
Python Programming Fundamentals

Eli Edrian Tan, Rubelito Abella

Jun 19, 2024

CONTENTS

To kickstart the training program, we'll start off with one of the most important tools in data science: python.

Python is used for diverse tasks in the data science pipeline. From data cleaning, data exploration, statistical analysis, visualization, model building, presentation, etc.

In this section of the training you will learn the basics of python programming.

Python is easy

Python is designed to be close to natural language. It was created to promote readability, maintainability and communication. Instead of dedicating the time to learn difficult syntax, you can easily jump into writing python scripts.

Python is well supported

Python is not only well-loved in data science, it is well-loved across many fields. Because of this python enjoys the support of a strong and passionate community that loves to contribute new libraries and tools.

Python tools for Data Science

Throughout this training course you will be learning python and the tools associated to python for data science.

pandas

`pandas` is used for all data cleaning, data exploration, data analysis. With pandas you're able to structure data in an intuitive way, you're able to manipulate data easily, and seamlessly

numpy

`numpy` is the heart of almost every involving vectorized operations in python. You will use numpy when you're interacting with numerical data,

matplotlib

`matplotlib` is used for creating rich customizable visualizations. You will use matplotlib to explore data, evaluate models, and present results

scikit-learn

`scikit-learn` offers a wide library of machine learning models. It allows you to tune, cross validate, and evaluate models

Table of Contents

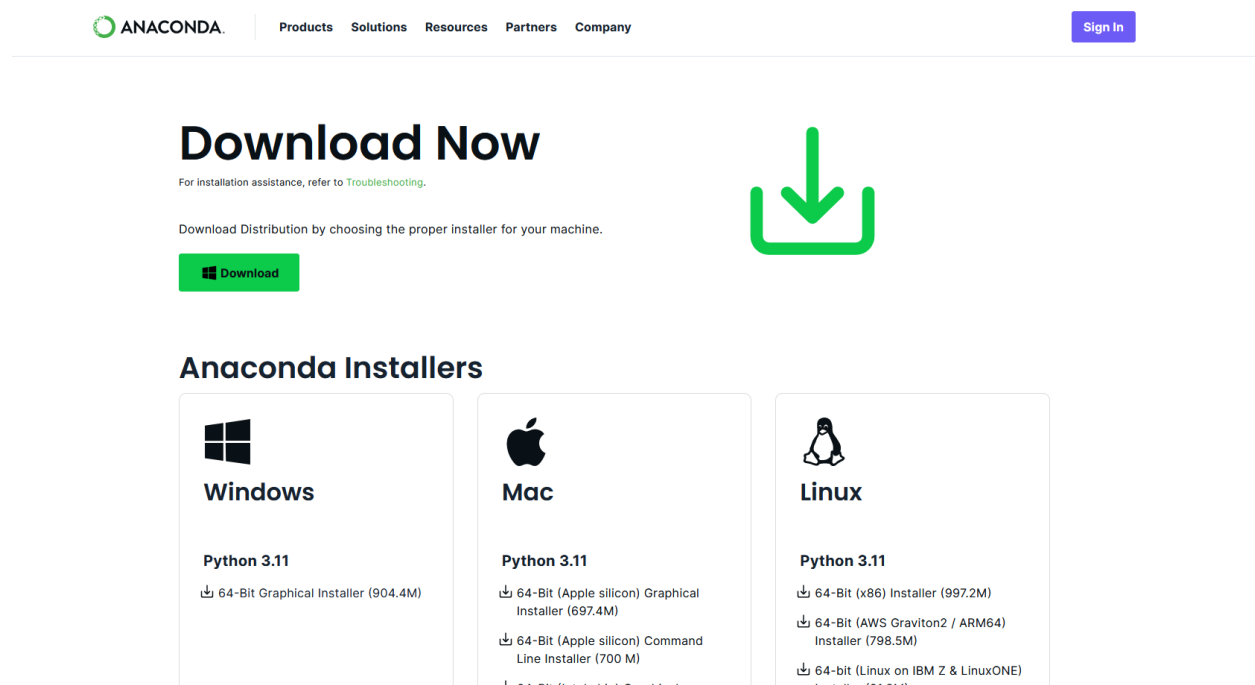
- *Installing Python*
- *Jupyter Notebooks*
- *Expressions*
- *Datatypes*
- *First program*
- *Conditionals*
- *Loops*

INSTALLING PYTHON

1.1 Anaconda python

Installing anaconda python comes with with useful resources. The anaconda install comes with python 3, jupyter, conda environment and more: You can download anaconda through the [anaconda download page](#).

Choose the correct installer for your operating system

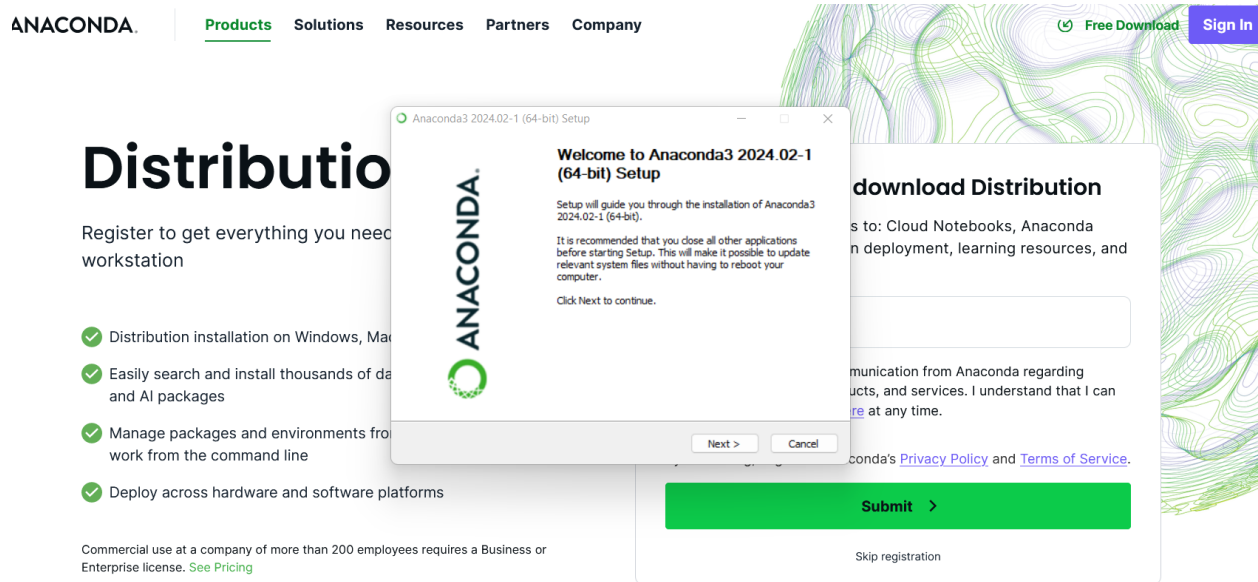


The screenshot shows the Anaconda website's download page. At the top, there's a navigation bar with the Anaconda logo, links for Products, Solutions, Resources, Partners, and Company, and a Sign In button. The main heading is "Download Now" with a subtext "For installation assistance, refer to [Troubleshooting](#)." Below this is a green button labeled "Download" and a large green download icon. The section "Anaconda Installers" features three columns for Windows, Mac, and Linux. Each column lists the Python version (3.11) and provides links to various installer types and their sizes.

| Operating System | Python Version | Installer Type | Size |
|------------------|----------------|---|--------|
| Windows | Python 3.11 | 64-Bit Graphical Installer | 904.4M |
| | | 64-Bit (Intel x64) Graphical | |
| Mac | Python 3.11 | 64-Bit (Apple silicon) Graphical Installer | 697.4M |
| | | 64-Bit (Apple silicon) Command Line Installer | 700 M |
| | | 64-Bit (Intel x64) Graphical | |
| Linux | Python 3.11 | 64-Bit (x86) Installer | 997.2M |
| | | 64-Bit (AWS Graviton2 / ARM64) Installer | 798.5M |
| | | 64-bit (Linux on IBM Z & LinuxONE) Installer | 101.9M |

Install the anaconda using the installer. To avoid issues try to choose an installation path that does not contain any spaces. The installer will warn you about this.

You can just follow the default installation choices from here.

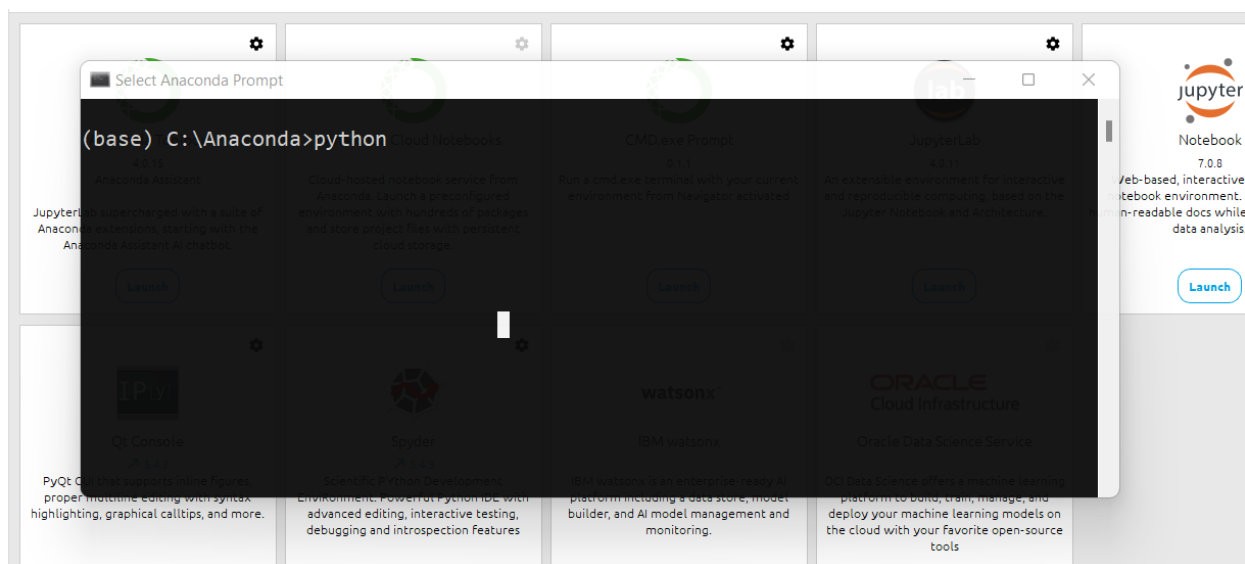


If successfully installed, the installer will install the following tools to your computer:

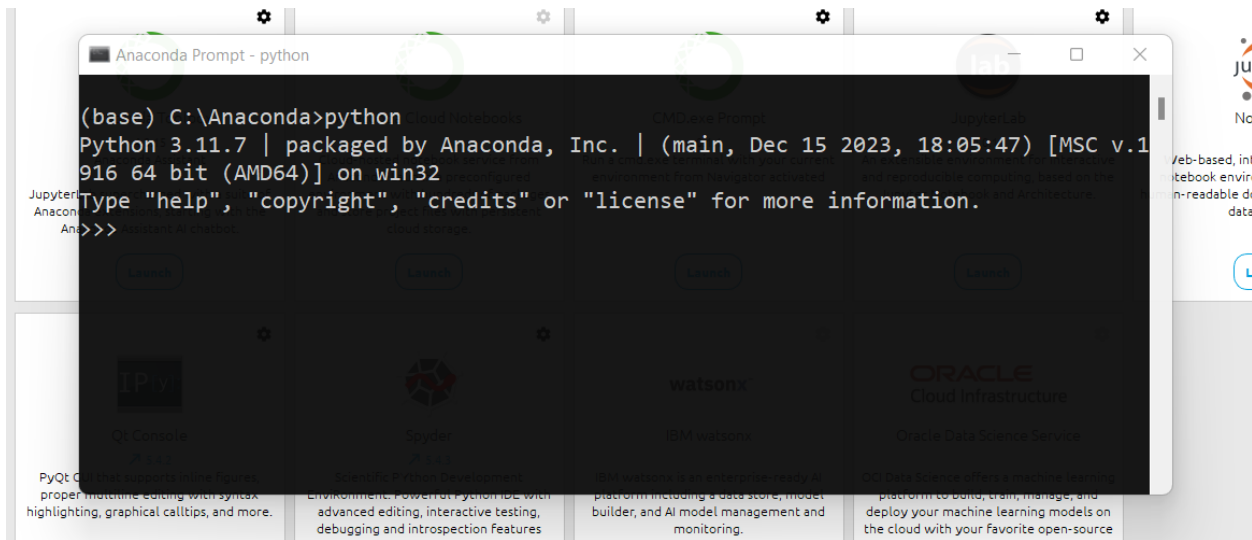
- Python
- Anaconda Prompt
- Anaconda Navigator
- Jupyter Notebook
- Jupyter Lab
- Spyder IDE

To verify your python installation you can run anaconda prompt. It will open the anaconda terminal. From the terminal you can use anaconda's command line tools.

Type the command `python` and press enter:



If everything is good, anaconda prompt should send you to the python interpreter



You can now use the anaconda terminal to run python scripts

1.2 Adding python to your PATH variable

If you want to use python in your own terminal, you can add the python executable in your PATH variable. The way to do this will depend on your operating system. Check the guides below for updating the value of PATH:

- [Windows](#)
- [Mac/Linux](#)

If you're successful in adding python to your PATH variable, you should be able to run the `python` command in your terminal of choice

If you already have a python installation in your machine, this is not advised.

For newer version of windows, running the `python` command may redirect you to the Microsoft Store Download Page for python. To change this, find the setting "App execution aliases" and toggle off the option for App Installer-python.exe and App Installer-python3.exe

1.3 Python Standalone

If you do not want to install the anaconda tool stack. You can also opt to install the python by itself. You can follow the instructions on [python's download page](#)

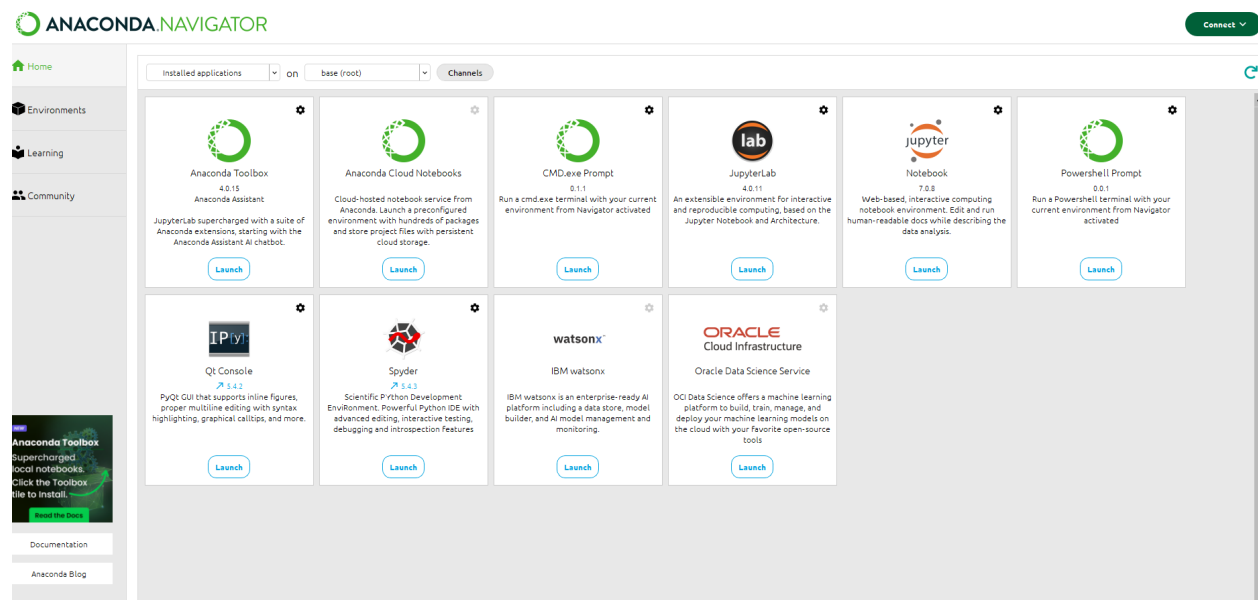
If you do choose to install python by itself you will also need to manually install jupyter, which will be used in this training course. You can select between jupyter lab and jupyter notebook, or even install it as a plugin in VSCode. You can find installation instructions in the [jupyter page](#)

JUPYTER NOTEBOOKS

Throughout this training course, you will be using jupyter to write python code, perform data analysis, present results, create visualization, and more. Jupyter notebooks offer data scientists a way to build interactive environments that combine code, documentation, and output. In fact the lecture notes that you will be reading are written in jupyter notebooks.

2.1 Launching Jupyter notebook

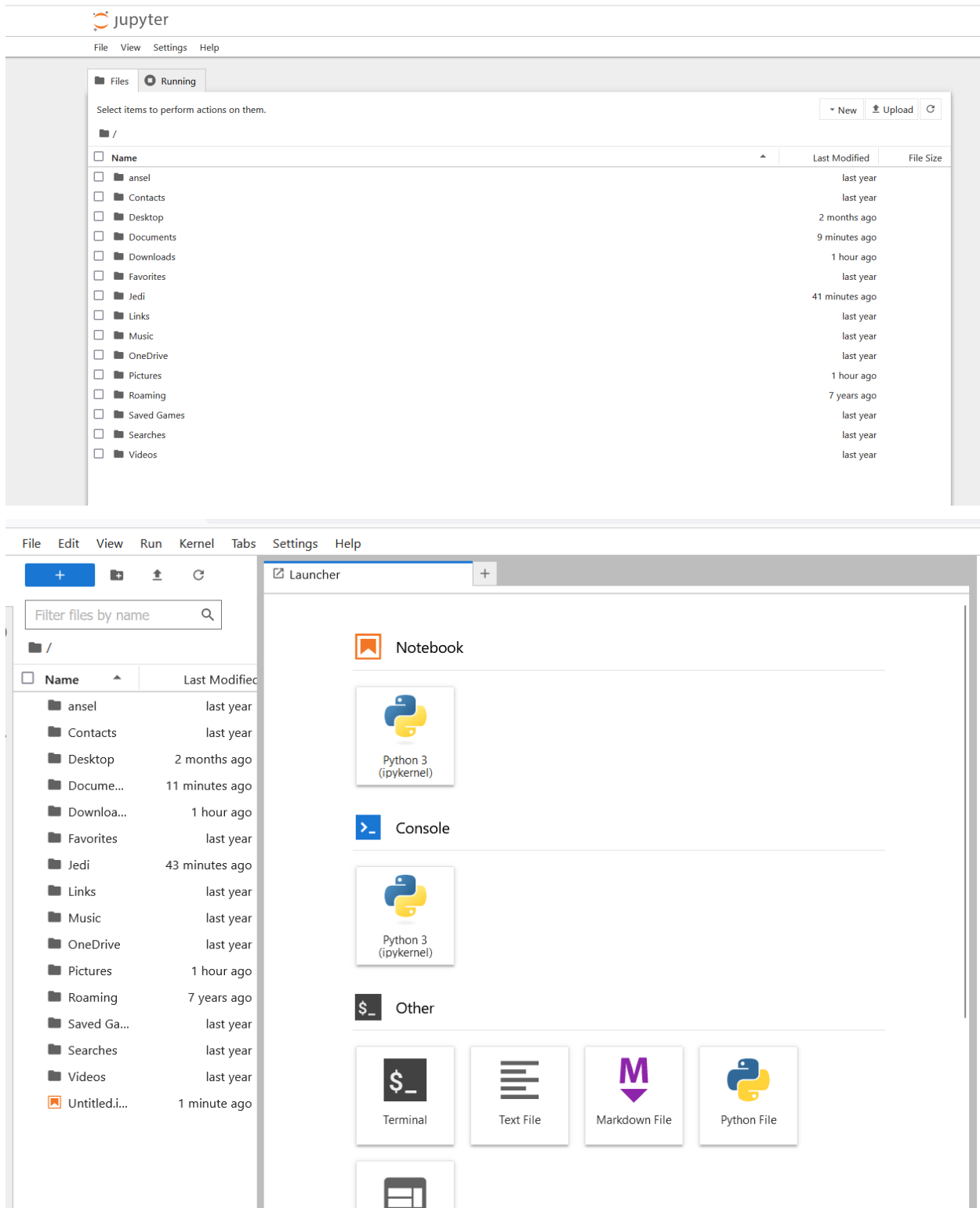
If you installed python through the anaconda distribution, you should be able to launch Jupyter Notebook or Jupyter Lab in Anaconda Navigator. You can also launch them through Anaconda Prompt. Just run the commands `jupyter lab` or `jupyter notebook`. You can select either jupyter lab or jupyter notebook.



It's up to you on which environment you prefer. Both of them will run through your browser.

2.2 Creating notebooks

Once you've launched jupyter through either lab/notebook, you should be able to see a file explorer page/tab.

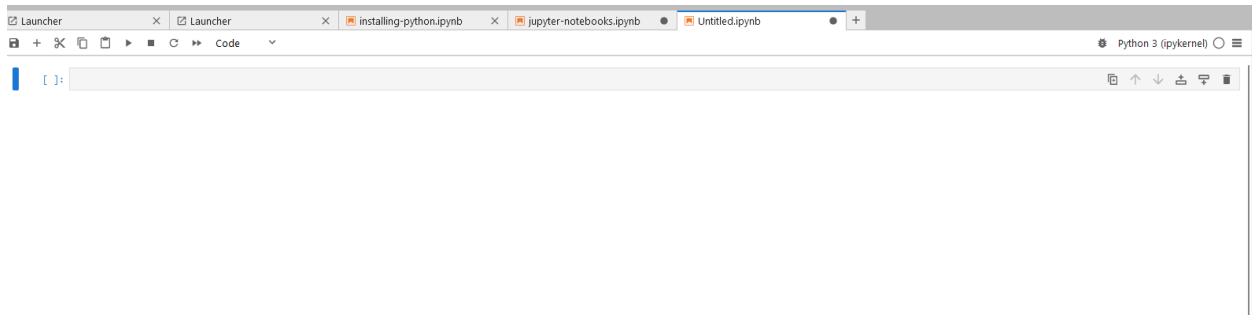


The file explorer shows the directory where you installed anaconda or where you ran the command. From here you can

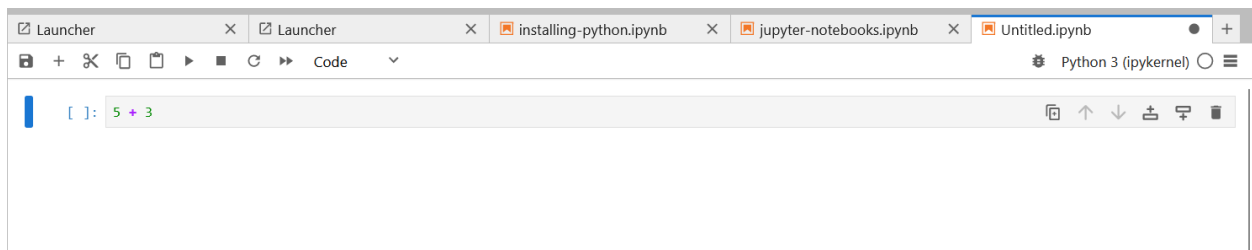
navigate to the folder of your choice and create a new notebook. When creating a notebook, jupyter might ask you to select a kernel. Choose Python3 (ipykernel).

2.3 Interacting with python through Jupyter Notebooks

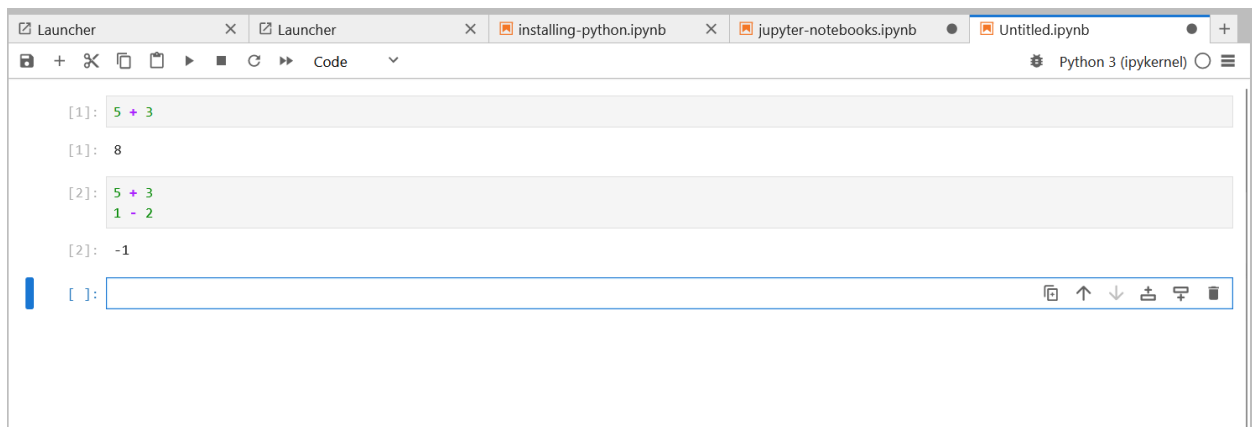
Jupyter notebooks are composed of cells. Cells can either contain code, markdown, or raw text. With a new notebook you start with one code cell



Click on the cell to edit it. Since this is a code cell, it is expecting python code. We can write a python expression here

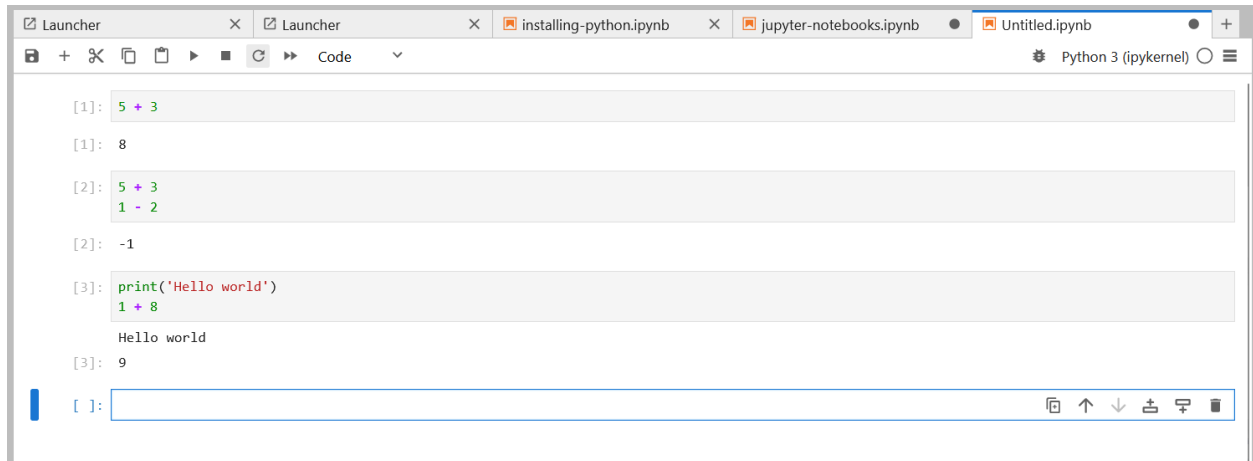


While the cell is selected, you can press Shift+Enter to run the cell (note how this also automatically adds an empty code cell in the bottom). When a code cell is run the code inside it will be executed. It will also check the last line of the cell and try to print its evaluation.



In the example above we have created a new cell and added two lines of expression. Notice how in the second cell, there is only one output. As said before, the cells output will automatically display the evaluation of the last line of code.

You can also force jupyter to display other things by explicitly calling python's `print()` function



If you want to change the cell type of a cell, you can select the desired type in the dropdown found in the tool bar. Other than code cells, you can choose between raw text cells, and markdown cells. Raw text cells will simply render the text placed inside it, while markdown cells will render the text with markdown formatting. Raw text cells and markdown cells are not executed by jupyter. Use these cells to add narratives or documentation.

For more detailed tutorials, tips, and tricks on how to use Jupyter you check the following external resource:

[How to Use Jupyter Notebook: A Beginner's Tutorial](#)

EXPRESSIONS

Expressions are the building blocks of programming. It is used to represent values. It can also be combined with other expressions to create more expressions.

3.1 Literals

Literals are the simplest forms of expressions. Just like any expression, literals represent some type of value. Literals are values that represent their value in a 'literal' sense. See the examples below. These expressions are literals

```
2
```

```
2
```

```
'this'
```

```
'this'
```

```
True
```

```
True
```

You can see why they represent their value in a *literal* sense. The literal `2` *literally* represents the integer value 2. The literal `this` represents the word `this`. As seen in the examples above, literals (and expressions in general) can come in the form of numbers, words, truth values, and more. We will learn about the different types of values expressions can be in the topic **Datatypes**.

3.2 Combining Expressions

Expression in python and programming are patterned after mathematical expressions. In the same way you can combine mathematical expressions, you can also combine programming expressions based on predefined rules. For example we can combine two integer literal expressions using addition and we will get a compound expression

```
2 + 1
```

```
3
```

The expression $2 + 1$ is a combination of the literals 2 and 1 using addition. The expression is combined using addition through the use of the operator +.

As you can see when you run the cell above, it produces an output of 3. When you run said cell, you tell python to evaluate the expression $2 + 1$. The evaluation of $2 + 1$ is simply the sum 3.

Since $2 + 1$ is also an expression, we can also combine it with other expressions through operations.

```
2 + 1 - 4
```

```
-1
```

When working with multiple operations in one expression, python follows predefined rules of precedence. On operators with the same amount of priority such as + and - in the cell above, python will combine the expressions from left to right. It will evaluate the leftmost expression first. In the case above it evaluates $2 + 1$ into 3 and then evaluates $3 - 4$ into -1.

In other cases some operators will take priority over other operators, regardless if it is to the left or to the right.

```
1 + 3 * 2
```

```
7
```

```
3 * 2 + 1
```

```
7
```

To promote code readability, you as a programmer can remove ambiguity by adding parentheses in your expressions. Expressions inside the **innermost** parentheses are prioritized above everything else

```
(1 + 3) * 2
```

```
8
```

```
4 + 2 * ((1 + 3) * 2)
```

```
20
```

3.3 Variables

Similar to math, another form of expression is the variable. It is called a variable because unlike literals, its value can **vary** depending on context.

To create a variable, you must assign it with a value.

```
x = 3
```


The cell above is known as an **assignment statement**. It is used to assign value to a variable. The variable in this case is named `x`. There are also rules on the allowed names for variables. And beyond those rules there are conventions on proper name choices.

Note how the cell above doesn't have an output. This is because assignment statements are not expressions that can be evaluated.

3.3.1 Rules for naming variables

- Variable names must start with a letter or underscore
- Variable names must be a combination of letters, numbers, and underscores
- Variable names cannot be reserved words (words that are reserved for python syntax)

3.3.2 Conventions for naming variables

- Variables must be descriptive - it should describe whatever it is representing
- Variables must be concise - it shouldn't be so long that it makes code unreadable
- And more...

If the rules are not followed, your variable name will produce an error in python. On the other hand conventions are not enforced by python. Not following conventions will still produce code that runs but it may result in code that is unreadable or hard to maintain.

The following are examples of well-named variables:

```
size = 10
airport_code = 'MNL'
nCols = 12
```

Variables are also expressions, so they can be used in the same way any other expressions are used. It can be combined through operations, or it can be evaluated as is.

```
size
```

```
10
```

```
size * 2
```

```
20
```

Since we assigned the value 10 to `size` in the previous cells, We can use it to represent the literal expression 10.

3.3.3 Assignment Statements

True to its name, a variable's value can change. To do this, we simply run another assignment statement

```
size = 15
size
```

15

There are two parts to an assignment statement, the left-hand side (to the left of `=`) represents the variable name, and the right-hand side (to the right of `=`) represents the assigned expression. The left-hand side cannot be anything else other than a variable name. On the other hand you can put any expression on the right-hand side.

```
size = 15 + 3
size
```

18

```
size1 = 10
size2 = 20
size3 = size1 + size2
size3
```

30

```
size = size + 3
size
```

21

One source of confusion when it comes to assignment statements is the misconception that it is the same as the mathematical equality. It's easy to make this misconception because after all, they use the same symbol (`=`). When you see an assignment statement, instead of an equal symbol, imagine an arrow pointing to the left.

```
size <- size + 3
```

Think of pushing the expression on the right-hand side into the variable in the left-hand side. The expression on the right-hand side will be evaluated, and whatever value it evaluates into, it will be pushed into the variable on the left-hand side, overwriting whatever value was originally inside. In the example `size = size + 3`, the right-hand side, evaluates into `30 + 3` or `33`. The value `33` is then pushed into `size`. As a result the next time you evaluate `size` by itself, it evaluates to `33`.

The cell below shows what could happen if you repeatedly overwrite `size` by adding 1 to it

```
size = size + 1
size = size + 1
size = size + 1
size
```

24

There is a special name for the assignment statement `size = size + 3` or `size = size + 1`, these are called increments/decrements. It updates the value of an **existing** variable by a certain value. There is a shortcut for increment/decrement statements, which uses the `+=` or `-=` operator.

```
size = 10
size += 1
size
```

11

```
size2 = 10
size2 -= 5
size2 -= 2
size2
```

3

DATATYPES

Python supports plenty of literals and data types, most of these you can discover on your own.

For purposes of this module, we'll only look at a few: `int`, `float`, `str`, `bool`, `list`, and `dict`. However, we might come across the other data types along the way.

4.1 Numerical types

These types represent numbers and will play along well with the arithmetic operators.

4.1.1 `int`

The `int` datatype stores whole numbers and represents Mathematical integers. Just like Mathematical integers, the `int` datatype is unbounded. That is, we can represent as big or as small numbers as we want.

Python also offers `int()` to coerce values to be represented as an `int`:

```
int(2.5)
```

```
2
```

```
int('12345')
```

```
12345
```

4.1.2 `float`

Represents floating point numbers. These are numbers that have a fractional part.

We also have `float()` to coerce values to be a `float`:

```
float(2)
```

```
2.0
```

```
float()
```

```
0.0
```

```
float("123.456")
```

```
123.456
```

4.1.3 Mixing types

Mixing `int` and `float` together in an expression will typically result in the `int` being converted to a `float` first.

```
2.0 + 3
```

```
5.0
```

```
5 * 10.0
```

```
50.0
```

The `int`, 3, is implicitly converted to a `float` and then added with the other `float`. Thus, we get a 5.0 as we're adding two floats.

4.2 Text sequence type

Python represents words and characters as strings. Strings are immutable sequences of characters.

Python accepts either single or double quotes to represent strings:

```
"Hello"
```

```
'Hello'
```

```
'Hello'
```

```
'Hello'
```

```
'Sam says, "jump!"'
```

```
'Sam says, "jump!"'
```

Even triple quotes are accepted:

```
"""Triple  
quotes  
allow
```

(continues on next page)

(continued from previous page)

```
multi
line"""
```

```
'Triple\nquotes\nallow\nmulti\nline'
```

`\n` is a special escape sequence that represents a linebreak (see more escape sequences: https://docs.python.org/3/reference/lexical_analysis.html#escape-sequences). If we print it out with `print`, the linebreaks are displayed as actual linebreaks:

```
print("""Triple
quotes
allow
multi
line""")
```

```
Triple
quotes
allow
multi
line
```

`str()` coerces values to be a string.

```
str(-5.0)
```

```
'-5.0'
```

```
str(1+3)
```

```
'4'
```

```
str(-5.0)
```

```
'-5.0'
```

```
str(1 + 3)
```

```
'4'
```

```
a = 23 / 7
str(a)
```

```
'3.2857142857142856'
```

4.2.1 Concatenation

String expression can be combined to form a new longer string expression. This is done through the operation concatenation (+).

```
"Hello" + " " + "world"
```

```
'Hello world'
```

4.2.2 Replication

Python strings support the * operator though. This multiplies strings to repeat them:

```
"Hello"*3
```

```
'HelloHelloHello'
```

```
"5" + ",000" * 3
```

```
'5,000,000,000'
```

```
"A" * 10
```

```
'AAAAAAAAAA'
```

4.3 Boolean type

This is a value to represent true and false values.

In Python, we use the keywords True and False:

```
True
```

```
True
```

```
False
```

```
False
```


4.3.1 Boolean operators

We can combine boolean expressions using the operators `and`, `or`, `not`.

`and`:

```
True and True
```

```
True
```

```
True and False
```

```
False
```

`or`:

```
True or True
```

```
True
```

```
True or False
```

```
True
```

`not`:

```
not True
```

```
False
```

```
not False
```

```
True
```

Compound Boolean

And just like any expression we can combine boolean expression into compound expressions

```
(True and True) and (True and False)
```

```
False
```

```
x = True  
y = False  
x or (not y)
```

```
True
```

Truth table

Refer to the truth table below to see how operators behave

| and | left operand | right operand |
|--------------|--------------|---------------|
| True | True | False |
| False | False | False |

| or | left operand | right operand |
|--------------|--------------|---------------|
| True | True | True |
| False | True | False |

4.4 Sequence type

A sequence type is used to store multiple values a collective structure. There are multiple sequence datatypes in python: Lists, tuples, and dicts

4.4.1 Lists

For Python, the basic type for sequences is the `list`. List literals are created with square brackets, `[]`.

```
[1, 2, 3, True, 'five']
```

```
[1, 2, 3, True, 'five']
```

```
[5, 0]
```

```
[5, 0]
```

```
[]
```

```
[]
```

The list `[1, 2, 3, True, 'five']` is a list containing the expressions, `1, 2, 3, True, 'five'`. The expressions in a list are known as its **elements**. The list `[5, 0]` is a list containing the elements 5 and 0. The list `[]` is an empty list. An empty list is a list that doesn't have any elements inside of it.

As we can see, the elements inside a list can be expressions of any datatype. And since lists are expressions themselves, we can also put lists inside lists.

```
[[1, 2, 3, [4]], [], [5, 6]]
```

```
[[1, 2, 3, [4]], [], [5, 6]]
```

Note that while lists can contain a mix of different expression types, it is generally not advisable to create a mixed list. Lists and sequences in general are interacted with as a collective. A mixed collective of different types can result in unexpected behavior and type incompatibility.

4.4.2 Inserting items

A list's contents can be changed using append and delete statements. Appending to a list means adding an element at the end.

```
items = ["Apple", "Banana", "Cherry"]
items.append("Durian")
items
```

```
['Apple', 'Banana', 'Cherry', 'Durian']
```

Items can also be inserted in arbitrary positions within a list

```
items.insert(2, "Orange")
items
```

```
['Apple', 'Banana', 'Orange', 'Cherry', 'Durian']
```

Using `items.insert(2, "Orange")`, inserts an item in position 2 of the list

Note how the value of the list variable `items` was modified after using `append` and `insert`

4.4.3 Retrieving items

We can retrieve items from a list using list indexing. An indexing expression retrieves an element of a list at a specified position.

```
items[0]
```

```
'Apple'
```

```
items[2]
```

```
'Orange'
```

```
i = 3
items[i]
```

```
'Cherry'
```

```
items[i-1]
```

```
'Orange'
```

Indexing with `items[0]` evaluates into the element at position 0 of `items`. Note that lists and other python sequences start at position 0. The first item is located at position 0 (also known as index 0) and the last item is located at position `size-1` (also known as index `size-1`)

4.4.4 Updating items

We can also modify a list by reassigning the value of a specific element.

```
items = ["Apple", "Banana", "Cherry"]  
items[2] = "Strawberry"  
items
```

```
['Apple', 'Banana', 'Strawberry']
```

Running the cell above will overwrite the element at position 2/index 2 of the list. In this case, 'Orange' is overwritten into 'Strawberry'

4.4.5 Deleting items

To remove an item from the list, we have `del` and `.remove()`.

To remove an item based on a given index, use `del`:

```
items = ["Apple", "Banana", "Cherry"]  
del items[1]
```

```
items
```

```
['Apple', 'Cherry']
```

In the case above, the element at index 1, is Banana. Running `del items[1]` deletes the element at index 1 which is Banana. When we evaluate `items` again we see that Banana is deleted from `items`

We can also delete items based on value. This is done through the `remove` function. This function removes the first item that matches the the specified value

```
items = ['Apple', 'Cherry', 'Banana', 'Cherry']  
items.remove('Cherry')  
items
```

```
['Apple', 'Banana', 'Cherry']
```

Note how the second instance 'Cherry' is not deleted since `remove` only deletes the first match.

4.4.6 Length

You can check the length/size of a list using `len()`. Using it on a list will evaluate to an integer that represents the number of elements in said list.

```
len([1, 2, 3, 4])
```

```
4
```

```
len([0])
```

```
1
```

```
len(items)
```

```
3
```

4.4.7 Concatenation

You can concatenate 2 list expressions together to form a new list. The process works similarly as string concatenation. It will evaluate into a new list where the elements of the right operand is attached to the end of the left operand

```
[1, 2, 3] + [4, 5]
```

```
[1, 2, 3, 4, 5]
```

```
items1 = ['Apple', 'Banana', 'Cherry']  
items2 = ['Durian']  
items1 + items2
```

```
['Apple', 'Banana', 'Cherry', 'Durian']
```

Unlike `append()`, using concatenation does not change the values of the original list. The concatenation expression evaluates into a new combined list

```
items1
```

```
['Apple', 'Banana', 'Cherry']
```

```
items2
```

```
['Durian']
```

4.5 Tuples

Python tuples are just like the Python list, except that they are **immutable**. That is, they are lists that cannot be modified after creation.

We can create a tuple by separating values with a `,`:

```
1,2
```

```
(1, 2)
```

Python represents tuples as parentheses. You can also use parentheses on your tuples to make it more readable

Reviewing the list operations discussed above, we can use operations on tuples that do not result in some form of modification. Retrieval, length, and concatenation are allowed

```
coords = 3.1,1.2  
coords[1]
```

```
1.2
```

```
len(coords)
```

```
2
```

```
coords + (2.1,)
```

```
(3.1, 1.2, 2.1)
```

The expression `(2.1,)` is evaluated as a tuple with one element. Just writing `2.1` will evaluate into a float
Appending, deletion, removal, and insertion are not allowed

```
coords.append(4.0)
```

```
-----  
AttributeError                                Traceback (most recent call last)  
Cell In[57], line 1  
----> 1 coords.append(4.0)  
  
AttributeError: 'tuple' object has no attribute 'append'
```

```
del coords[1]
```

```
-----  
TypeError                                Traceback (most recent call last)  
Cell In[59], line 1  
----> 1 del coords[1]  
  
TypeError: 'tuple' object doesn't support item deletion
```

```
coords.remove(2.1)
```

```
-----
AttributeError                                Traceback (most recent call last)
Cell In[60], line 1
----> 1 coords.remove(2.1)

AttributeError: 'tuple' object has no attribute 'remove'
```

```
coords.insert(0,0.1)
```

```
-----
AttributeError                                Traceback (most recent call last)
Cell In[61], line 1
----> 1 coords.insert(0,0.1)

AttributeError: 'tuple' object has no attribute 'insert'
```

4.5.1 Strings as tuples

As it turns out, string expressions are similar to tuples of characters. You can interact with a string expression just like you will any tuple expression. Just keep in mind that the elements of a string can only be single characters.

```
word = 'snakes'
word[2]
```

```
'a'
```

```
len(word)
```

```
6
```

4.5.2 Packing and unpacking

We've already seen what's called *tuple packing*.

```
coords = 2.1, 0.3, -5.0
coords
```

```
(2.1, 0.3, -5.0)
```

The above statement packs the 3 values together down into a tuple and then assigns it to `coords`.

The reverse is also possible, called *sequence unpacking*.

Here's an example of a sequence unpack:

```
x, y = (2.5, 13.1)
x
```

```
2.5
```

```
y
```

```
13.1
```

```
a, b, c = coords
a
```

```
2.1
```

The tuple on the right-hand side is unpacked and assigned into the variables `x` and `y`. `x` gets assigned `2.5` and `y` is assigned `13.1`.

Sequence unpacking requires the correct number of left-hand side variables to the number of right-hand side items.

Note that it's called *sequence* unpacking. This means it also works for other sequences, such as lists and even `str`.

4.6 Dictionaries

Dictionaries allow us to map almost arbitrary keys to values. For now, we can think of dictionaries as lists that allow us to “index” by a key. Unlike lists these keys do not have to be integers, it can be any hashable¹ expression.

Hashable values are values that are immutable, such as `int`, `float`, `str`, and other types that at least implement the `__hash__()` method.

Dictionaries are created with `{}`. Each key-value entry of the dictionary is separated by a `:`, with the key on the left and its value to the right.

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

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

In the dictionary above, we use the string expression `"one"` as the key for the value `1`, `"two"` as the key for the value `2`, and so on. Similar to lists, tuples, and strings, we can retrieve values by indexing using the keys

```
data = {"name": "Fish", "quantity": 3, "price": 412.25}
data["name"]
```

```
'Fish'
```

```
key = "price"
data[key]
```

¹ <https://docs.python.org/3/glossary.html#term-hashable>


```
412.25
```

The expression `data["name"]` retrieves the item associated to the key "name", which in this case is "Fish".

Adding an entry to the dictionary is also done by first retrieving an item using indexing and assigning a value to it.

```
data["date"] = "2 Feb 2024"
data
```

```
{'name': 'Fish', 'quantity': 3, 'price': 412.25, 'date': '2 Feb 2024'}
```

Updating is done the exact same way, but this time we retrieve an already existing entry and then reassigning a new value to it:

```
data["price"]
```

```
412.25
```

```
data["price"] = 450.00
data
```

```
{'name': 'Fish', 'quantity': 3, 'price': 450.0, 'date': '2 Feb 2024'}
```

```
data["price"]
```

```
450.0
```

Deleting a key is done with `del`:

```
del data["date"]
data
```

```
{'name': 'Fish', 'quantity': 3, 'price': 450.0}
```

```
data["date"]
```

```
-----
KeyError                                Traceback (most recent call last)
Cell In[77], line 1
----> 1 data["date"]

KeyError: 'date'
```

Now that the entry associated to "date" is now delete. Trying to retrieve it through indexing will cause a `KeyError`

4.7 Type compatibility

When it comes to expressions with operations, we must take into account if the operation is compatible with the datatype of the operands and vice versa. For example, we can expect that the addition operation (+) is compatible with numeric types.

```
1 + 2
```

```
3
```

```
2.0 + 1.1
```

```
3.1
```

```
1 + 2.2
```

```
3.2
```

Some operation-operand combinations are not compatible, an example of this is discussed earlier, strings and (-)

```
'abc' - 'a'
```

```
-----  
TypeError                                Traceback (most recent call last)  
Cell In[81], line 1  
----> 1 'abc' - 'a'  
  
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Trying to evaluate incompatible expressions will cause a `TypeError` as seen above

Sometimes you might expect values to be compatible but they are actually not

```
'123' + 4
```

```
-----  
TypeError                                Traceback (most recent call last)  
Cell In[82], line 1  
----> 1 '123' + 4  
  
TypeError: can only concatenate str (not "int") to str
```

In cases like this if you concatenate the two expression, you must explicitly apply the necessary coercions

```
'123' + str(4)
```

```
'1234'
```

Or if you were intending to add

```
int('123') + 4
```

```
127
```

In some cases, you might expect types to be incompatible but in fact they actually are

```
True + True
```

```
2
```

When using numeric operations, python coerces booleans into their integer representations. `True` is coerced into 1 and `False` is coerced into 0

```
(True * 3) + False
```

```
3
```

FIRST PROGRAM

The interpreter is handy for when we want to have Python calculate a few simple expressions, but it can get hard to use when we want to build programs out of it.

A Python program is simply a file containing Python expressions and instructions, which Python will execute. The program can be as simple as one print statement:

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

```
Hello world!
```

```
subject = "Sam"
verb = "eats"
object = "the apple"

print(subject + " " + verb + " " + object + ".")
```

```
Sam eats the apple.
```

The function `print()` allows us to write string expressions in the output. This is commonly called as *printing*. In the case of jupyter, the string expression will be printed in the output section of the cell. On the case of python programs, the string expression will be printed in the terminal. To demonstrate this let's create our first python program

5.1 Writing your first program

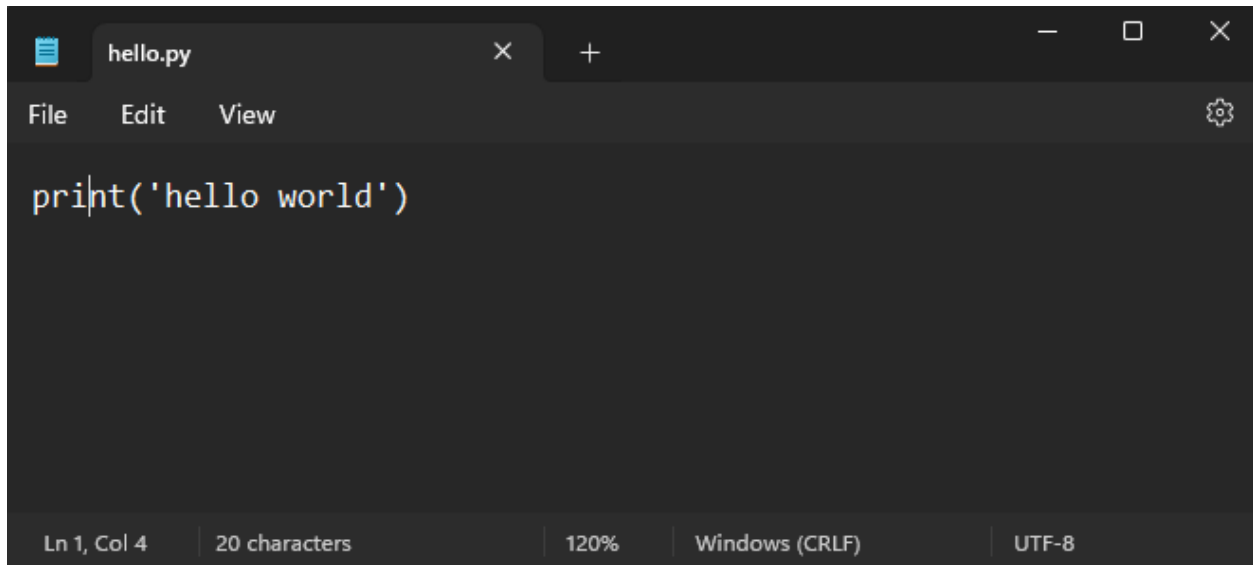
To start writing our Python program, we'll have to open up a new file in a text editor.

Use any text editor (e.g. VS Code, Sublime Text, etc.) or Python IDE (e.g. PyCharm) that you're comfortable with. And while it's completely not necessary for Python to have a full-fledged IDE, it can be quite a help.

Python is very strict with whitespaces and indentation as these are used to delineate sections of our program. In fact, random indentations in the source code is an error—they mean something! As such, if you can, set your text editor to interpret pressing the `<Tab>` key as inserting 4 spaces instead of an actual `<Tab>` character. We don't want to accidentally mix in `<Tab>`s and `<Space>`s in our source file later on.

Now, write our little program.

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



```
hello.py
```

```
File Edit View
```

```
print('hello world')
```

```
Ln 1, Col 4 | 20 characters | 120% | Windows (CRLF) | UTF-8
```

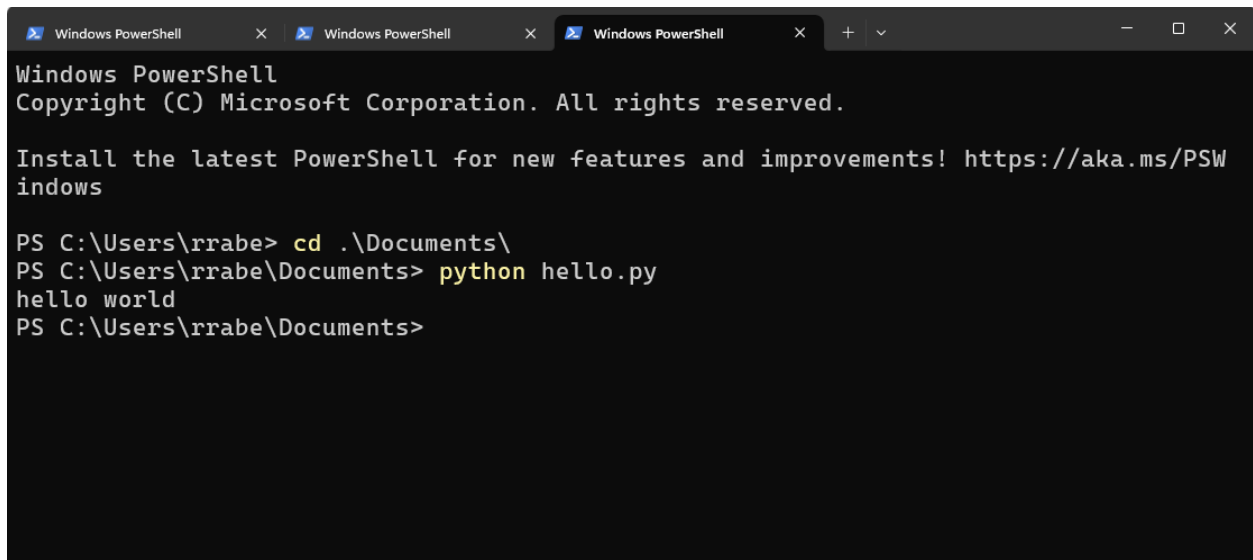
For this to run correctly, make sure you run the command at the same directory as `hello.py`

Save the file with the file extension `.py`. The `.py` indicates that the file is a Python program. In this case, let's save it as `hello.py`.

5.2 Running our program

To run our program, we'll use `python` directly via a terminal (if you installed python through anaconda, you can use the anaconda prompt as your terminal). Make sure you run the command on the same directory the file `hello.py` is in. If python is installed correctly and if there are no errors in your program, you should see the following output:

```
$ python hello.py
Hello world!
```



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSW
indows

PS C:\Users\rrabe> cd .\Documents\
PS C:\Users\rrabe\Documents> python hello.py
hello world
PS C:\Users\rrabe\Documents>
```

Python will execute the file a line at a time. When no more lines are left to execute, the Python program finishes and stops.

5.3 Asking for user input

To make our program more useful, we want it to accept user input. Instead of computing the same numbers over and over again, or having to edit the source code, we can ask for user input with `input()`.

```
print("What's your name?")
name = input()

print("Hello, " + name)
```

```
What's your name?
Sam
Hello, Sam
```

`input()` will always try to evaluate user input into a `str` literal, so we'll have to first coerce it to the appropriate datatype. Without the coercion you will encounter a type error due to type incompatibility. Remember this as it's an easy error to miss.

```
num = input()
print(num + 10)
```

```
Traceback (most recent call last):
  File "/home/user/program.py", line 2, in <module>
    print(num + 10)
    ~~~~^~~~
TypeError: can only concatenate str (not "int") to str
```

In this case we are trying to add two numbers. So we coerce, `num` into an integer first.

```
num = input()
num = int(num)
print(num + 10)
```

```
25
35
```

Here's another variant of this error:

```
print("Enter a number")
num = input()
print(num * 5)
```

```
Enter a number
25
2525252525
```

Python will not crash from this because the `*` operation is type compatible in this context. In this case it will be interpreted as a string replication, repeating `num` 5 times.

5.4 Writing comments

It is a good idea to write comments in your source code. This serves as notes for the programmer (you, or other people) in the future.

We can write comments using the # character.

```
# This is a comment.
```

Comments are ignored by Python when executing and they can even appear after Python expressions.

```
num = 1 + 1 # add 1 and 1 together
```


CONDITIONALS

To enable our program to adapt and be more useful, we have to make use of control flow statements and conditionals.

In a flowchart, conditionals are blocks that allow the execution to split off into another branch depending on certain values. This allows a flowchart to be more useful and be applicable to a variety of scenarios.

In the same way, we want our program to be useful and applicable to a variety of scenarios as well.

In Python, we have a few statements related to conditionals: `if`, `else`, and `elif`.

6.1 `if`

The `if` statement is a conditional that defines that a block of code is to be executed if the condition is true; otherwise, it is skipped.

```
if True:
    print("Hey, it's true!")

if False:
    print("This is false, it won't print.")
```

```
Hey, it's true!
```

Note that the indentation here is important. It determines whether the code falls under the `if` statement or not.

```
if False:
    print("Not printing")
print("I'm outside of the if so I get printed")
```

```
I'm outside of the if so I get printed
```

As with much of programming, we can compound the condition and nest it:

```
if True:
    print("This is printing")
    if False:
        print("But not this")
if False:
    print("None of this either")
    if True:
        print("Nor this")
```

```
This is printing
```

Of course, we wouldn't be checking if `True` is true or `False` is false in a real program. Most likely, we'd have a variable or a condition instead of a literal.

Notice the indentation of the last two lines of code, the last two lines are all inside the `if False` block. To indicate this we add one more level of indentation to every line in the block

```
num = 10
if num > 60:
    print("High")
if num <= 60:
    print("Low")
```

```
Low
```

Depending on the value of `num` provided, we'd get an output of `High` or `Low`.

The expression `num > 60` either evaluates to `True`, which will tell Python to proceed with the clause under the first `if` statement, or `num <= 60` is true, telling Python to print `Low`.

This either-or pattern is common enough that we have a construct for that.

6.2 else

The `if` statement may have an optional `else` statement after it, implementing the either-or pattern.

In an `if-else` statement, only one block of the two is executed.

```
if True:
    print("Only this")
else:
    print("and not this")
```

```
Only this
```

```
num = 90
if num > 60:
    print("High")
else:
    print("Low")
```

```
High
```

Using `else` without an `if` is a syntax error.

Creating `if-else` blocks, allows us to split the program into two branches. If we are looking for more than two we can compound `if-else` statements inside other `if` or `else` statements.

```
num = 24
if num > 60:
    print("High")
```

(continues on next page)

(continued from previous page)

```
else:
    if num <=60 and num > 40:
        print("Medium")
    else:
        print("Low")
```

Low

In the code above, the program can go in either three branches, if the inputted number is greater than 60, it goes to the first branch and prints “High”, if the inputted number is 60 or lesser but greater than 40, it goes to the 2nd branch. If it is not greater than 60, or if it is not between 40 and 60, it goes to the third branch. The third branch in this case is the `else` branch of the `else` block.

Notice the indentation of the last four lines of code, the last four lines are all inside the `else` block. To indicate this we add one more level of indentation to the every line in the block

You can write three branches neatly using the `elif` block. `elif` is short for `else-if` and it acts like a combination of an `else` block and an `if` block. Using the `elif` block we can write a more readable program equivalent to the previous program.

```
num = 45
if num > 60:
    print("High")
elif num <=60 and num > 40:
    print("Medium")
else:
    print("Low")
```

Medium

LOOPS

Imperative programs, such as python programs are built by combining code into three ways: chains, branches/selection, and loops/repetition. Chains are pretty simple. We've seen a few examples of it. It is simply executing lines of code one after the other. The example below is a chain of three lines of code:

```
x = 3
y = 7
print (x+y)
```

```
10
```

Branching/selection is also something we have discussed. We combine statements into branches using `if-else` and `if-elif` conditionals. The example below are two branches of code:

```
password = 'pword'

if len(password) < 7:
    print('password too weak')
else:
    print('password accepted')
```

```
password too weak
```

The third way of combining code is through loops/repetition. The name is self-explanatory. Repetition allows us to repeatedly run lines of code.

See the example below.

```
i = 0
while i < 3:
    print('again')
    i = i + 1
print('finish')
```

```
again
again
again
finish
```

The code above repeatedly runs the last two lines 5 times.

Lets dissect the code line by line to understand how it works.

The first line `i=0`, is simply an assignment statement that **initializes** the value of `i` to 0.

The second line is known as a `while` statement. It is written similar to `if` and `elif` statements. It is the `while` keyword followed by boolean expression (any expression that evaluates into a boolean type value). The boolean expression inside this `while` statements is `i<3`. Like the `if`, `else`, and `elif` statements, we end the `while` statement with a `:` (colon)

The third line is a `print` statement, simply printing the string literal `again`. The fourth line is another assignment statement. This statement reassigns the value of `i` to `i+1` (an increment statement, review assignment statements in the Expressions chapter). Notice how the third and fourth lines are indented. Again, indentation here is important. Changing the indentation will change the meaning of the program.

The fifth line is another `print` statement, this time without indentation. This line prints the string literal `'finish'`.

7.1 How python interprets the code

Lets take a step by step discussion on how python interprets the cell above:

- Starting from line 1, python assigns the variable `i` with 0.
- **On the second line, python checks if the boolean expression inside the while statement is true. In this case “Is `i < 3`”? Yes it is. Since 0 is less than 5, it proceeds to the indented lines.**
 - It executes the third line, printing the string `'again'`.
 - It executes the fourth line, overwriting the value of `i` to `i+1`. Since `i` is zero before this line, the increment, will change the value of `i` to `0+1` or simply 1.
- **Instead of proceeding to the fifth line, python goes back to the the while statement, and checks again, “Is `i < 3`? Since 1 (the current value of `i`) is still less than 3. The check is successful and it proceeds to the indented lines again**
 - It executes the third line, printing the string `'again'`.
 - It executes the fourth line, overwriting the value of `i` to `i+1`. Since `i` is 1 before this line, the increment, will change the value to `i` of `1+1` or simply 2.
- **python goes back to the the while statement, and checks again, “Is `i < 3`? Since 2 (the current value of `i`) is still less than 3. The check is successful and it proceeds to the indented lines again**
 - It executes the third line, printing the string `'again'`.
 - It executes the fourth line, overwriting the value of `i` to `i+1`. Since `i` is 1 before this line, the increment, will change the value to `i` of `2+1` or simply 3.
- **python goes back to the the while statement, and checks again, “Is `i < 3`? Since 3 (the current value of `i`) is not less than 3. The check fails. It skips the indented lines and proceeds to the fifth line**
- It executes the fifth line and prints `'finish'`

As you can see, the indented lines after the `while` statement are repeated *while* the boolean expression remains true.

Let’s look at some more examples:

```
print('countdown...')
i = 5
while i >= 0:
    print(str(i) + '...')
    i = i - 1
print('lift off!')
```

```

countdown...
5...
4...
3...
2...
1...
0...
lift off!

```

As expected, the indented lines right after the `while` statement are repeated until the boolean expression `i >= 0` evaluates to `False`. In this case value of `i` starts as 5 and during every repetition, it is reduced/decremented by 1. Until it becomes `-1` which causes the boolean expression, `i >= 0` to fail (i.e. evaluates to `False`).

In this example we are repeatedly printing the concatenation of `str(i)` (variable `i` coerced as an `int`), and `'...'`. Because of this print statement we are seeing how the value of `i` changes across every repetition.

7.2 Lists and loops

```

fruits = ['Apple', 'Banana', 'Cherry', 'Durian']
i = 0
string_of_fruits = ''
while i < len(fruits):
    string_of_fruits = string_of_fruits + fruits[i] + ', '
    i = i + 1

print('Here are the items: ' + string_of_fruits)

```

```
Here are the items: Apple, Banana, Cherry, Durian,
```

Here is a more complex example. We have a list containing strings called `fruits`. And we have `i` which starts as 0, and `string_of_fruits` which starts as `''` (an empty string, a string with no contents).

Our `while` loop has the boolean expression `i < len(fruits)`. This expression basically checks if the variable `i` is less than the length of the list `fruits`.

The statements that get repeated are lines 5 and 6. Line 5 reassigns the variable `string_of_fruits` with the concatenation `string_of_fruits + fruits[i] + ', '`. This expression concatenates the current value of `string_of_fruits` and `fruits[i]`, which is the element of `fruits` at position `i`.

To see this clearer, let's print the value of `string_of_fruits` at every repetition of the loop. I have added comments to show the print statements I have added.

```

fruits = ['Apple', 'Banana', 'Cherry', 'Durian']
i = 0
string_of_fruits = ''
print(string_of_fruits) #value of string_of_fruits before the while loop begins
while i < len(fruits):
    string_of_fruits = string_of_fruits + fruits[i] + ', '
    print(string_of_fruits) #value of string_of_fruits during every repetition (after_
    ↪reassignment)
    i = i + 1

print('Here are the items: ' + string_of_fruits)

```

```
Apple,  
Apple, Banana,  
Apple, Banana, Cherry,  
Apple, Banana, Cherry, Durian,  
Here are the items: Apple, Banana, Cherry, Durian,
```

As expected, before the loop `string_of_fruits` is an empty string. You can see this as the blank line at the start of the output. On the first repetition `string_of_fruits` gets updated by concatenating it with the `fruits[0]` or the first element of the list `fruit`. On the second repetition, `string_of_fruits` gets updated again by concatenating it with `fruits[1]`, or the second element of the `fruits`. And so on, until the boolean expression `i < len(fruits)` fails and the while loop stops.

7.3 Branches inside loops

```
values = [2,7,3,2,6,6]  
  
i = 0  
while i < len(values):  
    if values[i] > 3:  
        print(values[i])  
    i = i + 1
```

```
7  
6  
6
```

In the example above we can see how we can put branches inside our while loops. Here we have an while loop that starts with `i` as 0. It checks if `i < len(values)`: While this is true it repeats the following: It checks if the element at position `i` is greater than 3. If it is, it prints the element. It then increments `i` by one.

7.4 Loops inside loops

We can also put loops inside loops, these are known as nested loops

```
i = 0  
while i < 4:  
    j = 0  
    while j < 3:  
        print('i=' + str(i) + ', ' + 'j=' + str(j))  
        j = j + 1  
    print() # prints an empty line  
    i = i + 1
```

```
i=0, j=0  
i=0, j=1  
i=0, j=2  
  
i=1, j=0  
i=1, j=1
```

(continues on next page)

(continued from previous page)

```

i=1, j=2

i=2, j=0
i=2, j=1
i=2, j=2

i=3, j=0
i=3, j=1
i=3, j=2

```

The example above shows the values of `i` and `j` during every repetition. Looking at this example closely we can see how the inner loop:

```

... j=0
... j=1
... j=2

```

is repeated 4 times (i.e. the number of repetitions as specified by the outer loop)

Nested loops allow us to create interesting loop patterns. Such as the nested loop below which increases the number of repetitions of the inner loop

```

i = 0
while i < 4:
    j = 0
    while j < i: # instead of looping a set amount of times, it loops based on the
        value of i
        print('i=' + str(i) + ', ' + 'j=' + str(j))
        j = j + 1
    print() # prints an empty line
    i = i + 1

```

```

i=1, j=0

i=2, j=0
i=2, j=1

i=3, j=0
i=3, j=1
i=3, j=2

```

In the example above we simply change the boolean expression inside the inner loop from `j < 3` to `j < i`. With this change, the inner loops number of repetitions become dependent on the value of `i`. In this case `i` is incremented by 1 during every repetition of the outer loop. Because of this, the inner loop's repetition amount is also incremented by 1.

7.5 Infinite loops

One thing you need to be careful of when it comes to loops is accidentally creating an infinite loop. An infinite loop occurs when the loop has no way of ending.

```
while True:
    print('again')
```

If we try to run the code above, we repeat the second line again and again. But since the boolean expression is always True, the loop will never end. If you try to run this code, you'll see that Python will execute the indented line again and again until it cannot anymore.

The example above is very obvious. In some cases you might not notice that you accidentally created an infinite loop

```
int i = 0
while i < 5:
    print('again')
```

The example above will also produce an infinite loop. As you can see you missed the line that reassigns the value of `i`. And since the value of `i` doesn't change, Python will keep evaluating `0 < 5`, which is always true, resulting in a boolean expression that cannot fail, and therefore an infinite loop.

```
int i = 0
while i < 5:
    print('again')
    i = i - 1
```

The example above does change the value of `i` but because you are decrementing `i`. No matter how many times you decrease `i` it will not result in the failure of the boolean expression `i < 5`. Therefore the boolean expression is always true, causing an infinite loop

```
int i = 0
while i < 5:
    print('again')
i = i + 1
```

In the example above we have the correct modification for `i`, which is increment. But unfortunately we did not put the correct indentation on this line. This means that the `i = i + 1` is placed outside of the loop. Which means this line will never be reached until the loop is finished. And since the loop above is missing the correct increment, it will never finish.

7.6 for loops

Python has special repetition statements that are useful for working with sequence datatypes. These are known as `for` loops

```
fruits = ['Apple', 'Banana', 'Cherry', 'Durian']

for f in fruits:
    print(f)
```

```

Apple
Banana
Cherry
Durian

```

A `for` loop goes through the elements of a sequence (lists, tuples, strings, dictionaries, etc.) to allow us to interact with the elements individually.

In the code above, we write `for f in fruits:`. A good way to read this line in english is “*for every f in fruits*”.

On the first repetition of the `for` loop, the first element of `fruits` is assigned to the variable `f`. It then proceeds to the indented lines. In this case we have one line which simply prints the value stored in `f` which is 'Apple'. It then returns to the start of the loop, but this time reassigning `f` to the second element of `fruits`. It then proceeds to the indented lines again and prints the value stored in `f`. This time the stored value is `Banana`.

The `for` loop repeats this process until there are no more elements in `fruits`.

The `for` loop above has the same results as the `while` loop below:

```

fruits = ['Apple', 'Banana', 'Cherry', 'Durian']

i = 0
while i < len(fruits):
    f = fruits[i]
    print(f)
    i = i + 1

```

```

Apple
Banana
Cherry
Durian

```

As said before, `for` loops will work with other python sequence:

```

for element in 1,2,3,4:
    print(element)

```

```

1
2
3
4

```

```

for character in 'word':
    print(character)

```

```

w
o
r
d

```

```

items = {'a':1, 'b':2, 'c':3}
for key in items:
    print(key, items[key])

```

```
a 1  
b 2  
c 3
```

As you can see in the code above, `key` is assigned to the keys of the entries instead of the values