cap` fairly universal. We can now, e.g., write

```
ld :str = "--- this not so much"
print(my_cap(lc), ld)
```

You could achieve this with the previous definition as well, but it would be more convoluted.

Return values

All functions should return a value. As with function parameters, you return the value (aka memory address) and not the variable name:

```
def my_add(a: int, b: int) -> int:
    c = a + b
    return


# This will fail since c is only known inside the function
my_add(12, 3)
print(c)

# this will work since assign the return value to x
x: int = my_add(12, 3)
print(x)

# this will also work since print will take the return value as input
# see nested functions
print(my_add(12, 3))
```

Functions with multiple return values

There is nothing special about functions that return more than one value. If you look carefully, you see that the return argument is now a tuple.

```
from __future__ import annotations

def foo(a: float) -> tuple(float, float):
    x = a
    y = a * 2
    return (x, y)


# code
v: float = 2
k = foo(v)
```

```
print(f"k[0] = {k[0]}, k[1] = {k[1]}")
```

python is pretty smart about unpacking tuples, so we can also write

```
k, l = foo(v)
print(f"k = {k}, l = {l}")
```

So multiple values are returned as a tuple. It would be nice if we could add the type of data inside the tuple to the type hint. @@latex:. This will work out of the box with Python 3.11, but for now, you must now import the 'annotations' module. Probably a good idea to add the import statement to all your code.

```
from __future__ import annotations

# define funnction foo
def foo(a: float) -> tuple(float, float):
    x = a
    y = a * 2
    return (x, y)


# start of code
# define all variable we are using
v: float = 2
k: float
l: float
# call function foo
k, l = foo(v)

# print resulty
print(f"k = {k}, l = {l}")
```

Recursive functions

Functions can call themselves. For certain problem sets, this can be a rather elegant way of coding. However, python is not well suited to recursive programming - so we will not use it in our course. However, for good measure, it should at least be mentioned. The following example is a bit construed but demonstrates the principle.

```
from __future__ import annotations

def div2(n: float, c: int) -> tuple(float, int):
    """This function divides n by 2 and will do so until the result is
    smaller than 1. In other words, this function returns the number
    of times an integer value can be divided by two.

    """
```

```
    n = n / 2

    if n >= 1:
        c = c + 1
        (n, c) = div2(n, c)

    return (n, c)


# start of main code
x: float = 8
i: int = 0

nt: int
nb: float
nb, nt = div2(x, i)
print(f"{x} can be divided by two {nt} times")
```

5 Coding Projects

5.1 Coding as a Creative Process

This module is available as Jupyter Notebooks in:

- `PNTA-Notebooks/Puzzles/puzzles`
- `PNTA-Notebooks/Puzzles/puzzles_assignment`

In the previous chapters, we reviewed the various elements of a programming language (variables, conditionals, loops, etc .). However, more often than not, the challenge is not the mastery of those elements but rather the skill to combine them in a useful manner.

Coding requires you to solve a problem for which you see no obvious solution. And the only way out is to hit the "ok, let's be creative button". For some of you, this will work. For others, not so much, since few of us are trained or gifted creative thinkers. That being said, there are strategies you can employ that will help you to find solutions. You will need those strategies in pretty much all of the following coding projects.

This chapter is inspired by "Think like a Programmer" by V. A. Spraul. Unlike most programming books, it focuses on problem-solving rather than computing. Unfortunately, all its exercises are in `C++`, but the introductory and conclusions chapters are highly recommended reading.

The book's basic premise is that most students do not struggle with a computing language per se but rather with the process of using a program to solve a problem. The following chapter is a heavily condensed version of Chapter 1 in Spraul's book.

5.1.1 General Problem-Solving Strategies

Problem-solving skills vary tremendously between people, and it is not something you can learn like any other method because it is an inherently creative process, not a methodical one. However, you will get better with practice, and a few guidelines can help with the process. The following list will prove helpful whenever you struggle with a problem where there is no obvious solution:

Make have a plan: Even if you don't know what is going on, you can always write down a plan. Think of mapping, you may not have the faintest clue of what you are dealing with, but you know that by methodical mapping (taking stations, measure strike and dip, map perpendicular to strike, etc.), you will zero in on the solution. The same is true for programming. Use the following steps as a guideline to develop a plan. Start by making a list of steps you need, even though you may not know how to solve each step. You will likely have to refine your ideas as you go, but this is still infinitely more useful than just random exploration.

1. **Restate the Problem:** An excellent way to achieve this is to restate the problem in your own words. Next, explain how you see the task to one of your peers (or your instructor) for comments. This is often the first and most crucial step in formulating a plan. Remember to describe the goal, not the way.
2. **Divide the Problem:** Most complex problems can be understood as a series of smaller problems. In coding terms, we would think of loops, if-statements, etc... This is the part where you start writing high-level pseudo-code. Pseudo-code is like short form, but it is not actual code:

```
while loop to get user input
  y = input

for loop to test x>y
t = transform_magic(x,y)
print result
```

You can do this even though you may not yet know how to code the individual blocks.

3. **Start with what you know:** This is like writing an exam. Always start with the pieces you already know how to solve. In doing so, you may find essential clues on how to solve the pieces you don't know.
4. **Reduce the Problem:** Say you have to write code that converts any number into its binary notation. To simplify the problem, you can state that your function will only deal with positive integers. Or you can further simplify the problem to the point that your code will only deal with numbers between 0 and 10 etc. Once you have an intermediate solution, you will have gained much clarity on how to solve the entire problem or what specific questions to ask your peers or instructor.
5. **Look for Analogies:** Often, you may be able to reduce your problem so that you can recognize that you already solved a similar problem elsewhere. Consider the case of strings, which are just a specific type of lists. The exact commands may not match, but now you probably remember how to query the methods associated with a list object or that you need to ask the specific question "how do I replace the third element in a string".
6. **Experiment:** Often, it is useful to test the workings of a command or code sequence in a separate cell. The key techniques here are the idea to isolate the code snippet from your main code, and to test whether it does what you think it does. E.g., consider the following `a[3]: str = 4`. This may work well in your code, but it may not do what you expect. So a little experiment can go a long way... Similarly, test and explore new commands before adding them to your code.
7. **Don't get frustrated:** When you are frustrated, you won't think clearly, and

everything will take twice as long. Take a walk, rethink the problem (i.e., use the above steps), and come back to the problem when you are ready. Then, you can try to reduce the problem to it's most fundamental form, e.g., take a piece of the code out of your program and run it in isolation; use print statements to check whether your variables have indeed the values you think they have. Once you have a minimal example that shows the problem, ask for help. Or if all else fail, have a look at your plan and see if you can work on something else. If need be, take a creative break and work on something else for a while. To quote from Sprauls book:

When you allow yourself to get frustrated - and I use the word "allow" deliberately - you are, in effect, giving yourself an excuse to continue to fail. Suppose you're working on a difficult problem and you feel your frustration rise. Hours later, you look back at an afternoon of gritted teeth and pencils snapped in anger and tell yourself that you would have made real progress if you had been able to calm down. In truth, you may have decided that giving in to your anger was easier than facing the difficult problem.

5.2 Binary to decimal conversion

This module is available as Jupyter Notebook in:

- `PNTA-Notebooks/coding-projects/bin_2_dec/`

5.3 Decimal to binary conversion

This module is available as Jupyter Notebook in:

- `PNTA-Notebooks/coding-projects/dect_2_bin/`

5.4 The bubble sort algorithm

This module is available as Jupyter Notebook in:

- `PNTA-Notebooks/coding-projects/bubble_sort/`

6 Working with libraries

Python enables us to group code via block statements (`if`, `while`, `def` etc.). However, at times you want to refer to a group of code statements. This could be a named python program (as opposed to the code cells we run in a notebook) or a collection of function definitions that provide new language elements (aka library or module). The popularity of python rests, in fact, on the large number of python libraries that facilitate, e.g., seismic inversion, borehole log data analysis, statistics, graphical output etc. etc...

A python library is simply a file that contains function definitions. The key is that this file contains only function definitions and no other program code. The file must also be written in plain python; it can't be in notebook format. This is no major inconvenience since, most of the time, we only want to use the functions in the file rather than change the functions. Library development is, however, best done with a full-fledged python IDE (Integrated Development Environment).

So why would we want to use libraries:

1. Moving often-used functions into a library declutters your code.
2. Python has myriads of libraries that greatly extend the functionality of the language.
3. Another important reason why we prefer to use libraries (rather than writing our own code) is that the developers of the libraries usually spend more time optimizing the performances and improving the compatibility of the code. Also, if we encounter problems using public libraries, we often can find solutions or helpful information on the Internet. In that sense, by using the libraries, we are collaborating with the whole community.

Why you would not want to use a library:

1. Unless it's your own library, you depend on someone else. Imagine this great library you found, but it has this nasty bug and the person who wrote this library is no longer responding to e-mail.... it might be just easier to implement your desired functionality yourself...
2. Libraries can introduce considerable complexities, which may be overkill in your situation...
3. While python itself is in the public domain, third-party libraries are often under more restrictive licenses. Not a big deal for your private code, but if you work in a commercial environment, it may be a real show-stopper.

6.1 Using a library in your code

Consider the following simple library mylib.py. It only provides two functions. If you click on the link in the previous sentence, you can inspect the code, and you will see that it looks just like any other python code you have seen so far.

To use these functions, we first need to import the library, and then we can inspect its content.

```
import mylib
print(help(mylib))
```

Note, the file mylib.py is present in this module folder. However, if you have moved your assignment to the submission folder, you need to copy the file `mylib.py` as well! As you can see from the output of the above code, the `mylib` library provides two functions, `hello_world()` and `square()` Can you imagine what would happen if you load another library that provides the same functions? Yes, the earth would stop spinning, and we all would float helplessly into outer space...

Python provides an ingenious solution to avoid such undesirable outcome. Try the following:

```
import mylib
print(square(5))
```

yup, `mylib` provides `square()`, but you cannot use it by this name! To access a function that is defined inside a library, you have to write

```
import mylib
print(mylib.square(5))
```

Pure genius! Each library creates its own namespace upon import . So if two libraries define the same function, they will be known by different names! This mechanism avoids naming conflicts, but it also adds a lot of typing... There are two ways around it:

1. Often, you will only need one or two functions from a library (also called a module). In this case, you can import each function explicitly. The onus is then on you to make sure not to import functions that overwrite existing functions

```
from mylib import hello_world, square

hello_world()
print(square(6))
```

1. Or, you can create a library alias (aka short name). This is the preferred approach as it avoids the accidental redefinition of existing functions.

```
import mylib as ml # ml is now the shorthand for mylib

ml.hello_world()
print(ml.square(6))
```

6.2 Pandas

These modules are available as as Jupyter Notebooks in:

- PNTA-Notebooks/working_with_libraries/Pandas/accessing_data/accessing_-data.ipynb
- PNTA-Notebooks/working_with_libraries/Pandas/accessing_data/accessing_-data_assignment.ipynb
- PNTA-Notebooks/working_with_libraries/Pandas/working_with_pandas_series/working_-with_pandas_series.ipynb
- PNTA-Notebooks/working_with_libraries/Pandas/working_with_pandas_series/working_-with_pandas_series_assignment.ibynb

alias `pd`. So all of the functions provided by pandas are available as `pd.function_name()`. We will explore how to use some of the Pandas provided functions below. Since pandas is a system library, you do not need a local copy in each working directory.

To read the excel file , we need to know its name, and we need to know the name of the data sheet we want to read. If the read operation succeeds, the read data will be stored as a pandas data frame object. You can think of the Pandas data frame as a table with rows, columns, headers, etc. Pay attention to the way I use type hinting to indicate that `os_peak` is a variable that contains a data frame.

```
import pandas as pd # inport pandas as pd

# define the file and sheet name we want to read. Note that the file
# has to be present in the local working directory!
fn :str = "Yao_2018.xlsx" # file name
sn :str = "outside_peak"  # sheet name

# read the excel sheet using pandas read_excel function and add it to
os_peak :pd.DataFrame = pd.read_excel(fn, sheet_name=sn) # the pandas
                                                         # dataframe
```

6.2.1 Working with the pandas DataFrame object

In most cases, your datasets will contain many lines. So listing all of this data is wasteful. Pandas provides the `head()` and `tail()` methods that will only show the first (or last) few lines of your dataset. Remember that methods are bound to an object, whereas functions expect one or more variables as an argument.

Since line 9 above created a pandas data frame object with the name `os_peak`, the `head()` and `tail()` methods are now available through the data-frame object. If this does not make sense to you, please speak up! Otherwise, try both methods here:

```
print(os_peak.head())
```

If you are really on the ball, you may have noticed that the first column is not present in the actual excel file. The numbers in the first row are called the index. Think of them as line numbers. All pandas objects show them, but they are ignored when you do computations with the data. So we do have an index column, and then we have data columns.

Selecting specific columns

We can select specific columns by simply specifying the column name:

```
print(os_peak["d34S"])
```

Selecting specific row & column combinations

The above syntax is intuitive but not very flexible. Pandas provides the `iloc()` method (integer location), which allows us to access rows and columns by their index

```
# iloc[row,col]
print(os_peak.iloc[1,0]) # get the data in the 2nd row of the 1st col
```

You remember the slicing syntax (if not, review the slicing module). so if you want to see the first two rows of the third column:

```
print(os_peak.iloc[0:2,3])  # get the first 2 rows from the 4th column
```

order to get all data from the third column, you can write

```
print(os_peak.iloc[:,2]) # get all data from the third column
```

Now modify the above statement that you print all data from the second row.

Selecting rows/columns by label

Pandas also supports the selection by label rather than index. This is done with the `.loc(row_label,column_label)` method. The first argument is the row label, and the second is the column label. However, the statement below requires **some attention**. At first sight, it appears that we mix `iloc()` and `loc()` syntax here. However, this is not the case. Rather, this command treats the index column as a label. So if your first index number starts at 100, this code would yield no result since there is no label called "2".

As a side note, the index does not even have to be numeric, it could well be a date-time value, or even a letter code. So `loc[2:4,'d34S']` does not use slicing notation, rather, it means as long as the label is equal to 2, 3 or 4. This difference is illustrated by the following code.

```
print(os_peak.iloc[2:4,3]) # extract index values which are >= 2 and <4
print(os_peak.loc[2:4,'d34S']) # extract the d34S data for index
                               # labels that equal 2, 3, or 4
```

There is a third method to select a column by label

```
import pandas as pd
all_data pd.DataFrame = pd.read_excel("Yao_2018.xlsx", sheet_name="combined_data") # the panda
delta = all_data.d34S # this will work
# age = all.data.Age [Ma] # this will not work
```

This will only work if your column headers are single words only. I thus recommend sticking to the `iloc()` method.

Include only specific rows

Since the `loc()` will match against the value of a given cell, we can use this to select only specific rows. The third worksheet in the excel file contains an additional column called 'Location'. Each row in this column contains the string "in" or "out" (download the excel file to check this out). In this example, we can use this column to only select data where the Location equals the string "in"

```
import pandas as pd
all_data: pd.DataFrame = pd.read_excel(
    "Yao_2018.xlsx", sheet_name="combined_data"
)  # the panda
y: pd.Series = all_data.loc[all_data["Location"] == "in", "d34S"]
print(y)
```

Getting statistical coefficients

Pandas supports a large number of statistical methods. As an example, use the `describe()` method which will give you a quick overview of your data.

```
print(os_peak.describe())
```

What else can you do?

The short answer is lots! The data frame can act as a database, and you can add/remove, values/columns/rows, you can clean your data (e.g., missing numbers, bogus data), you can do boolean algebra, etc., etc. If these cases arise, please have a look at the excellent online documentation and tutorials. A good start is

https://towardsdatascience.com/30-examples-to-master-pandas-f8a2da751fa4

6.2.2 Working with the Pandas Series Object

First, we import a dataset to play with. As a new twist, I will also use a library that can test whether a file is present or not. This is because I spent 20 minutes debugging my code without realizing that I had the wrong filename. So I decided to include this here. It may prove useful. The code is straightforward and simply raises an error if the file is not present. To test whether the file is present, we import the pathlib-library, which imports tools to deal with file and path names (e.g., checking whether a file is present). I recommend using this code snippet in your submissions. Recall that to locate a file, you need to know the filename and the directory where to find the file.

The following piece of code first retrieves the current directory, and then builds a path object from the current working directory plus the filename.

```python
import pathlib as pl

# define the file and sheetname we want to read. Note that the file
# has to be present in the local working directory!
fn: str = "example_file.csv"  # file name

# get the current working directory
cwd :pl.Path = pl.Path.cwd()
print(f"the current working directory is\n \t {cwd}\n")

# build the fully qualified file name (i.e. path + filename)
fqfn :pl.Path = pl.Path(f"{cwd}/{fn}")

# this little piece of code could have saved me 20 minutes
if not fqfn.exists():  # check if the file is actually there
    raise FileNotFoundError(f"Cannot find file {fqfn}")
else:
    print(f"The fully qualified filename is\n \t {fqfn}\n")
    print(f"fqfn is of type {type(fqfn)}")
    df: pd.Dataframe = pd.read_csv(fqfn) # read csv data
```

So try this with a filename you know does not exist to see what will happen!

As you can see, there is quite a bit of typing, involved. I thus keep a repository of code snippets I use a lot, so I can simply use cut/copy/paste the next time I need to read a file. You could, e.g., use something like this:

```
# this is template, it will not run unless you add useful variable values!
import pathlib as pl
import pandas as pd

fn: str = ""  # file name
cwd :pl.Path = pl.Path.cwd() # get the current working directory
fqfn :pl.Path = pl.Path(f"{cwd}/{fn}") # fully qualified file name

if not fqfn.exists():  # check if file exist
    raise FileNotFoundError(f"Cannot find file {fqfn}")
```

Collect these snippets in a separate notebook, and they will save you a lot of time.

Working with the pandas series data object

The Pandas data frame is composed of smaller units, the so-called pandas series object. You can think of them as columns in a spreadsheet. You already used this data type implicitly in our last module, but here we will explore it more fully. The above code should have created a data frame already. Let's make sure it worked

```
print(df.head())
```

Now, let's extract the data from column `A`, and execute the following

```
A :pd.Series = df['A'] # extract column A, and save as pandas data series
B :pd.Series = A * 2   # multiply A with 2

# now we create a new dataframe from the two pandas series objects if
# you don't understand the syntax, go back to the "other data types"
# module and check out what I am doing here. If you cannot figure it
# out, speak up!
result :pd.DataFrame = pd.DataFrame({'A': A, 'A*2 =': B})
result.head()     # multiply A with 2 and print the result
```

Note in the above example the use of `pd.DataFrame` On the left, it is used as a type-hint, whereas on the right side of the assignment, it is used as a function to create a new data frame from a dictionary.

You probably remember that it was not possible to multiply a list with a number because lists can contain numbers, letters, other lists, tuples etc. To do this, you had to write a loop. A Pandas series, on the other hand, can only have one data type per column. In other words, a column can contain strings, integers, floats etc., but all entries in a given column must be of the same type. Since column `A` consists only of numbers, python can directly multiply each element with 2. What happens if you multiply `A*B`? Is the multiplication element by element, or do you get the cross product? What happens if you write A**B? Hurray! no more loops! (kinda...)

In a way, python treats a pandas series object like a vector, not unlike Matlab. There is some cool stuff we can do with this. We can e.g., apply a comparison operator to a pandas series

```
print(A>2)
```

Now, why would this be useful? Remember that False equals zero, whereas True equals 1. So if you want to count the number of values in A that are larger than 2, you can simply write

```
n :int = sum(A>2)
print (n)
```

neat! Please see the corresponding Jupyter Notebook in your Syzygy Jupyter account. See the beginning of this chapter to get the path to the assignment notebook.

- if you have a UofT account use this link:
  https://utoronto.syzygy.ca/jupyter/hub/user-redirect/git-pull?repo=https://github.com/uliw/PNTA-Notebooks&urlpath=tree/PNTA-Notebooks/&branch=main

6.3 Matplotlib - Plotting Data

These modules are available as as Jupyter Notebooks in:

- PNTA-Notebooks/working_with_libraries/Matplotlib/plotting_data.ipynb
- PNTA-Notebooks/working_with_libraries/Matplotlib/plotting_data_assignment.ipynb

Creating graphical output with python is typically achieved by using third-party libraries. Each of these libraries has its own way of describing graphical output, which can be very confusing. Furthermore, some libraries provide an object-oriented interface as well as a procedural interface. And to top it all, some libraries like pandas implement their own interface to some of the graphics libraries (and vice versa ....). So if you start looking for help, you can find all sorts of messy/conflicting advice.

Matplotlib is widely used in the sciences, and the interface is similar to Matlab. . Because of this historical connection, matplotlib.pyplot provides two different interfaces. I will show a trivial example of the procedural interface below, but for the remainder of the course, we will use matplotlib.pyplot with the object-oriented interface.

First, let's import some data

```
import pandas as pd  # import pandas as pd
import pathlib as pl

```

```
fn: str = "Yao_2018-b.xlsx"  # file name
sn: str = "both"  # sheet name
cwd :pl.Path = pl.Path.cwd()
fqfn :pl.Path = pl.Path(f"{cwd}/{fn}")

if not fqfn.exists():  # check if the file is actually there
    raise FileNotFoundError(f"Cannot find file {fqfn}")

os_peak: pd.DataFrame = pd.read_excel(fqfn, sheet_name=sn)

print(os_peak.head()) # do a visual confirmation
# extract the data we need for plotting
Depth: pd.Series = os_peak.iloc[:, 1]
Age: pd.Series = os_peak.iloc[:, 2]
d34S: pd.Series = os_peak.iloc[:, 3]
d34Serror: pd.Series = os_peak.iloc[:, 4]
```

Now that we have data to play with, creating plots is straightforward and only requires a certain sequence of commands. However, the matplotlib library provides two different interfaces. The older, and simpler procedural interface and the modern object-oriented interface. In this course, we will use the object-oriented style. However, many online examples still use the older interface, so caveat emptor!

6.3.1 The procedural interface

In the following example, I give a typical command sequence.

The matplotlib library is large and thus consists of several modules that address various graphing needs (see `https://matplotlib.org/`). Matlab style X-Y data plotting is handled by the pyplot module. Have a look at the import statement to see how it specifies that we will use the pyplot module.

Line 4 creates a scatter plot (i.e., each coordinate pair is represented by a point), line 5, saves the figure as a pdf file, and line 6 makes the plot appear. You may ask, why would we need a command to make the plot appear? Plotting can be computationally expensive if you have a large dataset. So you don't want to redraw your plot window each time you change a label etc. So we first create all plot elements and then request that the plot is rendered with the `plt.show()` command.

```
1 # plotting with the procedural interface
2 import matplotlib.pyplot as plt
3
4 plt.scatter(Age,d34S)      # create a scatter plot
5 plt.savefig(f"{cwd}/test.pdf")
6 plt.show()
```

Using the procedural interface is straightforward but limited. Imagine you have two plots in the same figure. If you try to set the x-axis label with `plt.xlabel` which of the two x-labels will you change?

6.3.2 The object-oriented interface

Using the object-oriented approach avoids this problem quite elegantly. However, the matplotlib community uses confusing terminology. First, we create (instantiate) a canvas object. Think of it as the page holding one or more of your figures. Weirdly enough, this object is not called `canvas`, but `Figure`. Next, we need to create an object that holds the actual plot. The logical name for this would be `figure`, but alas, matplotlib calls the actual figure `axes`. Have a look at this example:

```
# Create a canvas with one figure object
import matplotlib.pyplot as plt

fig: plt.Figure  # this variable  will hold the canvas object
ax1: plt.Axes  # this variable will hold the axis object

fig, ax1 = plt.subplots()  # create canvas and axis objects

ax1.scatter(Age, d34S)  # Create a scatter plot for ax

plt.show()
fig.savefig("plt_scatter.pdf")

```

So what happens here: The `subplots()` method creates a canvas (aka `fig`), as well as a figure object (aka `ax1`). The canvas contains everything (potentially more than one figure), and the axes object has all the data, labels etc.... Since we want to plot a scatter plot, we call the `scatter` method of the axes object.

Note, that there is nothing magical about the variable names, we might as well write

```
canvas, figure = plt.subplots()
```

However, everyone else uses `fig`, and `ax`, so this will become confusing once you look up online materials.

To me, it would be logical if the canvas handle would also be used to save and show the figure. Alas, this is not the case. The `fig.show()` method exists but has a slightly different meaning than the `plt.show()` function. Going forward, we will use the `plt.show()` function.

One more caveat: matplotlib knows the `plt.subplots()` and the `plt.subplot()` functions. Note, that `plt.subplots()` is very different from `plt.subplot()`. Use the help system to see just how different their meaning really is.

A note on using type hints with matplotlib

As of 2022, type-hints have not been implemented for the matplotlib library. Running the above code through a type checker will result in an error. However, since type hints are ignored by python, we can still leverage its syntax to clarify what we are doing.

Placing more than one graph into a figure

```
import matplotlib
import matplotlib.pyplot as plt

fig: plt.Figure  # figure (canvas)
ax1: plt.Axes  # first plot object
ax2: plt.Axes  # second plot object

# Create a figure with 1 row of 2 plots
fig, [ax1, ax2] = plt.subplots(nrows=1, ncols=2)  #

ax1.scatter(Age, d34S, color="C0")  # Create a scatter plot for ax
ax2.plot(Age, d34S, color="C1")  # Create lineplot for ax 2
plt.show()
```

Note that the resulting plots might overlap. See the "Visual Candy" section below for how to remedy this. And now the same for four figures:

```
import matplotlib.pyplot as plt

fig: plt.Figure
ax1: plt.Axes
ax2: plt.Axes
ax3: plt.Axes
ax4: plt.Axes

# Create a figure canvas with 2 plot objects
fig, [[ax1, ax2], [ax3, ax4]] = plt.subplots(nrows=2, ncols=2)  #

ax1.scatter(Age, d34S, color="C0")  # Create a scatter plot for ax
ax2.plot(Age, d34S, color="C1")  # Create lineplot for ax 2
ax3.scatter(Age, d34S, color="C2")  # Create a scatter plot for ax
ax4.plot(Age, d34S, color="C3")  # Create lineplot for ax 2
plt.show()
```

Note how the axis data returned by `plt.subplots()` function reflects the output geometry by returning a list of lists! In other words, row elements are returned as a list of axis handles, and those lists are elements of another list. So we could rewrite the code more elegantly as

```python
from __future__ import annotations
import matplotlib.pyplot as plt

fig: plt.Figure # canvas object
ax: list[list[plt.Axes]] # list of axes objects

# Create a figure canvas with 2 plot objects
fig, ax = plt.subplots(nrows=2, ncols=2)  #

ax[0][0].scatter(Age, d34S, color="C0")  # Create a scatter plot for ax
ax[0][1].plot(Age, d34S, color="C1")  # Create lineplot for ax 2
ax[1][0].scatter(Age, d34S, color="C2")  # Create a scatter plot for ax
ax[1][1].plot(Age, d34S, color="C3")  # Create lineplot for ax 2
plt.show()
```

6.3.3 Controlling figure size

Using the canvas (fig) handle, we can control almost any aspect of our figure (try and query the fig-handle with `help()`). Here we use the command in line seven to set the figure size to 6 by 4 inches

```python
1  import matplotlib.pyplot as plt
2
3  fig: plt.Figure
4  ax: plt.Axes
5
6  fig, ax = plt.subplots()  #
7  fig.set_size_inches(6, 4)
8
9  ax.scatter(Age, d34S)  # Create a scatter plot for ax
10
11 fig.savefig("test_figure.pdf")
12 plt.show()
```

6.3.4 Labels, title, and math symbols

Now, lets add a few bells and whistles to our plot

```python
import matplotlib.pyplot as plt

fig: plt.Figure
ax: plt.Axes

fig, ax = plt.subplots()
fig.set_size_inches(6, 4)
```

```

ax.scatter(Age, d34S)

ax.set_title("My first plot")
ax.set_xlabel("Age [Ma]")
ax.set_ylabel("d34S")

plt.show()
```

Lots of typing, but otherwise straightforward.

Math symbols

We can also add math symbols to the text. Matplotlib understands most LaTeX math symbols and will render them correctly if they are enclosed in dollar signs (see below). See this link for a fairly complete list.

```
import matplotlib.pyplot as plt

fig: plt.Figure
ax: plt.Axes

fig, ax = plt.subplots()
fig.set_size_inches(6,4)

ax.scatter(Age,d34S)

ax.set_title("My first plot")
ax.set_xlabel("Age [Ma]")
ax.set_ylabel("$\delta^{34}$S [$^0/_{00}$ VCDT] ")

plt.show()
```

6.3.5 Legends, and arbitrary text, and visual clutter

This is now straightforward. The only noteworthy thing is that the location of, e.g., the arbitrary text, is given the local coordinates of the data.

```
import matplotlib.pyplot as plt

fig: plt.Figure
ax: plt.Axes

fig, ax = plt.subplots()
fig.set_size_inches(6,4)

```

```
ax.scatter(Age,d34S)

ax.set_title("My first plot")
ax.set_xlabel("Age [Ma]")
ax.set_ylabel("$\delta^{34}$S [VCDT] ")
ax.legend(["Yao et al. 2018"])            # The legend
ax.text(55,18.75,"Some text")          # Some arbitrary text

plt.show()
```

The black frame around the figure (and legend), is a figure element without function. A better word is visual clutter, which distracts from the actual information. Let's remove the elements we do not need. For the legend, we can simply add an option to suppress the frame, and for the top and right spine, we can render them invisible by specifying a non-color.

```
import matplotlib.pyplot as plt

fig: plt.Figure
ax: plt.Axes

fig, ax = plt.subplots()
fig.set_size_inches(6, 4)

ax.scatter(Age, d34S)

ax.set_title("My first plot")
ax.set_xlabel("Age [Ma]")
ax.set_ylabel("$\delta^{34}$S [VCDT] ")
ax.legend(["Yao et al. 2018"], frameon=False)
ax.text(55, 18.75, "Some text")

# remove unneeded frame elemets
ax.spines["right"].set_color("none")  #
ax.spines["top"].set_color("none")

plt.show()
```

6.3.6 Adding a second data set with the same y-axis

Is as simple as calling the axis object again. Here we use the same dataset, but with a different plot method.

```
import matplotlib.pyplot as plt

fig: plt.Figure
ax: plt.Axes
```

```
fig, ax = plt.subplots()
fig.set_size_inches(6,4)

ax.scatter(Age,d34S)
ax.plot(Age,d34S)

ax.set_title("My first plot")
ax.set_xlabel("Age [Ma]")
ax.set_ylabel("$\delta^{34}$S [VCDT] ")
ax.legend(["Yao et al. 2018"], frameon=False)
ax.text(55,18.75,"Some text")
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')

plt.show()
```

You can control whether the scatter symbol is on top of the line, or behind the line, by using the `zorder` keyword. See https://stackoverflow.com/questions/2314379/how-to-plot-the-lines-first-and-points-last-in-matplotlib

6.3.7 Data with a shared x-axis, but an independent y-axis

At times, you will need to plot data that shares the x-axis, but has an independent y-axis (say concentration versus isotope ratio). This can be achieved by creating a twin of the x-axis of the axes object. See line 20 below:

```
1  import matplotlib.pyplot as plt
2  
3  fig: plt.Figure
4  ax1: plt.Axes
5  ax2: plt.Axes
6  
7  fig, ax1 = plt.subplots()  # create plot canvas
8  fig.set_size_inches(6, 4)  # set figure size
9  
10 ax1.scatter(Age, d34S, color="C0")  # scatter plot
11 ax1.plot(Age, d34S, color="C1")  # line plot
12 
13 ax1.set_title("Using a secondary y-axis")  # title
14 ax1.set_xlabel("Age [Ma]")  # x label
15 ax1.set_ylabel("$\delta^{34}$S [VCDT] ")  # y-label
16 ax1.legend(["$\delta^{34}$S Yao et al. 2018"], frameon=False)  # The legend wo frame
17 
18 # create new axes object which shares the x-axis, but has an
19 # independent y-axis
20 ax2 = ax1.twinx()
21 ax2.plot(Age, d34Serror, color="C2")
```

```

plt.show()  # render the figure
```

Not very pretty yet, so let's control the plot scale and add a legend

```
import matplotlib.pyplot as plt

fig: plt.Figure
ax1: plt.Axes
ax2: plt.Axes

fig, ax1 = plt.subplots()  # create plot canvas
fig.set_size_inches(6,4)  # set figure size

ax1.scatter(Age,d34S,color="C0") # scatter plot
ax1.plot(Age,d34S,color="C1")    # line plot

ax1.set_title("Using a secondary y-axis")   # title
ax1.set_xlabel("Age [Ma]")       # x label
ax1.set_ylabel("$\delta^{34}$S [VCDT] ")     # y-label
ax1.legend(["$\delta^{34}$S Yao et al. 2018"],
          frameon=False) # The legend wo frame

ax2 = ax1.twinx()
ax2.plot(Age,d34Serror,color="C2")
ax2.set_ylim([0,0.4])
ax2.legend(["Analytical error"],
           frameon=False)
plt.show()                           # render the figure
```

Somewhat better, but the legend is messy since each axis object has its own legend. There are a couple of ways around this, and they are summarized in this StackOverflow post:

- secondary-axis-with-twinx-how-to-add-to-legend

for our purposes, it is enough to position the legends manually. The location information is given in percent relative to the lower-left corner. A bit of a hack, but it works.

```
import matplotlib.pyplot as plt

fig: plt.Figure
ax1: plt.Axes
ax2: plt.Axes

fig, ax1 = plt.subplots()  # create plot canvas
fig.set_size_inches(6,4)  # set figure size

ax1.scatter(Age,d34S,color="C0") # scatter plot
```

```
ax1.plot(Age,d34S,color="C1")    # line plot

ax1.set_title("Using a secondary y-axis")   # title
ax1.set_xlabel("Age [Ma]")       # x label
ax1.set_ylabel("$\delta^{34}$S [VCDT] ")     # y-label

ax1.legend(["$\delta^{34}$S Yao et al. 2018"],
          loc=(0.02, 0.9),
          frameon=False ) # The legend wo frame

ax2 = ax1.twinx()
ax2.plot(Age,d34Serror,color="C2")

ax2.set_ylim([0,0.4])
ax2.set_ylabel("Error [$^{0}/_{00}$ VCDT]")
ax2.legend(["Analytical error"],
           loc=(0.02,0.82),
           frameon=False)
plt.show()                            # render the figure
```

I should also note that there is a `twiny()` command which creates an independent x-axis, and to take things even further, you can create mapping functions between paired axes. . .

6.3.8 Visual candy

The default plot style is utilitarian and not particularly pretty. The Plot style is fortunately independent of the plot commands. We can use the `plt.style.use('style')` command to achieve a specific look.

Below we use the ggplot style that is popular with the R-crowd.

```
import matplotlib.pyplot as plt

fig: plt.Figure
ax: plt.Axes

plt.style.use("ggplot")

fig, ax = plt.subplots()
fig.set_size_inches(6, 4)

ax.scatter(Age, d34S, color="C0")
ax.plot(Age, d34S, color="C1")

ax.set_title("My first plot")
ax.set_xlabel("Age [Ma]")
ax.set_ylabel("$\delta^{34}$S [VCDT] ")
ax.legend(["Yao et al. 2018"], frameon=False)
```

```
ax.text(55, 18.75, "Some text")
ax.spines["right"].set_color("none")
ax.spines["top"].set_color("none")

fig.savefig("test_figure.pdf")
plt.show()
```

See https://matplotlib.org/3.1.1/gallery/style_sheets/style_sheets_reference.html for examples of the default styles.

If you check the actual pdf file that is saved by the above code, you may find that some of the label text is cut off. To prevent this, we add the `fig.tight_layout()` command before `plt.show`. This is particularly important if you place more than one plot into a figure.

So now, we have a pretty good template we can use in our code!

```
import matplotlib.pyplot as plt

fig: plt.Figure
ax: plt.Axes

fig, ax = plt.subplots()

# plot data
ax.scatter(Age, d34S, color="C0")
ax.plot(Age, d34S, color="C1")

# plot options
plt.style.use("ggplot")
fig.set_size_inches(6, 4)
ax.set_title("My first plot")
ax.set_xlabel("Age [Ma]")
ax.set_ylabel("$\delta^{34}$S [$^0/_{00}$ VCDT] ")
ax.legend(["Yao et al. 2018"], frameon=False)
ax.text(55, 18.75, "Some text")
ax.spines["right"].set_color("none")
ax.spines["top"].set_color("none")
fig.tight_layout()
fig.savefig("test_figure.pdf")

plt.show()
```

6.3.9 Adding errors bars

Quite often, it is important to show error bars with your measurements.

```python
import matplotlib.pyplot as plt

fig: plt.Figure
ax: plt.Axes

fig, ax = plt.subplots()  # create plot canvas

# Add Scatter plot
ax.scatter(Age, d34S, color="C0")
# Add line plot
ax.plot(Age, d34S, color="C1")

# Add error bars
ax.errorbar(
    Age,  # x values
    d34S,  # y values
    yerr=d34Serror,  # y-error, single value or list of values
    color="C0",  # color
    fmt="none",
)  # none = plot only error bars

# set title and labels
fig.set_size_inches(6, 4)  # set figure size
ax.set_title("Plotting error bars")  # the plot title
ax.set_xlabel("Age [Ma]")  # x label
ax.set_ylabel("$\delta^{34}$S [VCDT] ")  # y-label
ax.legend(
    ["$\delta^{34}$S Yao et al. 2018"], loc=(0.02, 0.9), frameon=False
)  # The legend wo frame
ax.spines["top"].set_color("none")  # do not display the right spine
fig.tight_layout()  # tighten up layout
plt.show()  # render the figure
```

6.3.10 Save yourself the typing

and add the following to your collection of code snippets. Replace/delete the labels as necessary.

```python
import matplotlib.pyplot as plt

fig: plt.Figure
ax: plt.Axes

fig, ax = plt.subplots()

# plot data
# ax.scatter(x, y, color="C0")
```

```
ax.plot(x, y, color="C1")

# ax.set_xlim((0,1))
# ax.set_ylim((0,1))

# plot options
plt.style.use("ggplot")
fig.set_size_inches(6, 4)
ax.set_title("My first plot")
ax.set_xlabel("Age [Ma]")
ax.set_ylabel("$\delta^{34}$S [$^0/_{00}$ VCDT] ")
ax.legend(["Yao et al. 2018"], frameon=False)
ax.text(55, 18.75, "Some text")
ax.spines["right"].set_color("none")
ax.spines["top"].set_color("none")
fig.tight_layout()
fig.savefig("test_figure.pdf")

plt.show()
```

6.3.11 Recap

- Matplotlib refers to the figure canvas as `figure` and to individual plot(s) within a figure as `axes`
- Matplotlib supports a procedural as well as an object-oriented interface. Commands used in both approaches, often have similar names but different meanings. Care must be taken to differentiate between both methods when perusing examples found in books or the on the internet.
- The object-oriented interface allows you to modify plot elements in great detail (there is basically no limit), but it may be tedious to so.
- Matplotlib plots can be styled see: style-sheets-reference.html
- No one can remember all the different plot commands. So it is best to keep a generic code template

6.4 Statistics

These modules are available as as Jupyter Notebooks in:

- `PNTA-Notebooks/working_with_libraries/Statsmodels/linear_regression.ipynb`
- `PNTA-Notebooks/working_with_libraries/Statsmodels/linear_regression_-assignment.ipynb`

6.4.1 Linear Regression

Causation versus Correlation

Back in my home country, and before the hippy movement changed our culture, kids, who were curious about where the babies come from, were told that they are brought by the stork (a large bird, see Fig.6.1). Storks were indeed a common sight in rural areas, and large enough to sell this story to a 3-year-old.

Sadly, as grown-up scientists with a penchant for critical thinking, we want to know if there is data to support this idea. Specifically, we should see a good correlation between the number of storks and the number of babies. Low and behold, these two variables actually correlate in a statistically significant way. Countries with larger stork populations have higher birthrates. Since both variables increase together, this is called a positive correlation. See Fig. 6.2

Now, does this prove that the storks deliver the babies? Obviously (or so we think) not. Just because two observable quantities correlate does in no way imply that one is the cause of the other. The more likely explanation is that both variables are affected by a common process (i.e., industrialization).

It is a common mistake to confuse correlation with causation. Another good example is to correlate drinking with heart attacks. This surely will correlate, but the story is more complex. Are there, e.g., patterns like drinkers tend to do less exercise than non-drinkers? So even if you have a good hypothesis why two variables are correlated, the correlation itself proves nothing.

Irrespective of a causal relationship, we can express the correlation between two datasets as the Pearson Product Moment Correlation Coefficient (PPMC, typically called `r`) to describe the strength and direction of the relationship between two variables. The PPMCC (lower case r) varies between +1 (perfect positive correlation) and -1 (perfect negative or inverse correlation). Correlations are described as weak if they are between +0.3 and -0.3, strong if they are greater than +0.7 or less than -0.7". Note, that correlation analysis makes no assumptions about the functional form between y (dependent variable) and x (independent variable). In other words, the PPMC says nothing about whether the correlation is linear, logarithmic, exponential etc.

We can use the `corr()` method of the pandas series object to calculate the PPMCC

```python
import pandas as pd  # inport pandas as pd
import pathlib as pl

fn: str = "storks_vs_birth_rate.csv"  # file name
cwd: pl.Path = pl.Path.cwd()
fqfn: pl.Path = pl.Path(f"{cwd}/{fn}")

# this little piece of code could have saved me 20 minutes
if not fqfn.exists():  # check if the file is actually there
```

Figure 6.1: The Stork. Image by Soloneying, Wikimedia Downloaded Nov 22, 2019.

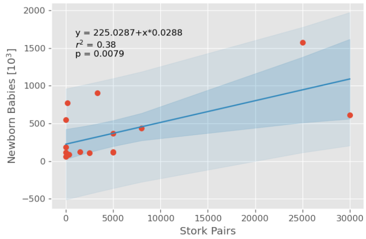


Figure 6.2: The birthrate and the number of stork pairs correlate in a statistically significant way. This analysis suggests that each stork pair delivers about 29 human babies, and that about 225 thousand babies were born otherwise. Data after Matthews 2000.

```python
    raise FileNotFoundError(f"Cannot find file {fqfn}")

df: pd.DataFrame = pd.read_csv(fn)  # read data
df.columns = ["Babies", "Storks"]  # replace colum names
b: pd.Series = df["Babies"]
s: pd.Series = df["Storks"]

print(f" r = {s.corr(b):.2f}")
```

So we can confirm that the number of babies and storks correlate.

Understanding the results of a linear regression analysis

Linear regression analysis takes correlation analysis one step further and determines how well a linear function (e.g., a straight line) can describe the relation between x and y. The equation of a straight line `y = mx +b` is fitted to the scatter of data by changing `a` and `m` in such a way that the difference between the measured data and the model prediction is minimized.

The Coefficient of Determination or r^2 expresses how well a sloping straight line can explain the correlation between the dependent (y) and a single independent (x) variable. In the above figure, $r^2 = 0.38$, which means that 38% of the new-born babies could be explained by a linear correlation with the number of storks.

In many cases, more than one independent variable is needed to explain the scatter in the data. In this case, Multiple Linear Regression is used where `y = x1 + x2 + x3....xn + b`. From this analysis (i.e. simultaneous solving for a system of linear equations – remember your Linear Algebra course!) the Multiple Coefficient of Determination (R^2) is used to express the amount of explained variation in `y` by a combination of independent variables. By default, many programs use the R^2 number even when there is only one independent variable. This can be misleading as capital R^2 should be reserved for analyses that involve multiple independent variables.

From a user perspective, we are interested to understand how good the model is, and how to interpret the key indicators of a given regression model:

r^2 or the coefficient of determination. This value is in the range from zero to one and expresses how much of the observed variance in the data is explained by the regression model. So a value of r^2=0.7 indicates that 70% of the variance is explained by the model, and that 30% of the variance is explained by other processes which are not captured by the linear model (e.g., measurements errors, or some non-linear effect affecting `x` and `y`). In Fig. 6.2 38% of the variance in the birthrate can be explained by the increase in stork pairs.

p When you do a linear regression, you state the hypothesis that `y` depends on `x` and that they are linked by a linear equation. If you test a hypothesis, you however also have to test the so-called **null-hypothesis**, which in this case would state that `y` is unrelated to `x`. The p-value expresses the likelihood that the null-hypothesis is true. So a p-value of 0.1 indicates a 10% chance that your data does not correlate. A p-value of 0.01, indicates a 1% chance that your data is not correlated. Typically, we can reject the null-hypothesis if `p < 0.05`, in other words, we are 95% sure the null hypothesis is wrong. In Fig. 6.2, we are 99.2% sure the null hypothesis is wrong. Note that there is not always a simple relationship between r^2 and p.

The statsmodel library

Python's success rests to a considerable degree on the myriad of third party libraries which, unlike MatLab, are typically free to use. In the following, we will use the "statsmodel" library, but there are plenty of other statistical libraries we could use as well.

The statsmodel library provides a few different interfaces. Here we will use the formula interface, which is similar to the R-formula syntax. However, not all statsmodel functions are available through this interface (yet?). First, we import the needed libraries:

```python
import pandas as pd  # import pandas as pd
import pathlib as pl

# define the file and sheetname we want to read. Note that the file
# has to be present in the local working directory!
fn: str = "storks_vs_birth_rate.csv"  # file name
cwd: pl.Path = pl.Path.cwd()
fqfn: pl.Path = pl.Path(f"{cwd}/{fn}")
if not fqfn.exists():  # check if the file is actually there
    raise FileNotFoundError(f"Cannot find file {fqfn}")

df: pd.DataFrame = pd.read_csv(fn)  # read data
df.columns = ["Babies", "Storks"]  # replace colum names
df.head()  # test that all went well
```

Before we perform a regression analysis, we must test if the data follows a normal distribution. There are a variety of tests to check for normality, but here we will simply use a histogram plot which either shows a bell curve distribution or not

```
import matplotlib.pyplot as plt

fig: plt.Figure
ax1: plt.Axes
ax2: plt.Axes

fig, [ax1, ax2] = plt.subplots(nrows=2, ncols=1)  #
fig.set_size_inches(6, 9)
ax1.hist(
    df.iloc[:, 0],
)
ax2.hist(df.iloc[:, 1])
ax1.set_title("Babies")
ax2.set_title("Storks")
plt.show()
```

As you can see from the histogram, our data shows anything but a normal distribution! We will use it anyway since it is a fun dataset. However, the above test is crucial if you ever want to do a real regression analysis!

For the regression model, we want to analyze whether the number of storks predict the number of babies. In other words, does the birth rate depend on the number of storks? For this, we need to define a statistical model, and test whether the model predictions will fit the data:

- The gory details of this procedure are beyond the scope of this course - if you have not yet taken a stats class, I do recommend doing so!
- There are many ways of doing this. Here we use an approach which is common in `R`

As of November 2021, there is no type hinting support for the statsmodel library. However, to distinguish between the various statsmodel object types, I use the following hints:

```
# smf.ols   ordinary least square model object
# smf.ols.fit  model results object
```

See below how this is used in code.

After importing the data, we now create a statistical model on line 16 in the code below. Pay attention to how the model is specified with the formula `"Babies ~ Storks`, which states that the number of Babies should depend on the number of storks. These names must correspond to the variable names in the data frame `df`!

Once the model is defined, we request to fit the model against the data (line 17) The results of the `fit()` method will be stored in the `results` variable. Line 18, then invokes the `summary()` method of the results object.

```python
import statsmodels.formula.api as smf
import pandas as pd  # import pandas as pd
import pathlib as pl

# define the file and sheetname we want to read. Note that the file
# has to be present in the local working directory!
fn: str = "storks_vs_birth_rate.csv"  # file name
cwd: pl.Path = pl.Path.cwd()
fqfn: pl.Path = pl.Path(f"{cwd}/{fn}")
if not fqfn.exists():  # check if the file is actually there
    raise FileNotFoundError(f"Cannot find file {fqfn}")

df: pd.DataFrame = pd.read_csv(fn)  # read data
df.columns = ["Babies", "Storks"]  # replace colum names

model: smf.ols = smf.ols(formula="Babies ~ Storks", data=df)
results: model.fit = model.fit()  # fit the model to the data
print(results.summary())  # print the results of the analysis
```

Plenty of information here, probably more than you asked for. Let's tease out the important ones:

- The first line states that `Babies` is the dependent variable. This is useful and will help you to catch errors in your model definition.
- The second line confirms that this is an ordinary least squares model
- Then, there are also a couple of warnings, indicating that your data quality may be less than excellent. But we knew this already from testing whether the data is normal distributed or not.

If you compare the output with Figure 6.2, you can see that r^2 value is called "R-squared", the p-value is called "Prob (F-statistic)", the y-intercept is the first value in the "Intercept" row, the slope is the first value in the "Storks" row. You can also extract these parameters from the model results object like this:

```python
# retrieve values from the model results
slope   :float = results.params[1]  # the slope
y_0      :float = results.params[0]  # the y-intercept
r_square :float = results.rsquared    # r_square
p_value  :float = results.pvalues[1] # the p_value
```

Adding the regression line and confidence intervals

The r^2 and p-value give us some indication of how well our regression model performs. However, we can add further information to our graph:

- The line which represents the regression model

- The confidence intervals that indicate the confidence we have in our regression model.
- The confidence intervals that indicate the confidence we have in the predictions we make based on our regression model

Accessing the confidence interval data It is not easy to find out how to access these parameters in the statslib library. So best to keep this code snippet in your template collection. Bottom line is, we will use the `summary_table()` function provided by the `statsmodels.stats.outliers_influence` module. We can then feed the `results` object of the regression analysis to this function, and it will return all sorts of data (most of which we can ignore). Note that the `alpha` keyword specifies the significance level for the confidence intervals we aim to retrieve. It is customary to provide this number as 1-significance (e.g., α=0.05 implies 1-α = 95%).

```
from __future__ import annotations
from statsmodels.stats.outliers_influence import summary_table

sig: float = 0.05  # = 1 - sig > 0.95 = 95% significance

# variable types for the return values from summary_table
st: SimpleTable  # table with results that can be printed
data: list  # calculated measures and statistics for the table
ss2: list[str]  # column_names for table (Note: rows of table are observations)

st, data, ss2 = summary_table(results, alpha=0.05)

# extract the data for predicted values and confidence intervals
fitted_values: np.ndarray = data[:, 2]  # the regression line

# confidence intervals for the model
model_ci_low :np.ndarray # lower confidence limits
model_ci_upp :np.ndarray # upper confidence limits
model_ci_low, model_ci_upp = data[:, 4:6].T  # don't ask....

# confidence intervals for the model predictions
predict_mean_ci_low :np.ndarray
predict_mean_ci_upp :np.ndarray
predict_mean_ci_low, predict_mean_ci_upp = data[:, 6:8].T
```

Plotting the confidence interval data The upper and lower confidence boundaries describe the upper and lower boundaries of an area. We can either plot these boundaries as a line plot or as a shaded area. Matplotlib provides the `fill_between` method to shade the area between two lines:

```
# ax.fill_between(storks, model_ci_low, model_ci_up, alpha=0.1, color="C1")
```

Note the use of the `alpha` keyword. This has nothing to do with the alpha which is used in statistics. Rather, it describes the transparency of the object you are drawing. If you set it to one (the default), the object will be fully opaque. If you set it to zero, it will be fully transparent (so you won't see it). See the code below for an actual example.

Creating the Stork Figure

Now let's put it all together. Note that when we draw the figure, it matters whether we draw the confidence intervals first or last. Change the order in the code below, to see the difference.

```
from __future__ import annotations
import pandas as pd  # inport pandas as pd
import matplotlib.pyplot as plt
import pathlib as pl
import statsmodels.formula.api as smf
from statsmodels.stats.outliers_influence import summary_table

fn: str = "storks_vs_birth_rate.csv"  # file name
cwd: pl.Path = pl.Path.cwd()
fqfn: pl.Path = pl.Path(f"{cwd}/{fn}")

# test if file is present
if not fqfn.exists():  # check if the file is actually there
    raise FileNotFoundError(f"Cannot find file {fqfn}")

# read data
df: pd.DataFrame = pd.read_csv(fn)  # read data
df.columns = ["Babies", "Storks"]  # replace colum names
storks: pd.Series = df["Storks"]
babies: pd.Series = df["Babies"]


# ------ create linear regression model ------
model: smf.ols = smf.ols(formula="Babies ~ Storks", data=df)
results: model.fit = model.fit()  # fit the model to the data


# ------ extract model parameters
slope: float = results.params[1]  # the slope
y_0: float = results.params[0]  # the y-intercept
r_square: float = results.rsquared  # r_square
p_value: float = results.pvalues[1]  # the p_value
ds: str = (
    f"y = {y_0:1.4f}+x*{slope:1.4f}\n" f"$r^2$ = {r_square:1.2f}\n" f"p = {p_value:1.4f}"
```

```
)


# ------ extract confidence intervals ------
sig: float = 0.05  # = 1 - sig > 0.95 = 95% significance

# variable types for the return values from summary_table
st: SimpleTable  # table with results that can be printed
data: np.ndarray  # calculated measures and statistics for the table
ss2: list[str]  # column_names for table (Note: rows of table are observations)

# variable types for the confidence interval data
model_ci_low: np.ndarray
model_ci_up: np.ndarray
predict_mean_ci_low: np.ndarray
predict_mean_ci_up: np.ndarray

# get data
st, data, ss2 = summary_table(results, alpha=sig)

# extract regression line
fitted_values: np.ndarray = data[:, 2]

# extract confidence intervals for the model
model_ci_low, model_ci_up = data[:, 4:6].T  #

# extract confidence intervals for the predictions
predict_mean_ci_low, predict_mean_ci_up = data[:, 6:8].T

# ------ create plot ------
fig: plt.Figure  # this variable  will hold the canvas object
ax: plt.Axes  # this variable will hold the axis object
fig, ax = plt.subplots()  # create canvas and axis objects

# plot confidence intervals first
ax.fill_between(storks, predict_mean_ci_low, predict_mean_ci_up, alpha=0.1, color="C1")
ax.fill_between(storks, model_ci_low, model_ci_up, alpha=0.2, color="C1")

# add data points
ax.scatter(storks, babies, color="C0")

# regression line
ax.plot(storks, fitted_values, color="C1")

# plot options and annotations
plt.style.use("ggplot")
fig.set_size_inches(6, 4)
fig.set_dpi(120)

```

```
ax.text(1000, 1750, ds, verticalalignment="top")
ax.set_xlabel("Stork Pairs")
ax.set_ylabel("Newborn Babies [$10^3$]")
fig.set_tight_layout("tight")
fig.savefig("stork_new.png")
plt.show()
```

The code above is quite lengthy, especially given the fact that you can do a regression analysis with a few clicks in excel. On the other hand, if you re-arrange the code a little bit, and use generic variables, you can create a code template where you only specify a few key parameters at the beginning of the code, and then you can generate a much more meaningful regression analysis with a few keystrokes:

```
""" Description:
    Author:
    Date:
"""
# ----------- third party library imports ------------------
from __future__ import annotations
import pandas as pd  # inport pandas as pd
import matplotlib.pyplot as plt
import pathlib as pl
import statsmodels.formula.api as smf
from statsmodels.stats.outliers_influence import summary_table

# ----------- user serviceable pararameters
data_file: str = ""  # csv file name
figure_name: str = ""  # figure name

independent_variable: str = ""  # must match col name
dependent_variable: str = ""  # must match col name

x_axis_label: str = ""
y_axis_label: str = ""
size_x: number = 6  # size in inches
size_y: number = 4  # size in inches

confidence_level: float = 0.05  # 1 - alpha in %

# ----------- main program ---------------------------------
# --- variable declarations

# --- code starts here
```

References

- Robert Matthews, Storks Devilver Babies (p = 0.008), Teaching Statistics 22:2, p 36-38, 2000, https://doi.org/10.1111/1467-9639.00013

Please see the corresponding Jupyter Notebook in your Syzygy Jupyter account. See the beginning of this chapter to get the path to the assignment notebook.

- if you have a UofT account use this link:

  https://utoronto.syzygy.ca/jupyter/hub/user-redirect/git-pull?repo=https://github.com/uliw/PNTA-Notebooks&urlpath=tree/PNTA-Notebooks/&branch=main

6.5 Numpy

These modules are available as as Jupyter Notebooks in:

- `PNTA-Notebooks/working_with_libraries/Numpy/Numpy.ipynb`
- `PNTA-Notebooks/working_with_libraries/Numpy/Numpy_assignment.ipynb`

The numpy (numerical python) library provides data types (and methods) to manipulate numerical data. If you have used matlab before, you will be right at home, and if not, let's explore numpy right here. Unlike lists, which can contain a sequence of numbers, and all sorts of other things, numpy datatypes, can only contain numbers. Furthermore, the numpy types typically carry a specific meaning that enables us to use them to do linear algebra (cross product, scalar product, matrix inversion, etc., etc.).

In the following, we will use this library for its ability to manipulate lists of numbers. The basic datatype in numpy, is the array. An array has one or more dimensions. Another word for 1-dimensional array is vector, and for a two-dimensional array is matrix. Since numpy arrays can have complex multidimensional structures, type hinting can be tricky. We will restrict ourselves to the trivial case to simply indicate that a variable is a numpy array.

```python
from __future__ import annotations
import numpy as np

ml: list[int] = range(0, 10, 1)  # create a list
v: np.ndarray = np.array(ml)  # create array from list
print(v)
print(v[1])
print(type(v))
```

In the above example, we first create a list, then we use `np.array()` function to convert the list into a 1-dimensional array (vector). Vector elements can be accessed by index similar to list elements, and lastly, we check that `v` is indeed a numpy object.

Numpy provides it's own functions to efficiently create arrays:

```
# get a list of zeros
print(np.zeros(5))  # vector of 5 zeros
print(np.arange(0, 10, 1))  # vector from 0 to 10 with step 1
print(np.linspace(0, 9, 10))  # vector from 0 to 9 with 10 elements
```

Arrays only contain numbers, and similar to the pandas dataseries object, you can do a lot of operations without the need for a loop. Unlike pandas, numpy arrays understand linear algebra, so you can do cross products, matrix inversion, etc., etc.. But back to our main focus, manipulating lists of numbers. Let's say you want to create a list of 10 X-Y coordinates, where X runs from 0 to 1, with 11 entries, and Y runs from 100 to 200 with 11 entries. So our desired array would then have two columns and 10 rows. There are a couple of ways to achieve this, but here we will use the `np.reshape()` function because this function will be central to our assignment.

```
X: np.ndarray = np.linspace(0, 1, 5)  # start, stop, number of elements
Y: np.ndarray = np.linspace(100, 200, 5)  # start, stop, number of elements
print(X)
print(Y)
```

so for now, we have two vectors, rather than a matrix. Now we combine both vectors

```
XY: np.ndarray = np.append(X, Y)
print(XY)
```

We created a new 1-dimensional array `XY` which contains both `X` and `Y`. But this is still not a list of coordinates - cue `np.reshape()` This commands allows us the change the geometry of an array

```
m: np.ndarray = np.reshape(XY, (5, 2))  # (# of rows, # of cols)
print(m)
```

This gives us the correct geometry (5 rows, with two elements each), but the numbers are all mixed up. If this reminds you of the Rubiks cube, you are on the right track. There is in fact a command to rotate the matrix elements, but before doing so, let's try something else

```
m :np.ndarray = np.reshape(XY, (2, 5))  # (# of rows, # of cols)
print(m)
```

Just flipping the matrix dimensions in the reshape function, gives us a matrix where the X and Y values are paired. Alas, the pairs are not side by side, instead they are on top of each other. If you have worked with excel before, you may have come across a function which allows you to switch rows and columns, a transformation that is called "transpose". Transposing a matrix is in fact such a common operation that numpy provides a shorthand to do so. We just add the `T` method to our matrix (note the absence of brackets after the `T`).

```
print("m befor transposing = \n")
print(m)
print("\n m after transposing = \n")
m = m.T
print(m)
```

Now, or matrix shows nicely paired X and Y values (compare this to where we started with individual vectors). If you look carefully, you will see that each row is rendered as a 1-dimensional array, and that the matrix itself is rendered as an array arrays. We access the individual matrix elements in the usual index syntax. However, since we now have a 2-dimensional array, we need two indices.

```
print("get the second element in the second row:")
print(m[1, 1])  # row 0, col 1

print("\n get all elements in the second row:")
print(m[1, :])

print("\n get all elements in the second column:")
print(m[:, 1])
```

Note that the last two print statements return 1-dimensional arrays that are printed sideways. Unlike matlab, numpy does not differentiate between row and column vectors. It is simply a 1-d array, and those are always printed sideways.

There is a lot more to numpy, but the above will be enough to solve our next assignment.

6.6 References

Index