Training Scientists

Python for Beginners using ChatGPT & Claude

Basics

By the end of this section, you will be able to:

1. Understand the concept of **Integrated Development Environments (IDEs)**
2. Know how **AI tools** can help you learn faster but also realize their limitations
3. Differentiate between Python **scripts** and Jupyter **Notebooks**
4. Write and execute a basic **"Hello World"** program
5. Recognize the importance of **code readability and PEP 8** guidelines

Comparison of Python, MATLAB, C++, and R

Aspect	Python	MATLAB	C++	R
Learning Curve	Easy, clean syntax **(Most beginner-friendly)**	Moderate, good for math background	Steep, requires low-level understanding	Moderate, can be challenging for beginners
Performance	Slower than C++, optimizable with NumPy	Good for matrix operations	Typically fastest	Can be slow for large datasets, optimized for stats
Use Cases	General-purpose, web dev, data science, AI/ML **(Most Versatile)**	Engineering, scientific computing, signal processing	System/software dev, games, resource-intensive apps	Statistical computing, data analysis, bioinformatics
Data Analysis & Visualization	Strong libraries (Pandas, Matplotlib)	Excellent built-in capabilities	Limited built-in, needs external libraries	Excellent tools (e.g., ggplot2)
Community & Ecosystem	Large, active, vast library ecosystem **(Largest community)**	Smaller, strong in academia/engineering	Large, extensive domain libraries	Strong in statistics and data science
Cost	**Free**, open-source	Proprietary, licensed (expensive)	Free compilers, some paid IDEs	Free, open-source

Aspect	Python	MATLAB	C++	R
Language Integration	Easy with C/C++ and others **(Excellent interoperability)**	Can integrate with C/C++, Java, Python	Integrates with most languages	Can integrate with C/C++ and Python
ML & AI Support	Excellent (TensorFlow, PyTorch, scikit-learn) **(Leader in AI/ML tools)**	Good support, less extensive than Python	Used for low-level ML and optimization	Good for statistical learning, less for deep learning

I focus mainly on the science and engineering applications of Python but e.g.

- **Netflix** uses Python extensively for its recommendation engine, data analysis, and backend services.
- **YouTube** uses Python for video sharing and viewing functionality
- **Instagram** uses Python (Django framework) for its backend
- **BitTorrent** used Python for the original BitTorrent client

Integrated Development Environments

I recommend either using Jupyter Lab Desktop (available for all operating systems) or Anaconda Cloud which doesn't require installation. Jupyterlab Desktop will run faster, however Anaconda Cloud has Anaconda AI Assistant built in for free. And if you are on a university computer where you can't install anything Anaconda Cloud is a good choice.

If you want to know about the other options for IDEs and why I decided to choose Jupyterlab for this and my other courses check out these two videos:

- 13 Beginner-Friendly Python IDEs Compared in 2024: Jupyter Lab, VS Code, PyCharm, Wing, Zed and More

- Choosing the Best Beginner Friendly Python IDE in 2024: VS Code vs. JupyterLab vs. Anaconda Cloud

Jupyter Lab

For a detailed video about the installation of JupyterLab Desktop check out this video:

- Jupyter Lab Desktop: Installation, Configuration, and Best Practices for Windows & Mac

Line Width and Limiter Lines (PEP8)

In this notebook, you'll notice two limiter lines: one at 80 characters and another at 100 characters. These lines relate to an important aspect of Python coding style.

PEP 8: The Style Guide for Python Code

PEP 8 is the official style guide for Python code. It provides guidelines to improve code readability and consistency across the Python community. One key recommendation concerns line length:

> 🔍 **Guideline**: Keep lines of code between 79-99 characters long.

Why Limit Line Length?

1. **Readability**: Shorter lines are easier to read and understand.
2. **Side-by-Side Viewing**: Allows multiple files to be open side-by-side.
3. **Printing**: Ensures code prints well on standard paper or small screens.

Example

```
# This is a very long line of code that exceeds the recommended
79-character limit and might be hard to read
```

```python
result = some_long_function_name(first_long_parameter_name,
second_long_parameter_name, third_long_parameter_name)

# Better: Split into multiple lines
result = some_long_function_name(
    first_long_parameter_name,
    second_long_parameter_name,
    third_long_parameter_name
)
```

More on Pep8 and good programming principles in the **advanced courses**:

- Python Basics
- Python for Scientists & Engineers
- Python for Biologists

> You can **download** this Jupyter Notebook from the video description (as PDF or as a Jupyter Notebook).

> There are exercises that you can get on my course website https://training-scientists.com (Python Beginner Course using AI)

Anaconda Cloud

For Anaconda Cloud

> https://anaconda.cloud

no installation is necessary, you can just create an Account on their website and start coding.

While Jupyter Notebooks run perfectly, running Python scripts with graphical output does not work.

AI Tools

We will use ChatGPT, Claude and Anaconda Assistant in this course to help you learn programming faster.

AI Tools are great at explaining code and concepts so you can 2X your learning curve.

> To use Claude go to https://claude.ai create an account and start chatting with it.

> To use ChatGPT do the same on: https://chatgpt.com

E.g. Ask Claude 🤖 💬 :

> 1. `Can you tell me how Python compares to Matlab, C++ and R?`

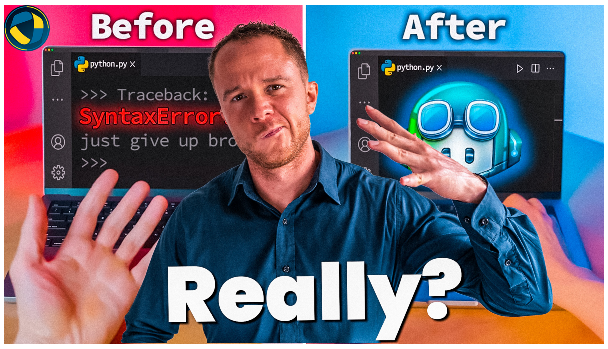
> 2. `Can you reformat that into a visually appealing table that I can copy paste into a Jupyter Notebook markdown cell?`

Using AI tools is not cheating. Cars will look like cheating for someone who sells horse carriages. Or dinosaurs who don't want to learn something new.

The code AI tools generate is not always working so we still need to learn programming ourselves. If you want to know more check out:

- Can Claude 3.5 | ChatGPT 4o | GitHub Copilot build Snake & Electron Cloud simulation in Python? #GPT

- GitHub Copilot: Accelerating Coding or False Hope? | Reaction Video



- Debunking AI Myths: My Reaction to 'Why is everyone LYING?

Python Scripts vs Jupyter Notebooks

Scripts always run completely top to bottom, so if there is an error somewhere in the end you will need to change the code and run everything again.

Whereas Notebooks you can run code cell by cell (line by line if you want to). This makes debugging and overall development a lot faster.

- You can show multiple plots, add text like this and structure the Notebook with a Table of Contents
- Jupyter Notebooks allow you to structure your code with markdown cells, headings etc.
- Scripts are better though if you want to run games (like snake) or simulations with a video like output
- Jupyter Notebooks have a lot of advantages but also some pitfalls like cell state we will look at later

Hello World (Jupyter) 👋

This would not be a programming tutorial without a Hello World script

In [1]:
```
print("Hello World")
```

```
Hello World
```

Hello World (Script)

Variables & Data Types 🏷️

By the end of this section, you will be able to:

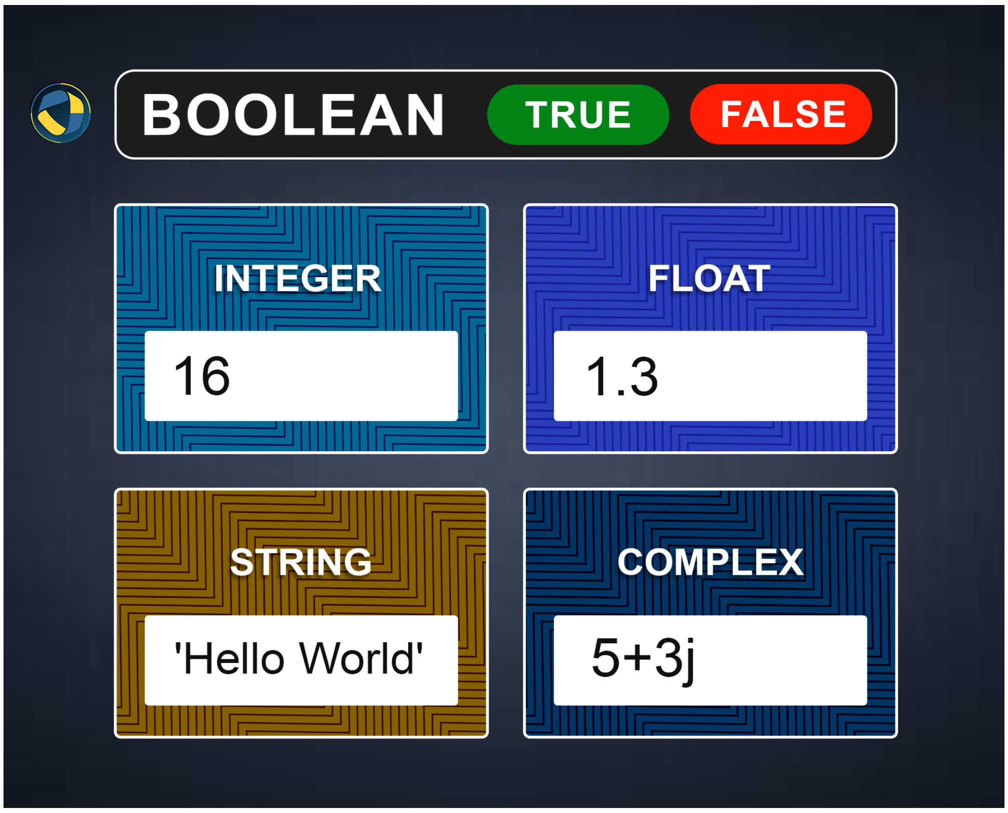
1. Define and use **variables** in Python
2. Identify and work with different **data types** (int, float, string, boolean)
3. Understand and use **f-strings** for string formatting
4. Create and manipulate **lists, tuples, and dictionaries**
5. Recognize the **appropriate use cases** for different data structures

In [2]:
```
z = "Hello World" # string
x = 16 # integer
u = 1.3 # float
complex_number = 5+3j # complex
on_or_off = True # boolean

print(z)
print(x)
print(u)
print(complex_number)
print(on_or_off)
```

```
Hello World
16
1.3
(5+3j)
True
```

In [3]:
```
# Use type() function to give you variable type
print(type(complex_number))
```

```
<class 'complex'>
```



Strings

Let's look at an example to understand what variables are what they are useful for

In [4]:
```
print("Tim is 4 and loves to play.")
print("He builds with blocks every day.")
print("4-year-old Tim stacks them high,")
print("Tim's towers almost touch the sky.")
```

```
Tim is 4 and loves to play.
He builds with blocks every day.
4-year-old Tim stacks them high,
Tim's towers almost touch the sky.
```

What if we want to change the name or the age though? We would have to change it in multiple places manually

In [5]:
```
name = "Max"
age = 4
```

In [6]:
```
# We can use f-strings to insert our variables into the text:
print(f"{name} is {age} and loves to play.")
print(f"He builds with blocks every day.")
print(f"{age}-year-old {name} stacks them high,")
print(f"{name}'s towers almost touch the sky.")
```

```
Max is 4 and loves to play.
He builds with blocks every day.
4-year-old Max stacks them high,
Max's towers almost touch the sky.
```

We do need to execute both cells for the output to update

f-strings are useful e.g. when creating file names and you want to store variable data (like a temperature) to the filename

more on f-strings in the advanced courses

In [7]:
```python
# Single and double quoted strings are the same
text = 'I am a cheetah'
print(text)
```

```
I am a cheetah
```

In [8]:
```python
# with double quoted strings you can still use apostrophes inside of the
print("Tim's towers almost touch the sky.")
```

```
Tim's towers almost touch the sky.
```

Integer variables & Basic Math

In [9]:
```python
a = 5
print(a)
```

```
5
```

In [10]:
```python
# instead of using the print statement, we can just use the variable name
# This works for only one variable per cell though
b = 3
b
```

Out[10]:
```
3
```

In [11]:
```python
c = a + b
c
```

Out[11]:
```
8
```

Cell state (Jupyter Notebook pitfall)

In [12]:
```python
x = 5
y = 2
```

In [13]:
```python
# If the cells are executed top to bottom everything is working
# If you execute this cell without the previous one Python will not know
z = x + y
z
```

Out[13]:
```
7
```

One common mistake I see beginners do is name all their variables x and y which leads to different results depending on the order in which cells are executed.

```
In [14]: x = 10
         y = 15
```

This is still an issue even if you delete the cell because the variables you define are kept in memory.

You need to restart the kernel to clear the memory and rerun the cells to add the variables that you do want to memory again.

Comments

You see me use comments throughout the Notebook to add context, clarify things and for explanations.

Comments are text that Python ignores when running code. They make your code more readable and understandable.

Start with `#`

```
# This is a single-line comment
   x = 5  # Comment at the end of a line
```

Best Practices for Using Comments

- **Be Clear and Concise**: Write comments that are easy to understand and to the point.
- **Update Comments**: Always update comments when you change your code to avoid misleading information.
- **Avoid Obvious Comments**: Don't state the obvious. Focus on explaining 'why' rather than 'what'.

```
# Bad: Increment x by 1
    x += 1

    # Good: Increment age after birthday
    age += 1
```

- Use Comments for Complex Logic: **Explain** tricky, non-obvious, or important parts of your code.

- **Code Sectioning**: Use comments to divide your code into logical sections. (in Jupyter you can use markdown cells and headings for that)

```
# Data Preprocessing
...

# Model Training
...

# Results Analysis
...
```

- **TODO Comments**: Mark areas that need future work.

```
# TODO: Implement error handling for invalid inputs
```

Remember: While comments are important, **clear and self-explanatory code is even better**. Use **descriptive variable and function names** to reduce the need for excessive commenting.

Tuples 🔗

Tuples cannot be changed after creation, so they are constants

In [15]:
```
# Create by using round brackets ()
coordinates = (5, 2)
coordinates
```

Out[15]:
```
(5, 2)
```

In [16]:
```
# This will not work:
#coordinates[1] = 5
```

If a function (section 5) has more than 1 return value it will be returned as a tuple

Lists 📝

In [17]:
```
temp1 = 5
temp2 = 7
temp3 = 10
```

In [18]:
```
# Create a list by using square brackets []
temp = [5, 7, 10]
```

In [19]:
```
temp
```

Out[19]:
```
[5, 7, 10]
```

In [20]:
```
# Appending

temp.append(15) # add 15 to list at the end
temp
```

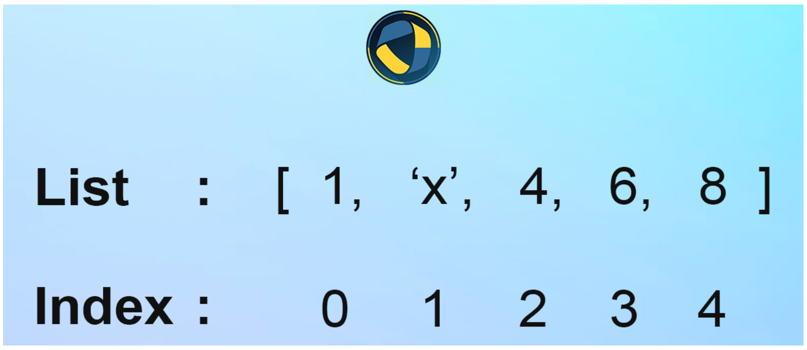
Out[20]:
```
[5, 7, 10, 15]
```

In [21]:
```
# Mixing of datatypes is possible:

testlist_1 = [1,'x',4,6,8]

print(testlist_1)
```

```
[1, 'x', 4, 6, 8]
```

In [22]:
```
# Indexing, Slicing

print(testlist_1[1]) # Use indexing, in python indices start at 0

print(testlist_1[0:2]) # Use slicing, :2 means 'until but not including 2
print(testlist_1[2:]) # 2: means from index 2 until the end
```

```
x
[1, 'x']
[4, 6, 8]
```



In [23]:
```
# Concatenating

testlist_2 = ['boat', 42, 39.9, 'x']

merged_list = testlist_1 + testlist_2
print(merged_list)
```

```
[1, 'x', 4, 6, 8, 'boat', 42, 39.9, 'x']
```

In [24]:
```
# Remove by value
merged_list.remove('x') # remove 'x'
print(merged_list)
```

```
[1, 4, 6, 8, 'boat', 42, 39.9, 'x']
```

In [25]:
```
# Remove by index
merged_list.pop(1) # remove item with index 1
print(merged_list)
```

```
[1, 6, 8, 'boat', 42, 39.9, 'x']
```

In [26]:
```
# Sorting
list_of_strings = ['car','house','boat','cow', 'pig']
list_of_strings.sort() # Can use sort to sort alphabetically or numerical
print(list_of_strings)
```

```
['boat', 'car', 'cow', 'house', 'pig']
```

We will mostly be using numpy arrays and pandas dataframes in the advanced courses

so if you want to know more about lists,

> ask Claude 🤖💬 : `Tell me about python lists and everything I can do with them`

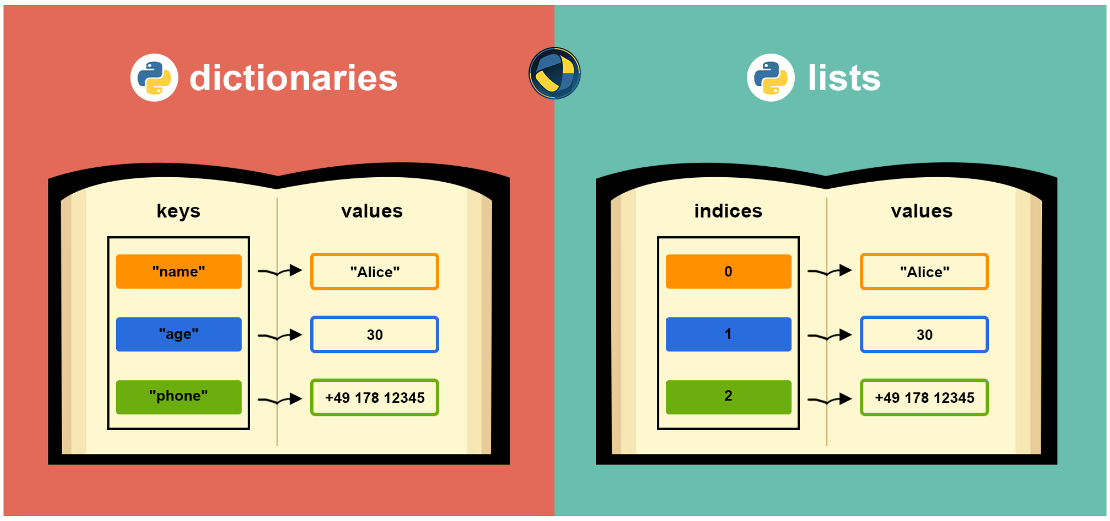
Dictionaries 📖

A dictionary in Python is a collection of key-value pairs. Each key is unique, and it is associated with a value. You can think of a dictionary as a real-world dictionary where you look up a word (the key) and find its definition (the value).

Dictionaries are created using curly braces {} and the key-value pairs are separated by a colon :

In [27]:
```python
# Creating a dictionary to store a person's details
person = {
    "name": "Alice",
    "age": 30,
    "phone": "+49 178 12345"
}

# Accessing values using keys
print(person["name"])
print(person["age"])

# alternatively use the .get() function which defaults to none if the key
print(person.get("phone"))
```

```
Alice
30
+49 178 12345
```



Dictionaries are for example useful when we are plotting (more later)

> Ask Claude 🤖 💬 : `What are the main differences between lists, tuples, and dictionaries in Python?`

Why Use Dictionaries?

Dictionaries are particularly useful when:

- **You need to look up values by a unique key.**
  For example, if you have an employee ID and you want to quickly find the corresponding employee name, a dictionary is ideal.
- **You want to store data with named properties.**
  This is common in cases where you have related information (like user data) and need to access parts of it frequently.

Comparison of Data Structures

Here's how dictionaries differ from other data structures:

- **Lists:**
  - Lists are ordered collections of items that are accessed by their index (a position number starting from 0).
  - Useful when the order of elements is important or you want to store multiple items of the same type.
  - *Example*: `my_list = [1, 2, 3, 4]`
  - **Dictionaries are better** when you need named access to items rather than indexed access.
- **Tuples:**
  - Tuples are similar to lists, but they are immutable (cannot be changed after creation).
  - Useful for fixed collections of items.
  - *Example*: `my_tuple = (1, 2, 3)`
  - **Dictionaries provide more flexibility** as they allow for dynamic modifications (adding/removing key-value pairs).
- **NumPy Arrays:** (later)
  - NumPy arrays are specialized for numerical data and mathematical operations. They offer fast processing for large amounts of numerical data.
  - *Example*: `np.array([1, 2, 3, 4])`
  - **Dictionaries are better** for mixed data types (like strings and numbers) and quick lookups by key.
- **Pandas DataFrames:** (in advanced course)
  - DataFrames are 2-dimensional tabular data structures in the pandas library. They are great for handling and analyzing structured data.
  - *Example*: `pd.DataFrame({"Name": ["Alice", "Bob"], "Age": [30, 25]})`
  - **Dictionaries are simpler and more lightweight** for cases where you just need a quick lookup table or small, unstructured data.

> **Note**: **Lists []** use square brackets, **Tuples ()** use round brackets, **dictionaries {}** use curly brackets.
>
> For numpy arrays it depends whether we convert a list to an array or if we create an array from scratch (more later)

If statements 🔀

By the end of this section, you will be able to:

1. Write basic conditional statements using **if, elif, and else**
2. Use **comparison operators** (==, !=, <, >, <=, >=) in conditional statements
3. Combine conditions using logical operators **(and, or, not)**
4. Understand and apply **boolean logic** in programming contexts

Indentation

In [28]:
```
# Indentation indicates blocks of code

i = 0
if (i==0):
    print('i is 0')
else:
    print('i is not 0')
```

```
i is 0
```

In [29]:
```
# Arbitrary use of indentation creates error

f=1
# This will not work:
    #g=1
```

If statements control the program flow:

If you only want parts of the code executed in case a certain condition is met then use if statements.

In [30]:
```
# The if statement checks for a boolean variable (True/False)
# We can define it beforehand:

is_sunny = True
```

```
if is_sunny:
    print("Enjoy the sunny day!")
else:
    print("Don't forget your umbrella!")
```

```
Enjoy the sunny day!
```

"and" operator

In [31]:
```
# "and" to check for multiple conditions
is_sunny = False
is_warm = False

if is_sunny and is_warm:
    print("Enjoy the warm, sunny day. Take sunglasses.")
else:
    print("It is either not sunny or not warm or neither")
```

```
It is either not sunny or not warm or neither
```

"or" operator

In [32]:
```
# "or" to check for multiple conditions:
is_sunny = False
is_warm = False

if is_sunny or is_warm:
    print("It is either sunny or warm or both")
else:
    print("It is neither sunny nor warm. Just stay home")
```

```
It is neither sunny nor warm. Just stay home
```

Boolean Logic Weather example

Let's explore how boolean variables work using two weather conditions: sunniness and warmth.

Key:

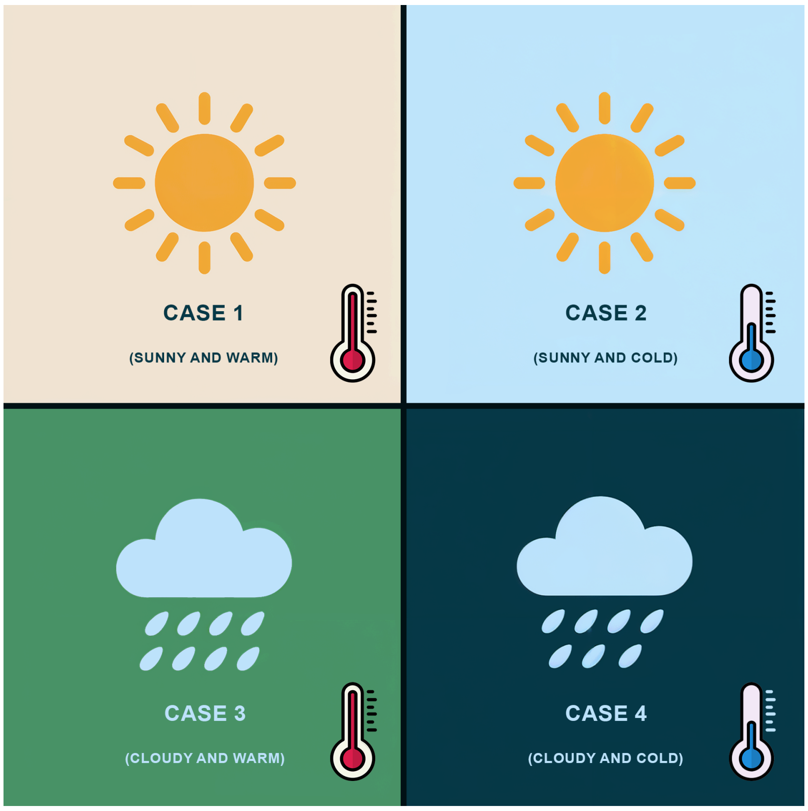
- X means True (Yes)
- O means False (No)

All Possible Combinations:

Weather Condition	Case 1	Case 2	Case 3	Case 4
Is it sunny?	X	X	O	O
Is it warm?	X	O	X	O

Logical Operations:

Operation	Case 1	Case 2	Case 3	Case 4	Explanation
Is it sunny AND warm?	X	O	O	O	True only when both are true
Is it sunny OR warm?	X	X	X	O	True if at least one is true



elif

In [33]:
```python
# Let's catch all four cases:
is_sunny = False
is_warm = True

if is_sunny and is_warm: # case 1
    print("Enjoy the warm, sunny day. Take sunglasses.")
elif is_sunny and not(is_warm): # case 2
    print("It is sunny but it isn't warm. Take a jacket")
elif is_warm: # case 3
    print("It is not sunny but it is warm. Take an umbrella")
else: # case 4
    print("It is neither sunny nor warm. Just stay home")

# for case 3 there is no need to check again whether it is sunny because
# we already checked the 2 cases where it is sunny
```

```
It is not sunny but it is warm. Take an umbrella
```

> Ask Claude 🤖💬: `Explain this code cell to me. How do these if statements work? Why did we not have to check again for is_sunny in case 3?`

Creating booleans by comparison

Comparing two numbers with `>` `<` `==` `!=` `<=` `>=` creates booleans

In [34]:
```
print(1 > 0)
print(1 < 0)
print(1 == 0)
```

```
True
False
False
```

In [35]:
```
# You can create booleans by comparison:

i = 5
if (i==0):
    print('i is 0')
elif(i < 0):
    print('i is smaller than 0')
else:
    print('i greater than 0')
```

```
i greater than 0
```

As a beginner, it's enough to understand the following concepts:

1. **Boolean Variables**: What they are and how they work
2. **Conditional Statements**: The syntax and logic behind `if` ... `elif` ... `else`
3. **Logical Operators**: Combining multiple conditions with:
    - `and`
    - `or`
    - `not(..)`
4. **Comparison Operators**: Used to compare values
    - `==` (equal to)
    - `!=` (not equal to)
    - `<` (less than)
    - `>` (greater than)
    - `<=` (less than or equal to)
    - `>=` (greater than or equal to)

🚀 This foundational knowledge is all you need for the advanced courses:

- Python Basics
- Python for Scientists & Engineers
- Python for Biologists

Functions 🧮

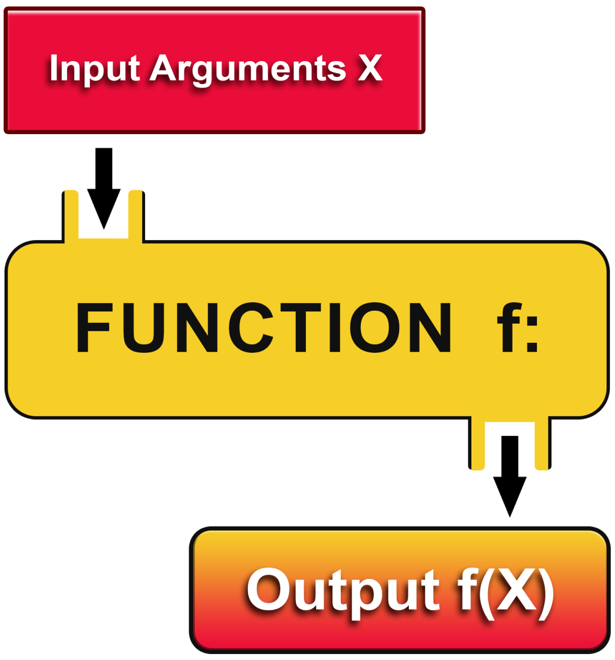
By the end of this section, you will be able to:

1. Define and explain the **purpose of functions** in Python
2. **Create functions** using the `def` keyword
3. Understand the concept of function **parameters and return values**
4. Call functions and use their **return values**
5. Explain the difference between **local and global scope** in functions
6. Recognize and apply **best practices** in function naming and design

In [36]:
```
# To define a function we need to use the "def" specifier, the name of th
# and in parenthesis the input arguments + a colon at the end
# all the code that will be inside the funtion needs to be indented

def hello_world():
    print("Hello World")
```

In [37]:
```
# We need to call the function for something to happen
hello_world()
```

```
Hello World
```



In [38]:
```
def square_function(x):
    return x**2
```

In [39]:
```
square_function(3)
```

Out[39]:
```
9
```

In [40]:
```
# We can save the return value of the function in another variable:

result = square_function(3)
```

In [41]:
```
result
```

```
Out[41]:  9
```

```
In [42]:  # A function can have multiple input values:
          def multiple_input(x,y):
              return x**2 + y**2
```

```
In [43]:  z = multiple_input(2,3)
          z
```

```
Out[43]:  13
```

> "Ask ChatGPT: Can you explain the concept of return values in functions in Python and why they're important?"

Why Use Functions?

Functions are fundamental building blocks in programming that offer several advantages:

1. **Organize Code**: Functions help structure your program into logical, manageable chunks.
2. **Avoid Repetition**: Instead of copy-pasting code, functions allow you to reuse code efficiently.
3. **Enhance Readability**: Well-named functions make your code self-documenting and easier to understand.
4. **Improve Maintainability**: When code is organized into functions, it's easier to update and debug.

> 💡 **Pro Tip**: Whenever you find yourself copy/pasting parts of code, there's usually a better way – and that way often involves functions!

Here's an **example** of how functions can **simplify our code**:

```
In [44]:  temp1 = 30
          result_1 = temp1**2 + temp1 + 5
          print(f"Result for {temp1}: {result_1}")

        Result for 30: 935
```

```
In [45]:  temp2 = 40
          result_2 = temp2**2 + temp2 + 5
          print(f"Result for {temp2}: {result_2}")

        Result for 40: 1645
```

```
In [46]:  def polynomial_function(x):
              return (x**2 + x + 5)
```

```
In [47]:  polynomial_function(temp1)
```

```
Out[47]:  935
```

```
In [48]:  polynomial_function(temp2)
```

```
Out[48]:  1645
```

Multiple return values

```
In [49]:  def min_max_average(numbers):
              minimum = min(numbers)
              maximum = max(numbers)
              average = sum(numbers) / len(numbers)
              return minimum, maximum, average
```

```
In [50]:  numbers = [4, 2, 9, 7, 5, 1, 8]
           min_max_average(numbers)
```

```
Out[50]:  (1, 9, 5.142857142857143)
```

Note, how this returns a tuple: (min, max, average)

```
In [51]:  # We can "unpack" the tuple like this:
           min_val, max_val, avg_val = min_max_average(numbers)

           print(f"Minimum: {min_val}")
           print(f"Maximum: {max_val}")
           print(f"Average: {avg_val:.2f}")
```

```
Minimum: 1
Maximum: 9
Average: 5.14
```

> 💡 **Note**: For **math operations** like this we will be using **numpy (section 8)** and numpy arrays. Numpy has a lot of built in functions that have **multiple return values**

Understanding Scope in Python 🔍

In Python, the scope of a variable determines where it can be accessed in your code. Let's explore two main types of scope:

Global Scope

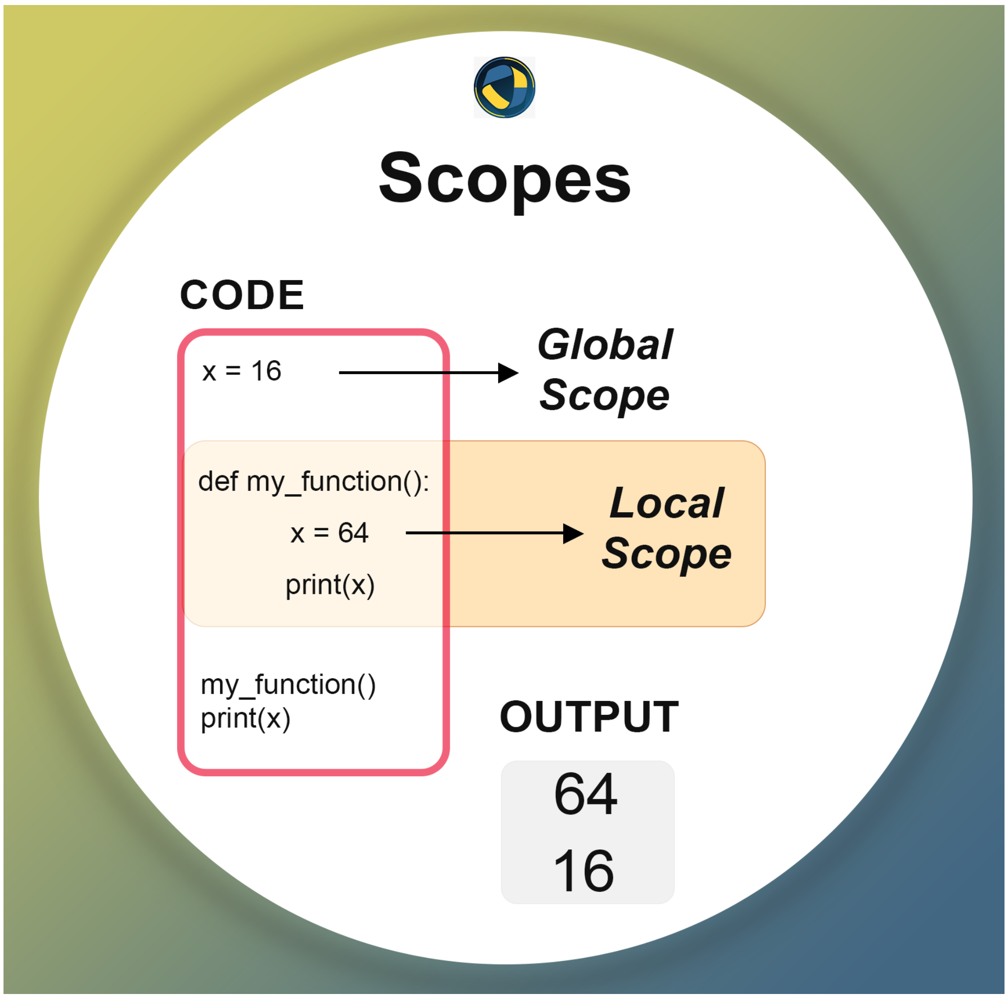
- Variables defined outside of functions
- Accessible throughout the entire code
- Can be read from anywhere, but modifying them requires special handling

Local Scope

- Variables defined inside functions
- Only accessible within that specific function

- Helps prevent naming conflicts and unintended modifications

> 💡 **Best Practice**: Keep the scope of variables as small as possible. This helps prevent errors and makes your code more maintainable.



```
x = 16

def my_function():
    x = 64
    print(x)

my_function()
print(x)
```

```
64
16
```

Why Scope Matters

1. **Prevents Naming Conflicts**: Local variables can have the same name as global variables without interfering with each other.
2. **Improves Code Organization**: Clearly defined scopes make it easier to understand where variables are used and modified.
3. **Enhances Debugging**: Limiting scope makes it easier to track down issues in your code.

In [52]:
```python
# Let's look at an example of global vs. local variables
global_var = "I'm global"

def scope_example():
    local_var = "I'm local"
    print(global_var + " - inside the function")  # Can access global var
    print(local_var + " - inside the function")   # Can access local vari

scope_example()
print(global_var)  # This works
#print(local_var)  # This would raise an error because local_var is not a
```

```
I'm global - inside the function
I'm local - inside the function
I'm global
```

Good Programming Practice ⭐

While global variables are sometimes necessary, it's generally considered good programming practice to avoid accessing them directly within functions. Instead:

- Pass required data as input parameters to your functions
- Return modified values from functions rather than changing global state

This approach, known as "passing parameters," offers several benefits:

1. **Improved Readability**: It's clear what data the function needs to operate.
2. **Better Testability**: Functions that don't rely on global state are easier to test.
3. **Reduced Side Effects**: Functions don't unexpectedly modify global variables.
4. **Enhanced Reusability**: Functions can be used in different contexts without relying on specific global variables.

Example of good practice:

In [53]:
```python
# Instead of this:
global_data = 10

def process_data():
    global global_data
    return global_data * 2

# Prefer this:
def process_data(input_data):
    return input_data * 2

result = process_data(10)
```

Error messages ⚠

```python
def polynomial_function(x):
    return (x**2 + x + 5)
```

In [54]:
```python
test_list = [1, 2, 3]
#polynomial_function(test_list)
```

We will see later how to apply mathematical functions to numpy arrays (so multiple values at once). It does not work with lists

> "Ask Anaconda Cloud 🤖 💬 : Why are we getting an error message here?

Common Errors (How to Handle Them)

When learning Python, encountering errors is part of the process. Here are some common errors you might face and how to address them:

1. **SyntaxError**

    - Cause: Incorrect Python syntax
    - Example: `print "Hello World"` (missing parentheses in Python 3)
    - Fix: Correct the syntax: `print("Hello World")`
2. **IndentationError**

    - Cause: Incorrect indentation of code blocks
    - Example:
      ```
      if True:
      print("This is incorrectly indented")
      ```
    - Fix: Properly indent the code:
      ```
      if True:
          print("This is correctly indented")
      ```
3. **NameError**

    - Cause: Using a variable or function name that hasn't been defined
    - Example: `print(undefined_variable)`
    - Fix: Ensure the variable is defined before use or check for typos in the name
4. **TypeError**

    - Cause: Performing an operation on an inappropriate data type
    - Example: `"2" + 2` (trying to add a string and an integer)
    - Fix: Convert types appropriately: `int("2") + 2` or `"2" + str(2)`
5. **IndexError**

    - Cause: Trying to access a list index that doesn't exist
    - Example: `my_list = [1, 2, 3]` then `print(my_list[3])`
    - Fix: Ensure your index is within the valid range: `print(my_list[2])` (remember, indexing starts at 0)
6. **KeyError**

    - Cause: Trying to access a dictionary key that doesn't exist
    - Example: `my_dict = {"a": 1, "b": 2}` then `print(my_dict["c"])`
    - Fix: Check if the key exists before accessing or use the `.get()` method: `my_dict.get("c", "Key not found")`

When you encounter an error:

1. Read the (end of the) error message carefully - it often points to the line where the error occurred and gives a description of the problem.
2. Check the line number and surrounding code for issues.
3. If you're unsure, try searching the error message online or ask AI tools for help.

Remember, errors are not failures - they're opportunities to learn and improve your code!

Loops 🔄

By the end of this section, you will be able to:

1. Understand the concept of **iteration** in programming
2. Write and use **while loops** for indefinite iteration
3. Implement **for loops** to iterate over sequences (like lists or strings)
4. Avoid and handle infinite loops

while loop

In [55]:
```python
i = 0
while i < 6:
    print(i)
    i = i + 1

print("Have fun")
```

```
0
1
2
3
4
5
Have fun
```

---

careful when defining your criteria, if it is always true the loop will never end

this can be useful in generators though that we cover in the advanced courses

---

In [56]:
```python
# use kernel interrupt should you be stuck
# Keyboard shortcut: Esc i i

#while True:
#    print("This is an infinite loop that will never end")
```

for loop

In [57]:
```python
for letter in "Hello World":
    print(letter)

print("Have fun")

# Note that "letter" is not a keyword in Python, we could give it another
# Jupyter marks Python keywords in green
```

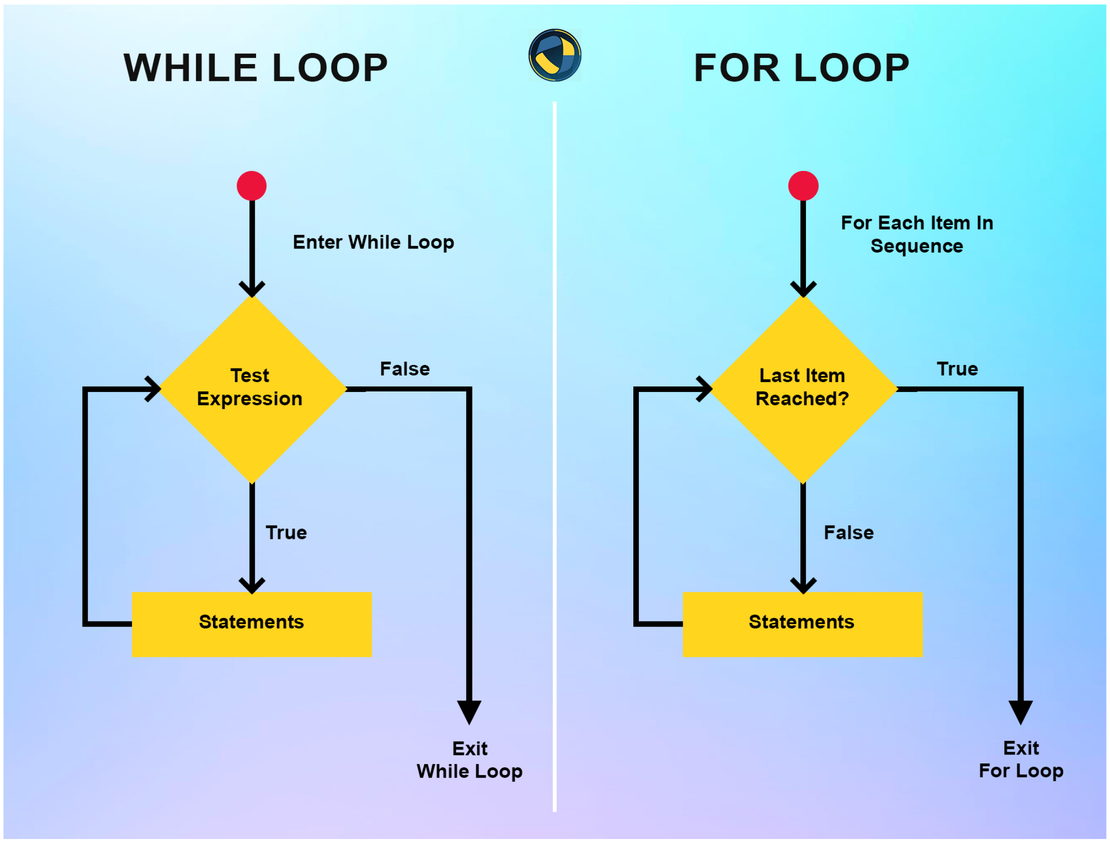
```
H
e
l
l
o

W
o
r
l
d
Have fun
```

In [58]:
```python
binaries = [2, 4, 8, 16, 32, 64]
```

In [59]:
```python
for number in binaries:
    print(f"I am at {number} now")
```

```
I am at 2 now
I am at 4 now
I am at 8 now
I am at 16 now
I am at 32 now
I am at 64 now
```

> Ask Claude 🤖💬: `Explain the difference between a for loop and a while loop in Python. When would you use one over the other?`
>
> Ask ChatGPT 🤖💬: `Can you give me an example of how to use a for loop to iterate over a dictionary in Python?`



Keyboard Shortcuts ⌨️

> **Note**: The shortcuts shown here also work in VS Code by the way.

General Shortcuts

Shortcut	Action
`[ESC]`	Go to command mode
`[Enter]`	Enter edit mode for the selected cell

Command Mode Shortcuts

Shortcut	Action
`i i`	Interrupt Kernel
`a`	Insert cell above
`b`	Insert cell below
`m`	Convert cell to Markdown
`y`	Convert cell to Code
`d d`	Delete selected cells
`Shift Enter`	Execute cell and select below
`Ctrl Enter`	Execute cell and stay
`↑` / `↓`	Select cell above/below
`Shift ↑` / `↓`	Extend selection above/below
`Shift m`	Merge selected cells
`Ctrl Shift -`	Split cell at cursor
`c`	Copy selected cells
`v`	Paste cells below
`x`	Cut selected cells

Edit Mode Shortcuts

Shortcut	Action
`Ctrl /`	Comment/uncomment selected lines (Windows/Linux); On Mac use `Cmd /`
`Ctrl z`	Undo (within a Cell); On Mac use `Cmd z`
`Ctrl Shift z`	Redo (within a Cell); On Mac use `Cmd Shift z`

For more shortcuts, visit: Jupyter Notebook Shortcuts

Virtual Environments 📦

By the end of this section, you will be able to:

1. Explain the **purpose and benefits** of virtual environments in Python
2. Create and activate a virtual environment using **conda**
3. Install and manage **libraries** within a virtual environment
4. Understand the difference between **conda and pip** for package management
5. Create, use and export **environment.yml** files for project reproducibility

Libraries

A library in Python (like NumPy or Matplotlib) is simply a **collection of pre-written code** that someone has created to solve specific problems. You can use this code in your own programs without having to write it from scratch.

The vast number of available libraries is a great strength of Python because they **enhance the functionality and versatility** of Python.

In [60]:
```
import numpy as np
```

In [61]:
```
# Convert a list to a numpy array

test_list = [1, 2, 4, 8, 16]
print(test_list)
```

```
[1, 2, 4, 8, 16]
```

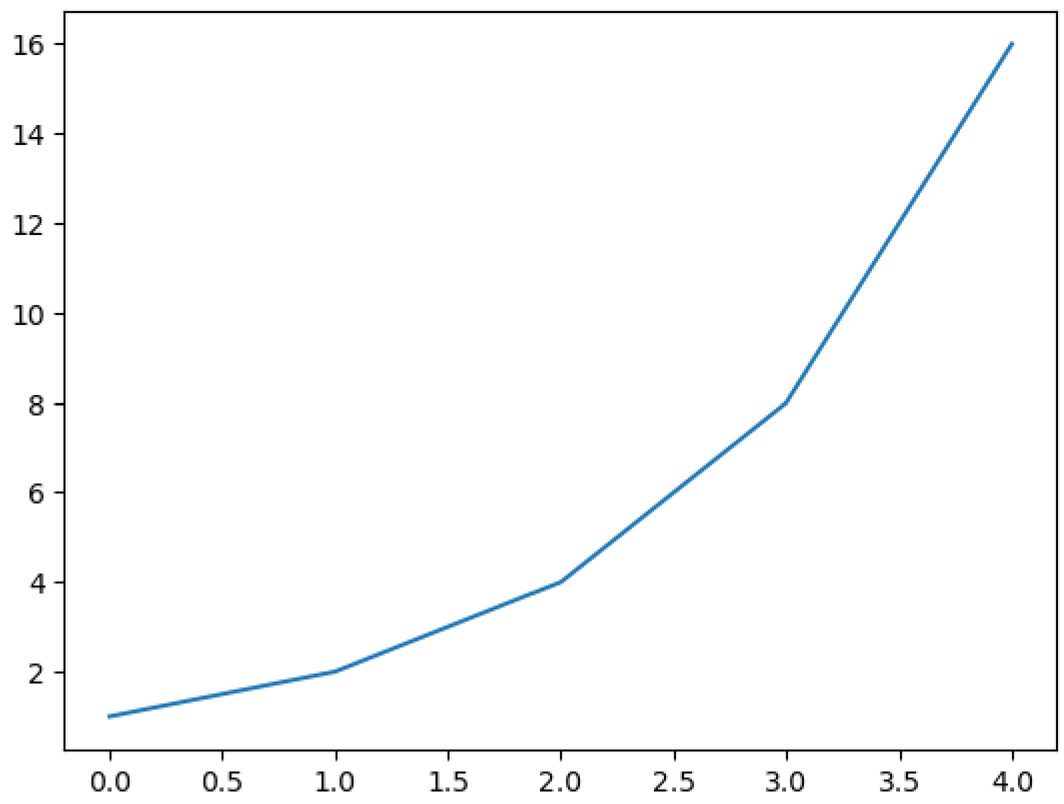
In [62]:
```
test_list_converted = np.asarray(test_list)
print(test_list_converted)
```

```
[ 1  2  4  8 16]
```

In [63]:
```
import matplotlib.pyplot as plt
```

In [64]:
```
plt.plot(test_list_converted)
```

Out[64]:
```
[<matplotlib.lines.Line2D at 0x10fe809b0>]
```



Why use virtual environments?

In a nutshell: Think of environments like sandboxes and the libraries (like Numpy) like children. If you put too many children in one sandbox there will be conflicts. So it is better to have separate sandboxes (environments) to keep it civil.

> Ask Claude 🤖💬 : `Why are virtual environments important in Python development?`

In Python, virtual environments are used to create isolated environments for your projects, allowing you to manage dependencies and packages separately for each project. They are used to solve a common problem in software development: conflicting dependencies and package versions.

In Detail:

- **Isolation**: Virtual environments provide a sandboxed environment for your Python projects. Each virtual environment is self-contained, meaning it has its own directory structure and doesn't interfere with other Python projects or the system-wide Python installation. This isolation helps prevent conflicts between packages and dependencies.

- **Dependency Management**: In a virtual environment, you can install and manage specific versions of Python packages and libraries independently of the global Python environment. This allows you to specify the exact package versions required for your project.

- **Collaboration**: When collaborating on a Python project, you can share the project's virtual environment configuration with others. They can then create the same virtual environment, ensuring that everyone is working with the same packages.
- **Testing and Development**: Virtual environments are crucial for testing and development. They allow you to create an isolated environment where you can experiment with different package versions and configurations.
- If you install everything in your base environment installing new packages can lead to conflicts

Anaconda environments

Anaconda environments are the go to solution when you are on your own computer and you can install anaconda or if you are in Anaconda Cloud.

Very nice cheat sheet about the commands:

https://docs.conda.io/projects/conda/en/4.6.0/_downloads/52a95608c49671267e40c68
cheatsheet.pdf

Advantages:

```
- Anaconda automatically checks dependencies for your
packages and installs the necessary additional libraries
- It is easy to switch environments
```

Disadvantages:

```
- Anaconda is a bit intrusive
- Not available on every machine
```

Benchmarks have shown that since anaconda uses packages that use the Intel MKL (Math kernel library) they can often be faster on Intel CPUs than when using PIP:

https://www.youtube.com/watch?v=AWWaL6pZieo

Pip is nevertheless used a lot. Often when you google how to install a package, you will find the pip install command first.

DO NOT MIX Conda and pip installs in the same environment, as both package managers cannot cross check for compatibiltity with the packages installed by the other one.

You will **not** get an error message and the installation will most likely go through **BUT** you might get **unexpected behavior** later.

Only use pip install in a conda environment as a last resort. By now 95% of packages available in pip are also available in conda. Just google "conda install packagename"

Yaml files

The best and easiest way to install virtual environments is by creating .yaml files that contain all the packages of your virtual environment. This way, the conda package manager can check beforehand which version numbers of all the packages work together.

They are regular text files with a .yml or .yaml file ending and look like this:

---

name: lab_python_course_env channels:

- conda-forge
- defaults

dependencies:

```
- jupyterlab
- matplotlib
- numpy
- scipy
- pandas
- altair
- h5py
- openpyxl
- vega_datasets
- sympy
- dask
- ipywidgets
- ipympl
- nodejs
- conda-forge::ffmpeg
- conda-forge::jsonschema-with-format-nongpl
- conda-forge::webcolors
```

install by running this command in the command line while in the same folder as the .yml file:

> `conda env create --file lab_python_course_env.yml`

In .yml files **name** specifies the environment name, **channels** tell conda where to look for the packages, **dependencies** are the libraries that you want.

Add Conda to Powershell

In Windows out of the box you unfortunately have to deal with multiple shells. To add the functionality of conda commands in Windows Powershell use these two commands.

In Anaconda Prompt:

> `conda init powershell`

In Powershell:

> `Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser`

Activating environments

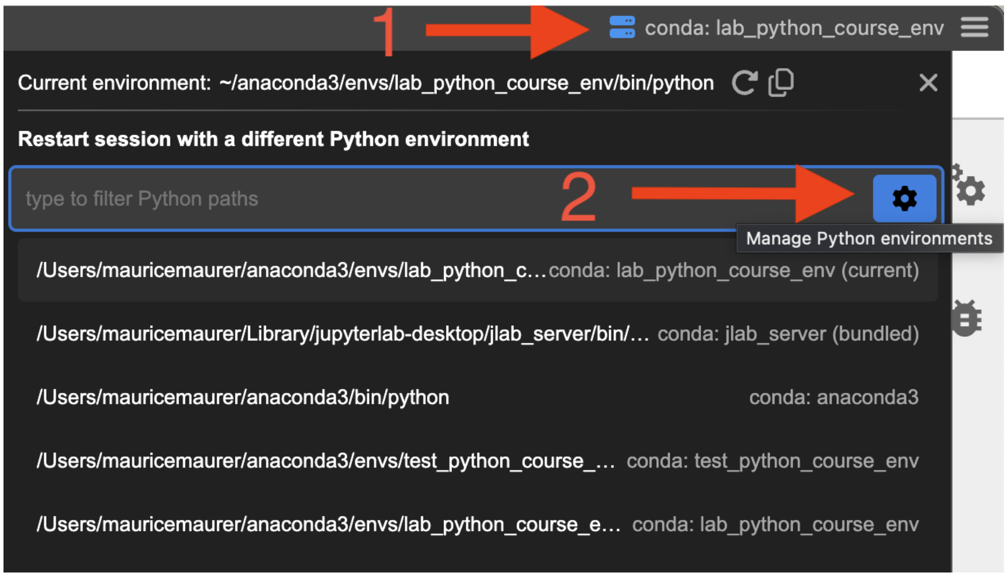
After installation you need to activate the environment, it does not activate itself

> `conda activate python_course_env`

You need to do this every time you start a new shell as the default environment is the base environment

Default environments in Jupyter Lab

- Click on your active environment in the top right
- Click on the gear icon



Current environment: ~/anaconda3/envs/lab_python_course_env/bin/python

Restart session with a different Python environment

type to filter Python paths

/Users/mauricemaurer/anaconda3/envs/lab_python_c… conda: lab_python_course_env (current)

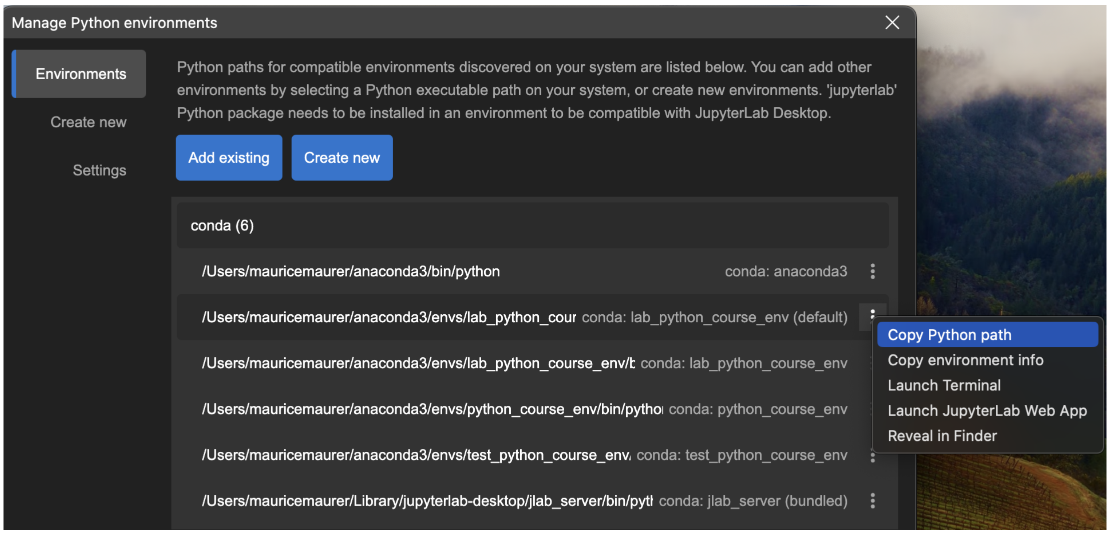
/Users/mauricemaurer/Library/jupyterlab-desktop/jlab_server/bin/… conda: jlab_server (bundled)

/Users/mauricemaurer/anaconda3/bin/python conda: anaconda3

/Users/mauricemaurer/anaconda3/envs/test_python_course_… conda: test_python_course_env

/Users/mauricemaurer/anaconda3/envs/lab_python_course_e… conda: lab_python_course_env

- Copy the Python Path from the environment you want to make default

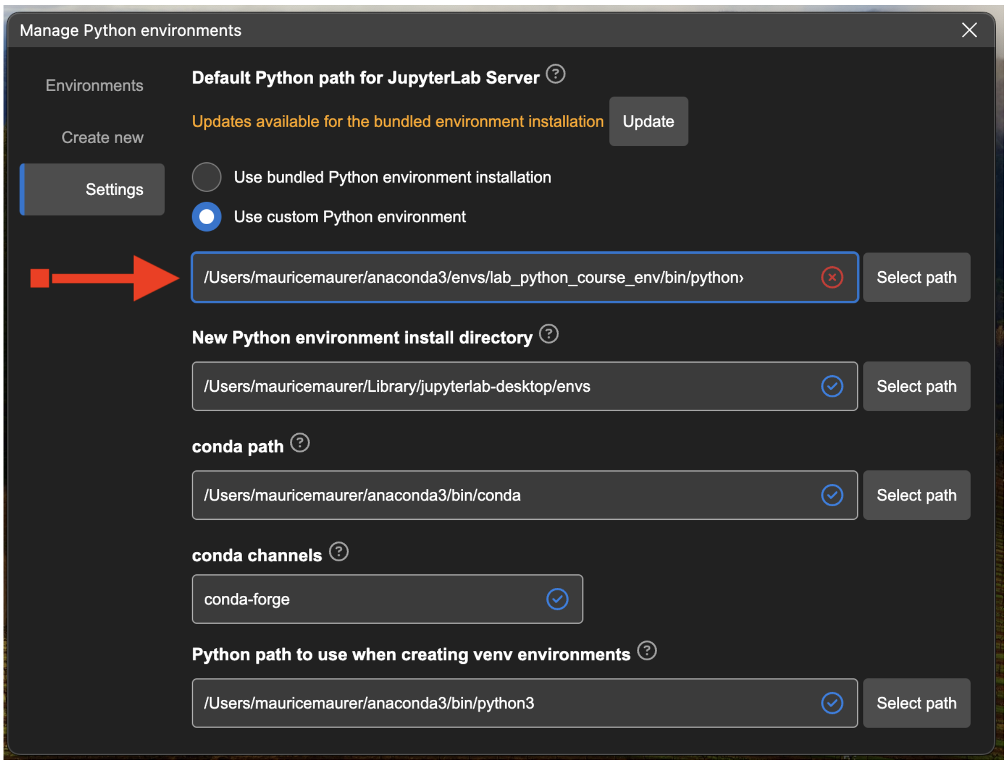

Manage Python environments

Python paths for compatible environments discovered on your system are listed below. You can add other environments by selecting a Python executable path on your system, or create new environments. 'jupyterlab' Python package needs to be installed in an environment to be compatible with JupyterLab Desktop.

conda (6)

/Users/mauricemaurer/anaconda3/bin/python conda: anaconda3
/Users/mauricemaurer/anaconda3/envs/lab_python_cour conda: lab_python_course_env (default)
/Users/mauricemaurer/anaconda3/envs/lab_python_course_env/t conda: lab_python_course_env
/Users/mauricemaurer/anaconda3/envs/python_course_env/bin/pytho conda: python_course_env
/Users/mauricemaurer/anaconda3/envs/test_python_course_env conda: test_python_course_env
/Users/mauricemaurer/Library/jupyterlab-desktop/jlab_server/bin/pytl conda: jlab_server (bundled)

- In the settings, paste it into the first box


Manage Python environments

Default Python path for JupyterLab Server

Updates available for the bundled environment installation

Use bundled Python environment installation [✓ Use custom Python environment]

/Users/mauricemaurer/anaconda3/envs/lab_python_course_env/bin/python

New Python environment install directory

/Users/mauricemaurer/Library/jupyterlab-desktop/envs

conda path

/Users/mauricemaurer/anaconda3/bin/conda

conda channels

conda-forge

Python path to use when creating venv environments

/Users/mauricemaurer/anaconda3/bin/python3

Installed environment are available everywhere on your computer, not just in the folder you installed them

> Ask Claude 🤖 💬 : `Give me a machine learning example in Python + the YAML file with the necessary libraries`

Anaconda Cloud

Anaconda cloud has **pre-installed** environments but you **can't install additional libraries** into them.

You can **install your own environment** with a .yaml file the same way you can do on your computer.

> **However**: for free you only get 5GB of storage, so make sure to **clear the cache** after installation (Disk usage->Clear Cache).

Exporting environments

You can export your current environment with the exact version numbers of all the libraries using

> `conda env export > export_env.yml`

This will guarantee that the other person has the **exact same setup** as you.

When you are using a version control software like **GIT** (we cover gid in the advanced courses), you can put the **.yaml file** for your Python code into the **repository**

Word of advice

Once you have a working environment, do not update/change it. If you need more modules later it is generally a better idea to create a new environment with the additional packages such that conda can check again which version numbers are compatible

First steps with numpy 🔢

By the end of this section, you will be able to:

1. Understand the basic concept and **benefits of NumPy arrays**
2. **Create** and manipulate NumPy arrays
3. Perform basic **mathematical operations** on NumPy arrays
4. Use NumPy's built-in functions for **array manipulation and analysis**
5. Recognize the **performance benefits** of NumPy over standard Python lists

Numpy arrays are stored continuously in memory. Processing can therefore be 100x faster than lists.

Built in functions for fast computation are written in C or C++

In [65]:
```python
import numpy as np
```

Initialize an array

In [66]:
```python
# Initialize 1D array

test_array = np.array([1, 2, 3, 4, 5])
print(test_array)
```
```
[1 2 3 4 5]
```

```
In [67]: # Convert a list to a numpy array

         test_list = [1, 3, 5, 7, 9]
         print(test_list)
```

```
[1, 3, 5, 7, 9]
```

```
In [68]: test_list_converted = np.asarray(test_list)
         print(test_list_converted)
```

```
[1 3 5 7 9]
```

```
In [69]: # Create an array with a range of values and the step in between
         arr1 = np.arange(start=-5, stop=5, step=0.5)
         arr1
```

```
Out[69]: array([-5. , -4.5, -4. , -3.5, -3. , -2.5, -2. , -1.5, -1. , -0.5,  0. ,
                 0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5])
```

```
In [70]: # Create an array in a range with a fixed number of values (symmetric)
         arr2 = np.linspace(start=-5, stop=5, num=21)
         arr2
```

```
Out[70]: array([-5. , -4.5, -4. , -3.5, -3. , -2.5, -2. , -1.5, -1. , -0.5,  0. ,
                 0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ])
```

Accessing array elements

Numpy arrays start counting at 0 (like in C):

```
In [71]: # Accessing 1D array

         arr = np.array([2, 4, 8, 16])

         print(f'First element: {arr[0]}')
```

```
First element: 2
```

```
In [72]: # Accessing array counting from the end (negative indexing)

         print(f'Last element: {arr[-1]}')
```

```
Last element: 16
```

Array Iteration

```
In [73]: # Simple iteration on 1D array similar to lists
         for x in arr:
             print(x)
```

```
2
4
8
16
```

```
In [74]: # Or use enumerate, if you want the loop iteration index
         for idx, x in enumerate(arr):
             print(f"index {idx}: {x}")
```

```
index 0: 2
index 1: 4
index 2: 8
index 3: 16
```

numpy.where for searching

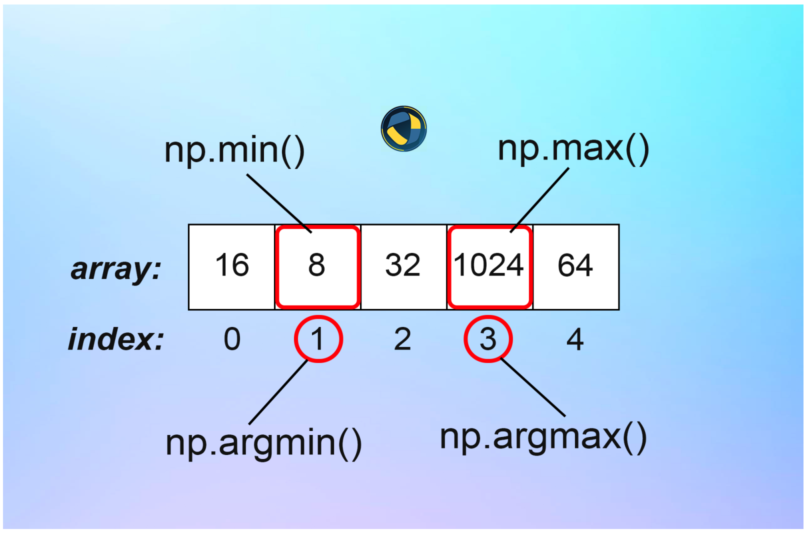
In [75]:
```
# Suppose you want to find the indices where the value of an array is 4:

np.where(arr == 4)
```

Out[75]:
```
(array([1]),)
```

np.min() np.max() etc.

In [76]:
```
arr = np.array([16, 8, 32, 1024, 64])
minimum = np.min(arr)
maximum = np.max(arr)
min_idx = np.argmin(arr)
max_idx = np.argmax(arr)

print(f"The minimum of arr is {minimum} at index {min_idx}")
print(f"The maximum of arr is {maximum} at index {max_idx}")
```

```
The minimum of arr is 8 at index 1
The maximum of arr is 1024 at index 3
```



In [77]:
```
num = 16
print(f"The squareroot of {num} is {np.sqrt(num)}")
```

```
The squareroot of 16 is 4.0
```

In [78]:
```
np.pi
```

Out[78]:
```
3.141592653589793
```

In [79]:
```
np.exp(2)
```

Out[79]:
```
7.38905609893065
```

In [80]:
```
np.abs(-4.9)
```

```
Out[80]: 4.9
```

Summary

This is not a full tutorial on Numpy, this is just a quick look at what external libraries can do.

In the advanced courses (https://training-scientists.com) we look at

- multidimensional array initialization, access, slicing and iteration
- joining / stacking of arrays
- Filtering arrays (e.g. give me all values in the array larger than a certain value, or all even numbers)
- Performance comparisons between numpy arrays and lists
- pre-allocation of arrays
- linear algebra
- statistics
- Fourier Transforms

and a lot more hands on examples of everything numpy can do like

- interpolation
- fitting
- filtering noise out of large data sets

> Ask Claude 🤖💬: `What are the main advantages of using NumPy arrays over Python lists for numerical computations?`

First steps with matplotlib 📊

By the end of this section, you will be able to:

1. **Create plots** using Matplotlib (line plots, scatter plots, histograms)
2. **Customize appearance** (colors, labels, titles, legends)
3. Generate and understand different **types of visualizations**

In [81]:
```python
import numpy as np
import matplotlib.pyplot as plt
```

x / y plot:

In [82]:
```python
# Generate two arrays to be plotted as x-values

x = np.linspace(start=0, stop=10, num=11)
x_fine = np.linspace(start=0, stop=10, num=101)
```

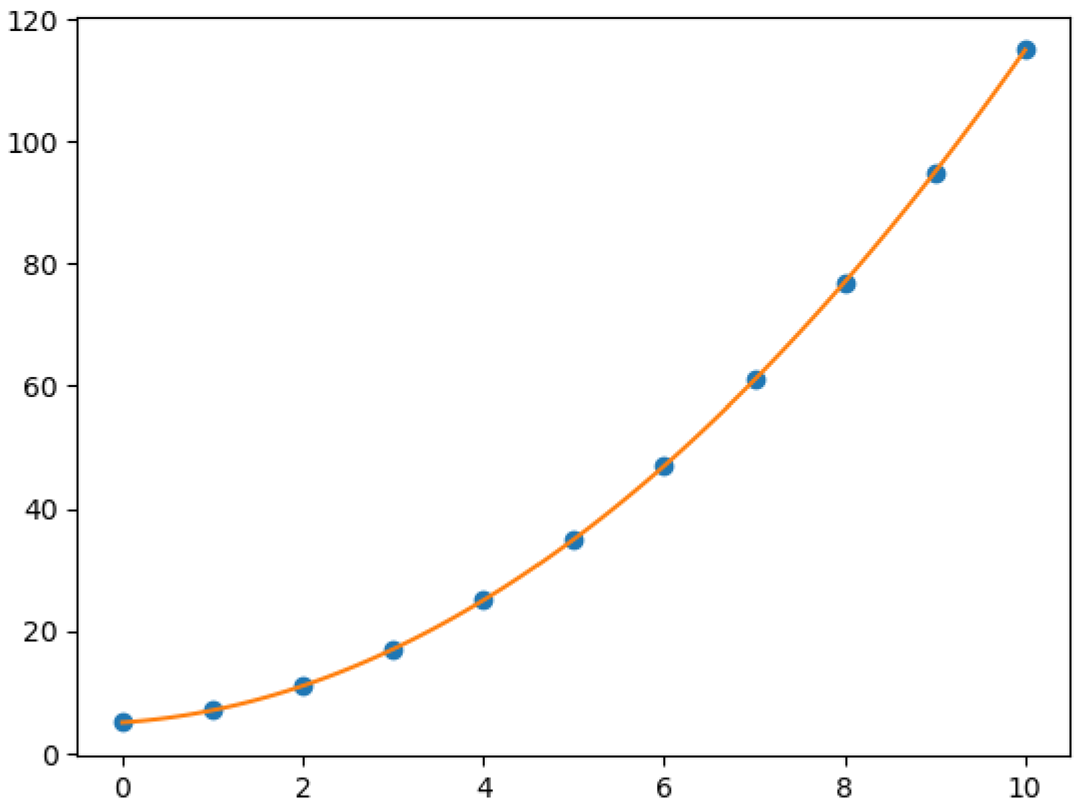
Polynomial function from functions section DO NOT define a function twice in a production script

```
def polynomial_function(x):
    return (x**2 + x + 5)
```

In [83]:
```
# Calculate the y values running the polynomial function on the x values

y = polynomial_function(x)
y_fine = polynomial_function(x_fine)
```

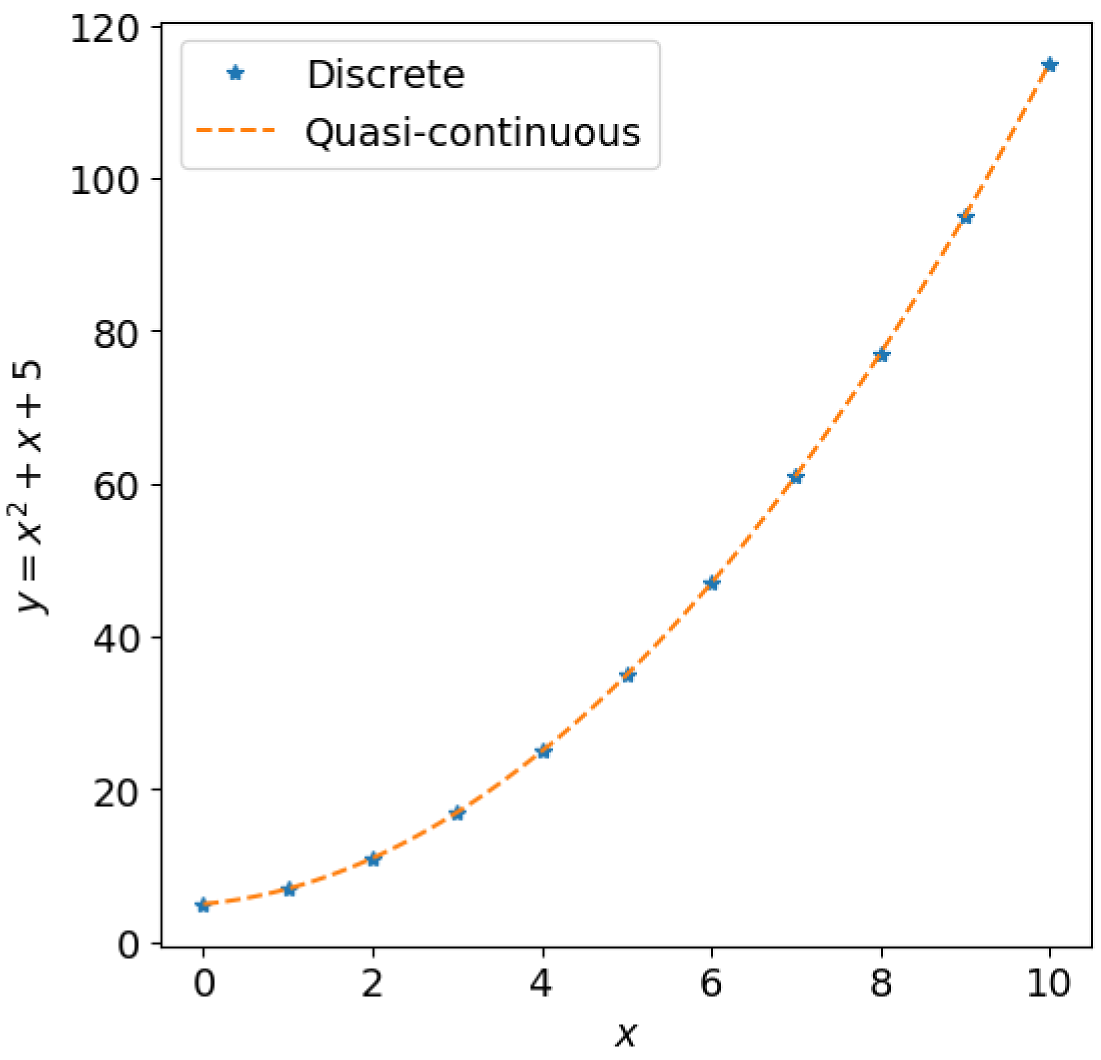
In [84]:
```
# Create a figure to plot these two x/y sets

plt.figure()
plt.plot(x,y,'o')
plt.plot(x_fine, y_fine)
```

Out[84]:
```
[<matplotlib.lines.Line2D at 0x10ff40a10>]
```



In [85]:
```
# Customize the plot

plt.rcParams.update({'font.size': 14}) # Increase font size (for entire n

plt.figure(figsize = (6,6)) # Change figure size and aspect ratio

plt.plot(x,y, '*', label='Discrete') # Add labels
plt.plot(x_fine, y_fine,ls='--', label='Quasi-continuous')

plt.xlabel('$x$') # Use LaTeX notation
plt.ylabel('$y = x^2 + x + 5$')

plt.legend(loc='upper left') # Add legend
```

```
Out[85]: <matplotlib.legend.Legend at 0x109aa4ce0>
```



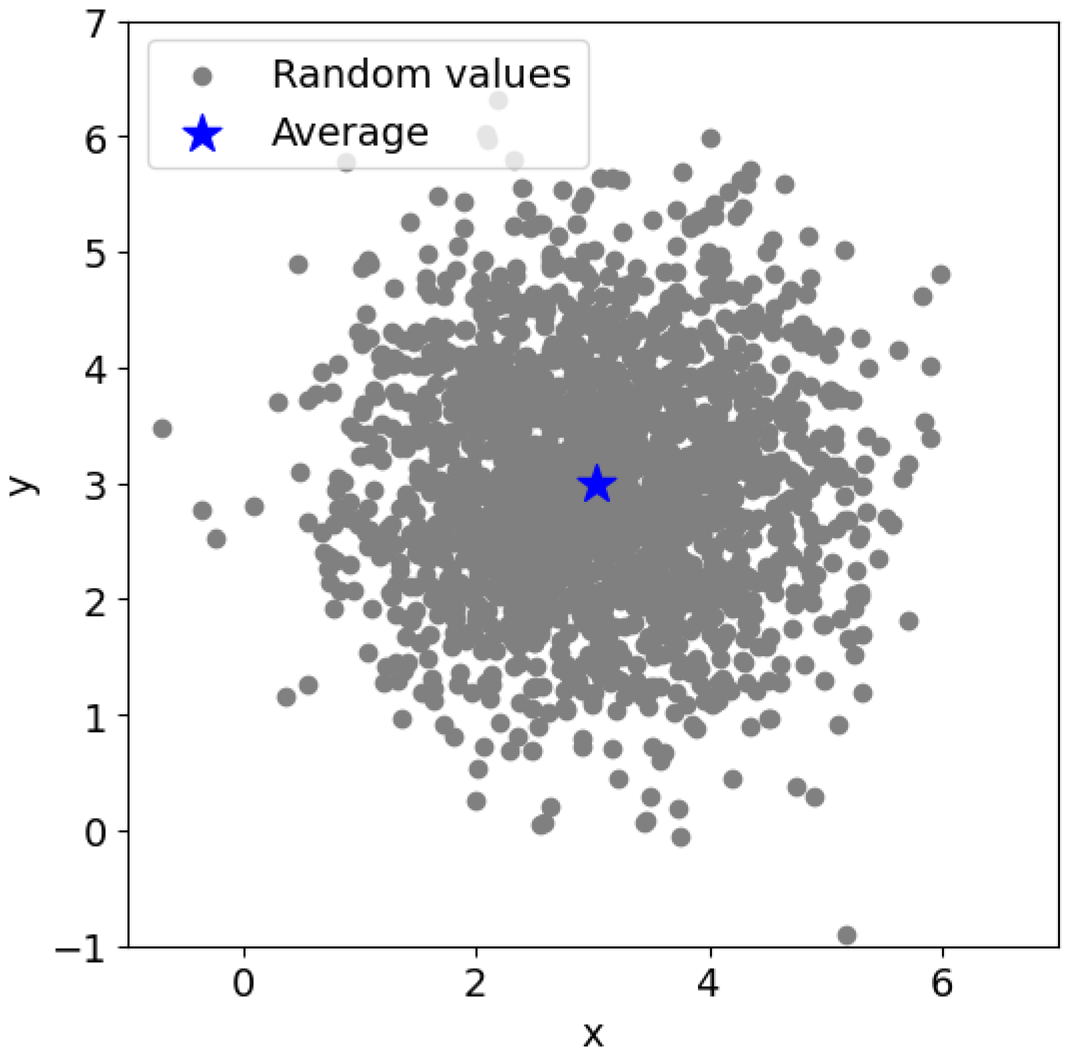
Scatter plots

```
In [86]: # Generate array of random numbers with normal distribution for x values:
x_rand = np.random.normal(loc=3, scale=1.0, size=2000)
# The same for y values:
y_rand = np.random.normal(loc=3, scale=1.0, size=2000)
```

```
In [87]: # define the dictionaries for plotting options:
random_dict = {"color": 'gray', "label": 'Random values'}
mean_dict = {"s":200, "color":'blue', "marker":'*', "label": 'Average'}
```

```
In [88]: # Plot as scatter

plt.figure(figsize=(6,6))
plt.scatter(x_rand, y_rand, **random_dict) # all values as orange points

# The mean (x,y) as a blue star:
plt.scatter(np.mean(x_rand), np.mean(y_rand), **mean_dict)
plt.xlabel('x')
plt.ylabel('y')
plt.xlim([-1 , 7])
plt.ylim([-1 , 7])
plt.legend(loc='upper left')
```

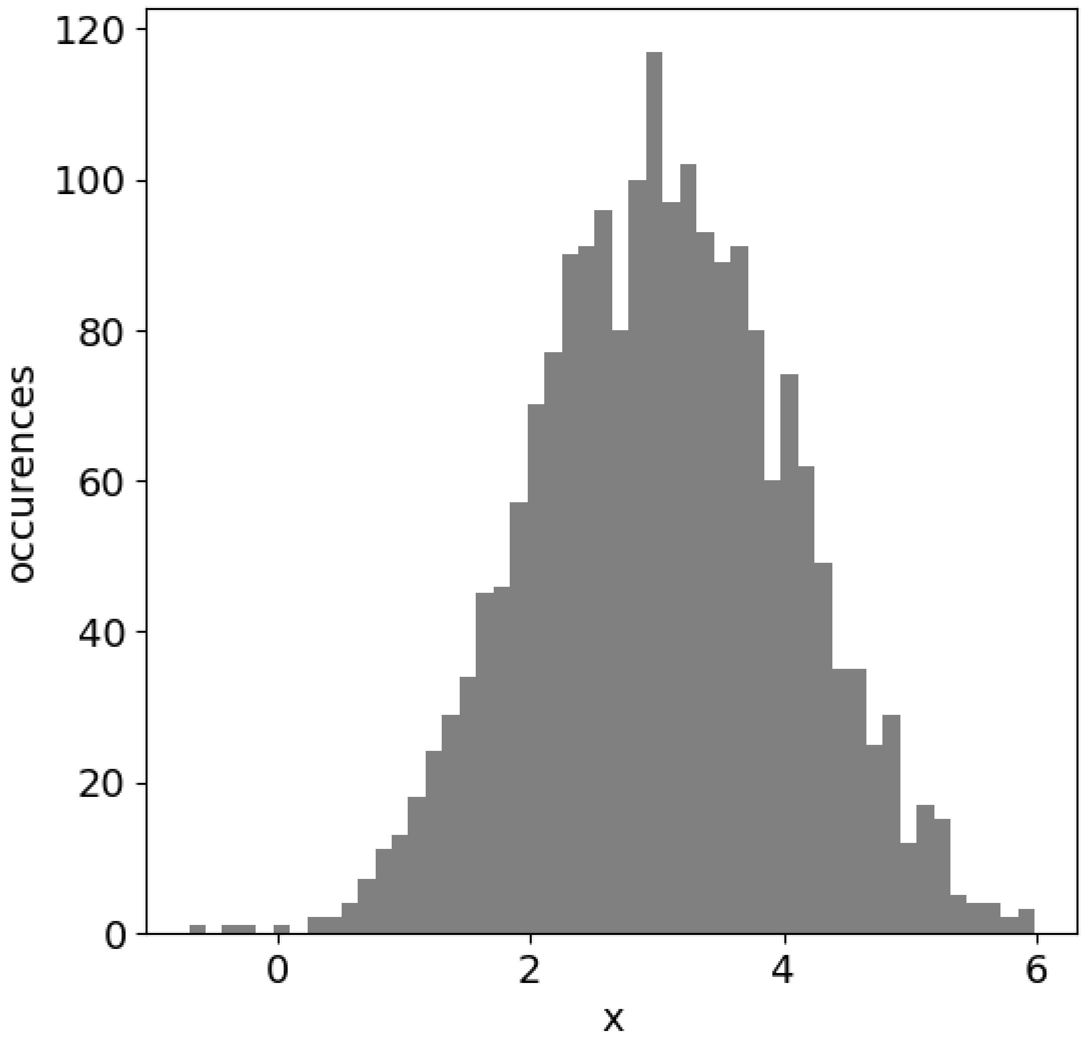
```
Out[88]:  <matplotlib.legend.Legend at 0x109ffe2a0>
```



Histogram

```
In [89]:  # Plotting the same data in a different way using the same plot style
          plt.figure(figsize = (6,6))
          plt.hist(x_rand, bins=50, **random_dict)
          plt.xlabel('x')
          plt.ylabel('occurences')
```

```
Out[89]:  Text(0, 0.5, 'occurences')
```

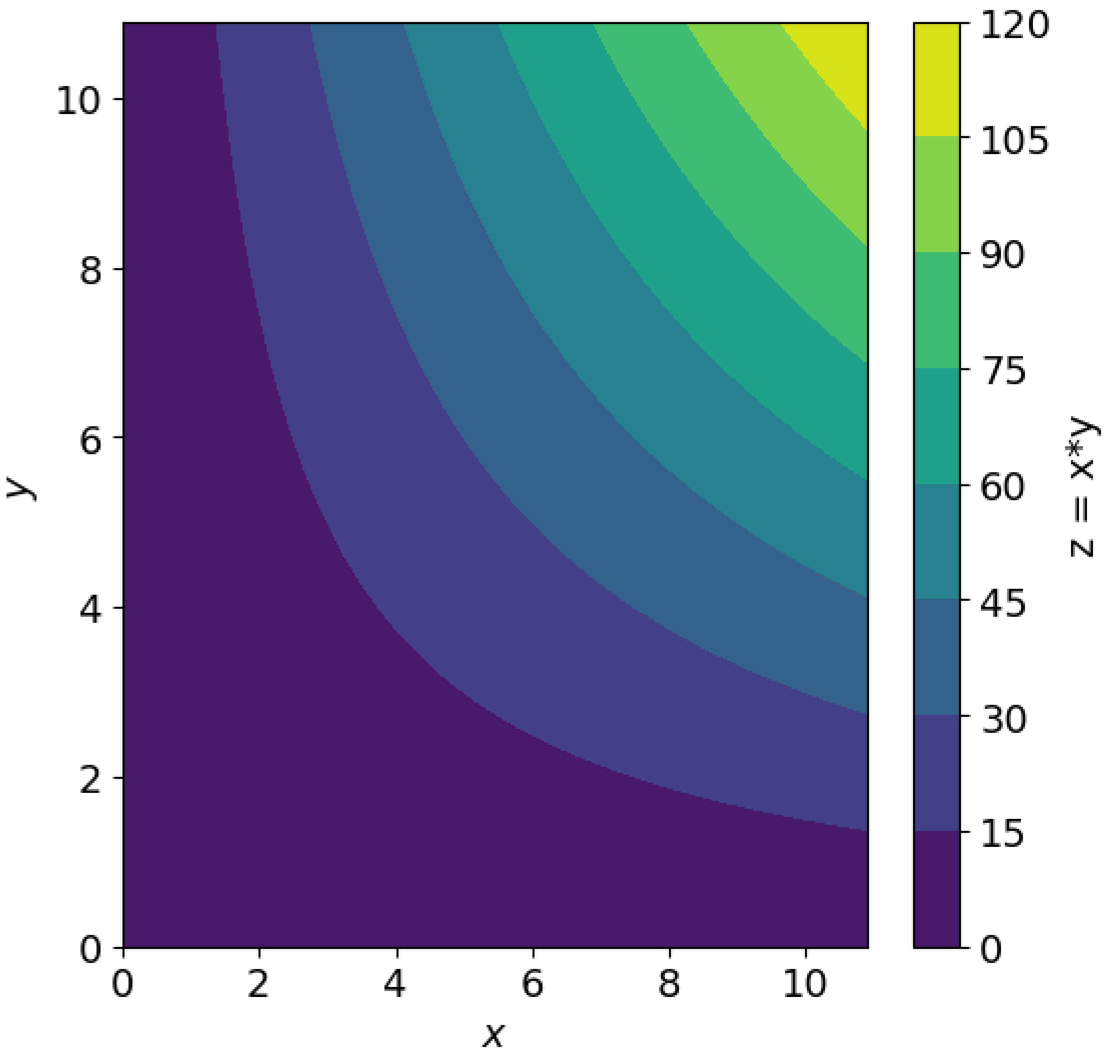


> **when we change the dictionary, all we need to do is rerun the code**

Contour plots

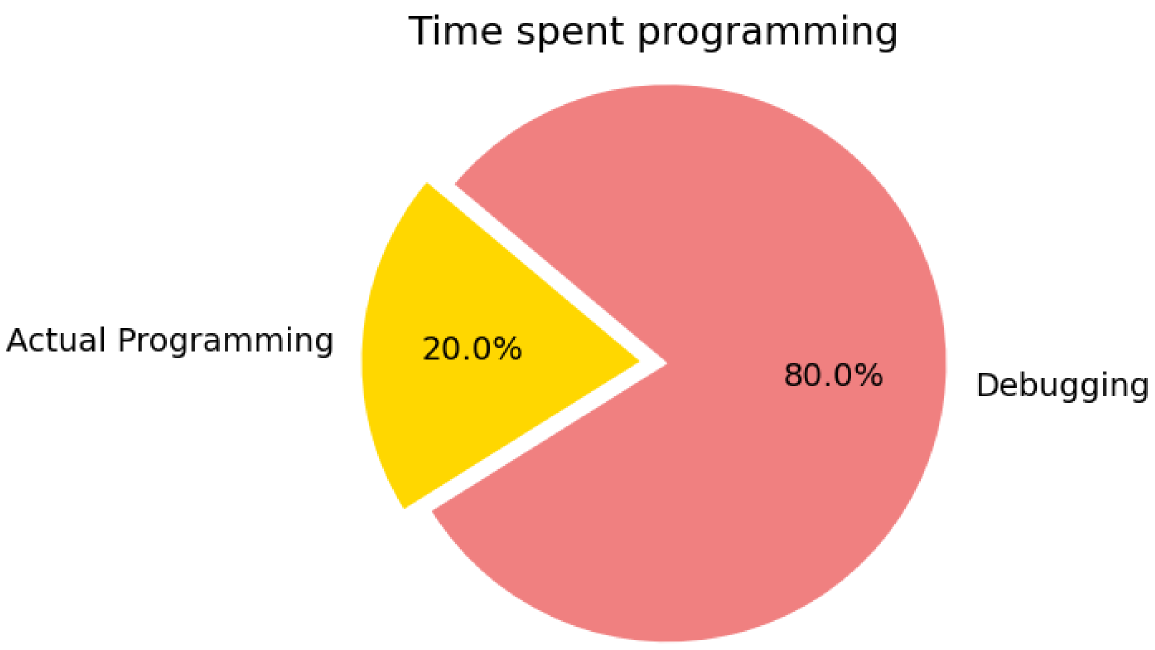
```
In [90]: x_contour = np.arange(0, 11, 0.1)
         y_contour = np.arange(0, 11, 0.1)
         # Create 2D meshgrid of x and y values:
         X,Y = np.meshgrid(x_contour, y_contour)
         z = X * Y

         plt.figure(figsize = (6,6))

         plt.contourf(X, Y, z) ##
         cbar = plt.colorbar() ##
         cbar.set_label('z = x*y') ##

         plt.xlabel('$x$')
         plt.ylabel('$y$')
```

```
Out[90]: Text(0, 0.5, '$y$')
```



Pie charts

In [91]:
```
sizes = [20, 80]
labels = ['Actual Programming', 'Debugging']
colors = ['gold', 'lightcoral']
explode = (0.1, 0)  # explode 1st slice

plt.pie(sizes, explode=explode, labels=labels, colors=colors, autopct='%1
plt.axis('equal')
plt.title('Time spent programming')
plt.show()
```



Summary

Matplotlib is a powerful plotting library that can produce professional looking plots.

In the advanced courses we will look at

- inset Plots
- interactive plots & widgets
- creating videos from a series of plots
- advanced plotting options
- making plots publication ready

If you are interested:

- Python Basics
- Python for Scientists & Engineers
- Python for Biologists

> Ask Claude 🤖💬: `What are the key components of a matplotlib figure and what does each do?`

> Ask ChatGPT 🤖💬: `Create a matplotlib plot with multiple y axes`

FAQ ❓ & Common issues I see beginners struggle with

Whenever I look at your code it makes sense, when I try it I get errors 💻❓

Learning programming is like learning anything else. Think of learning a new language or a music instrument.

It is one thing to watch someone else play the piano and think "He plays well". It is another thing entirely to be able to play yourself.

A few tips:

- keep the code cells small. It is easier to find a bug in a cell with 3 lines of code than to find the needle in the haystack
- work your way forward with as small increments as possible. Write a line of code, execute. Write another line, execute etc.
- learn how to read error messages (the important stuff is always in the last line)
- use AI tools. Copy/paste buggy code to ChatGPT/Claude or use Anaconda Cloud. With basic errors they can usually help

How to think like a programmer? 🤔

Tricky question. In this video I showed you the tools you need to get started. Imagine you're a handyman and just learned how to use a screwdriver, hammer, saw etc. 🛠️ Instead of trying to build an entire house now, start small. Build a birds house. Then build a dogs house. Then a garage. And THEN try to build a house.

A few tips 🧩 :

- programmers usually have a "divide and conquer" approach to solving problems. Break the project you have into as many small parts as possible and then solve them one by one. You climb a mountain one step at a time. Don't try to jump to the top in one go. AI tools might help you get there faster and be your Sherpa but they might also lead you on the wrong path. Divide and Conquer example:
  - File IO 💾
    - What format is the data in that in want to read in?
    - How do I read that type of data in?
    - In what kind of data structure do I want to store the data in Python?
    - Read in the data
    - Was the data read in correctly? (Compare the array in Python with the data in the file)
  - Data analysis 🔍
    - Here it might make sense to work backwards and start with the question: how do I want the end result to look like?
    - decide for the method of data processing: interpolation, fitting, filtering etc.
    - process the data
    - check if the result makes sense
  - Visualization 📊
    - decide for the type of plot (histogram, contour plot etc.) that best represents your result and the point you are trying to make with it

    - plot the end result of the data analysis
    - make the plot publication ready
- Learn from other people. Like watching DIY home improvement videos you can learn from how other people solve problems. I will create some videos like this in the future, so subscribe ;)

Naming Variables: ❌❌❌ Avoid Keywords and Built-in Functions 💀💀💀

Basically, if the name turns green, it is a reserved name

In [92]:
```
# Absolutely do not do this:
#for = 3
```

Python Keywords and Built-in Functions Reference 🐍

Keywords 🔑

Keywords	Keywords	Keywords	Keywords	Keywords	Keywords	Keywords
`False`	`None`	`True`	`and`	`as`	`assert`	`async`
`await`	`break`	`class`	`continue`	`def`	`del`	`elif`
`else`	`except`	`finally`	`for`	`from`	`global`	`if`
`import`	`in`	`is`	`lambda`	`nonlocal`	`not`	`or`
`pass`	`raise`	`return`	`try`	`while`	`with`	`yield`

Common Built-in Functions 🧰

Functions	Functions	Functions	Functions	Functions
`abs()`	`all()`	`any()`	`ascii()`	`bin()`
`classmethod()`	`compile()`	`complex()`	`delattr()`	`dict()`
`filter()`	`float()`	`format()`	`frozenset()`	`getattr()`
`id()`	`input()`	`int()`	`isinstance()`	`issubclass()`
`max()`	`memoryview()`	`min()`	`next()`	`object()`
`property()`	`range()`	`repr()`	`reversed()`	`round()`
`str()`	`sum()`	`super()`	`tuple()`	`type()`

Note:

- Keywords are reserved and cannot be used as identifiers.
- Built-in functions are predefined but can be overwritten (not recommended).
- `True`, `False`, and `None` are constants but treated as keywords.
- This list is based on Python 3.x and may vary slightly in different versions.

What's the difference between `=` and `==` in Python?

- `=` is the assignment operator. It's used to assign a value to a variable. Example: `x = 5` assigns the value 5 to the variable x.
- `==` is the equality comparison operator. It's used to check if two values are equal. Example: `if x == 5:` checks if the value of x is equal to 5.

Which one of the AI tools should I be using?

- If you don't want to spend any money I would use all 3 and take advantage of the free usage per day limits
- **Anaconda Cloud** is great for in-place debugging

- **Claude** is great with its Artifacts and versions
- **ChatGPT** just released ChatGPT o1 which takes more time to "think" which probably just overtook Claude 3.5 Sonnet

How do I choose between using a list, tuple, or dictionary?

- Use a **list** when you have a collection of related items that may change (mutable) and order matters. Example: `todo_list = ['Study', 'Exercise', 'Cook']`
- Use a **tuple** for collections of items that shouldn't change (immutable) and order matters. Example: `coordinates = (4, 5)`
- Use a **dictionary** when you want to store key-value pairs for quick lookup. Example: `person = {'name': 'Alice', 'age': 30, 'city': 'New York'}`

What are some common Python libraries for data analysis and when should I use them?

Answer:

- **NumPy**: For numerical computing and working with arrays. Use when you need to perform mathematical operations on large datasets efficiently.
- **Pandas**: For data manipulation and analysis. Great for working with structured data in tables or time series.
- **Matplotlib**: For creating static, animated, and interactive visualizations. Use when you need to create basic plots and charts.
- **SciPy**: For scientific and technical computing. Use for more advanced statistical functions, optimization, and signal processing.
- We cover all of them in the advanced courses
  - Python Basics
  - Python for Scientists & Engineers
  - Python for Biologists

How can I collaborate on Python projects with others using version control?

Use Git as your version control system and GitHub, GitLab, or Bitbucket as your remote repository host. We set this up in the advanced courses, try it out and use it together with practical examples.

How can I improve my skills further?

1. The exercises for this course are available on https://training-scientists.com. To get a 50% discount, leave a like and a comment under the video and contact me on LinkedIn.
2. Go ahead and play around with this notebook. If anything is unclear, delete it and see what happens. Break things and understand what they are good for. If you

are afraid of breaking things, just create a copy of the notebook.
3. Try solving small problems (that ideally are relevant to you) and start coding. Use AI tools to help you but always understand what you are copy pasting.
4. Consider signing up for one of my advanced courses with lots of hands-on exercises, live Zoom sessions to discuss exercises, ask questions and discuss topics beyond the course

   > "Ask ChatGPT: Can you suggest some small Python projects for beginners to practice their skills?"

Glossary 📚

Term	Definition
Variable	A named storage location in computer memory that holds data.
Data Type	A classification of data which tells the compiler or interpreter how the programmer intends to use the data.
Function	A block of organized, reusable code that performs a specific task.
List	An ordered, mutable collection of elements in Python.
Tuple	An ordered, immutable collection of elements in Python.
Dictionary	An unordered collection of key-value pairs in Python.
Loop	A programming construct that repeats a group of commands.
Conditional Statement	A programming language construct that performs different computations or actions depending on whether a boolean condition evaluates to true or false.
Scope	The region of a program where a variable is recognized and can be used.
Indentation	The spaces at the beginning of a code line used to determine the grouping of statements in Python.
String	A sequence of characters in Python, typically used to represent text.
Boolean	A data type that has one of two possible values: True or False.
Index	A number representing the position of an element in a sequence (like a list or string).
Slice	A portion of a sequence, specified by a range of indices.
Iteration	The process of repeatedly executing a set of statements.

Best Practices Summary 🌟

Throughout this course, we've covered several best practices for Python programming. Here's a summary of key points to remember:

1. **Code Readability**

- Use descriptive variable and function names
- Follow PEP 8 guidelines for code style (covered in more detail in the advanced courses)
- Keep lines of code between 79-99 characters long
- Use comments to explain 'why', not 'what'

2. **Function Design**

   - Keep functions small and focused on a single task
   - Use parameters instead of relying on global variables
   - Return values rather than modifying global state

3. **Variable Scope**

   - Keep variable scope as small as possible
   - Avoid using global variables within functions

4. **Data Structures**

   - Choose the appropriate data structure (list, tuple, dictionary) for your needs
   - Use NumPy arrays for numerical computations when performance is crucial

5. **Virtual Environments**

   - Use virtual environments to manage dependencies for different projects
   - Keep your base Python installation clean

Remember, writing clean, readable, and maintainable code is a skill that develops with practice. Keep these best practices in mind as you continue your Python journey!

> "Ask ChatGPT: Can you explain the concept of DRY (Don't Repeat Yourself) in programming and give an example in Python?"