PYTHON CHEATSHEET

VARIABLES

Variables are a fundamental concept in Python. Python is a **dynamically typed language**, meaning you can reassign a variable at any point, and it will update to the **latest assigned value**, even if the type changes.

Variables in Python can hold **any data type**, such as numbers, strings, lists, or more complex structures like dictionaries.
They can also be **used in operations**, **passed to functions**, **returned from functions**, and **combined** to build more complex expressions.

Variables are **case-sensitive**, meaning `Value` and `value` are considered different, and they follow naming rules such as starting with a letter or underscore, and not using reserved keywords

```
my_var = 1
my_var = "Hello world"
```

now the variable is set to “Hello world” and not 1

ARITHMETIC OPERATORS

Here a list with a brief explanation (if needed) of the most common **ARITHMETIC** operators

- **Sum :** $+$
- **Subtraction:** $-$
- **Multiplication:** $*$
- **Division:** $/$
- **Power of:** $**$
- **Modulo:** %
  The modulo operator provides the remainder *r* of a division. The remainder comes from the equation
  $$a = b * q + r$$
  Where *a* and *b* the dividend and the divisor, and *q* the quotient.

```
8 % 2 = 0  # 8 divides evenly by 2  2 * 4 = 8; q = 4), with no remainder (r = 0)
9 % 3 = 0  # 9 divides evenly by 3 (3 * 3 = 9; q = 3) with no remainder (r = 0)
7 % 4 = 3  # 4 goes into 7 once (4 * 1 = 4; q = 1) with remainder 3 (r = 7 - 4 = 3)
9 % 2 = 1  # 2 goes into 9 four times (2 * 4 = 8; q = 4) with remainder 1 (r = 9-8 = 1)
2 % 8 = 2  # 8 goes into 8 zero times (q = 0) with remainder 2 (from equation a = r = 2)
```

The modulo is a very useful operator since it's used in many logical istances. For example it can be used to determine wether an element is even or odd: in fact, if *x % 2 = 0*, then x is even, else x is odd.

- **Floor division:** $//$
  This operator gives back the quotient of a division, ignoring the remainder or discarding decimal parts.

```
# 7 / 2 = 3.5 # normal division returns the exact result including the decimal part
# 7 // 2 = 3  # ignores the decimal part 0.5 and keeps only the integer quotient (q = 3)
# 7 // 4 = 1  # 4 fits in 7 once (4 * 1 = 4; q = 1), ignores the remainder (r = 3)
# 2 // 8 = 0  # 8 does not fit in 2 (q = 0), ignores the remainder (r = 2)
```

- **Aumented assignment operator:** $+=$
  Augmented assignment operators perform a **calculation and assignment in a single step**, such as adding, subtracting, or multiplying a variable with itself and **storing the result back**. They are typically used when we need to **update a variable**, for example, as a **counter in a while loop**

```
+=  # Add and assign           x += 2  => x = x + 2
-=  # Subtract and assign       x -= 2  => x = x - 2
*=  # Multiply and assign       x *= 2  => x = x * 2
/=  # Divide and assign x /= 2  x = x / 2
//= # Floor divide and assign   x //= 2 => x = x // 2
%=  # Modulo and assign x %= 2  x = x % 2
**= # Exponentiate and assign   x **= 2 => x = x ** 2
```

COMPARISON OPERATORS

Here a list with a brief explanation (if needed) of the most common **COMPARISON** operators

- **Equal:** ==
- **Not Equal:** ! =
- **Greater Than**: >
- **Less than:** <
- **Greater than or Equal to:** >=
- **Less than or equal to:** <=

DATA TYPES

There are mainly 5 types of data built-in in Python (not considering more advanced or custom types):

1) **Numeric**: integers, float, complex
2) **Boolean:** True, False
3) **Sequence:** list, strings, tuples.
4) **Map**: dictionary
5) **Set**: set, frozenset

The first two data types are commonly referred to as **"Primitive Types"** (less commonly **"Atomic"**, or **"Built-in"** as described in Python's documentation), since they cannot be broken down into smaller objects. The last three are known as **"Data Containers"**, due to their ability to store and organize multiple values.

Primitive types:

Primitive data types are non iterable, as they are single elements

1) NUMERIC

Numeric types are used to represent numbers in Python. They include integers (int), floating-point numbers (float), and complex numbers. They are used for mathematical operations, counting, and calculations.

The most common number types are integer or floating-point numbers (decimals). The last type is especially useful when performing certain operations like division, as the result might be a non-integer number.

Numeric data types work with arithmetic operator as they normally do

```
1 + 1
```

Output:
2

2) BOOLEAN

Boolean is a data type used to represent logical values. It can only take two possible values: True or False. Booleans are commonly used in conditions, comparisons, and control flow to determine whether a statement is true or false.

```
3>5
```

Output:
False

```
3<5
```

Output:
True

Container types:

Data containers are data types designed to store and organize multiple values.
They can hold elements of the same or different types, and include structures like lists, tuples, sets, and dictionaries.
They allow grouping, accessing, and manipulating collections of data efficiently **through operations like indexing and slicing.**

All Data Containers are iterable, meaning that it’s possible to loop through their elements (not alla iterables are data containers

Indexing and slicing

When talking about data containers, **indexing** and **slicing** are two of the most important operations for **sequences** (like strings, lists, and tuples).
These operations allow us to **select either a specific item** (indexing) or **a range of items** (slicing) from the container.

In basic Python, the previous actions are applied to Pythons’ native data containers, so indexing retrieves only one element from a list, touple, dict, etc. and slicing referes to a raneg (star-end) of elements from the same containers. This is not the case in other libraries (Pandas, Numpy) where both actions apply differently to the specific library data container (scalar in Pandas’ series, scalar/rows/columns in Numpy’s arrays, , cells(scalar)/rows/columns in Pandas’dataframe or series), therefore the definition of single element and set of elements changes accordingly.
We’ll see concrete applications of these concepts on different data containers and libraries in the coming sections.

NOTE: if an object cannot index, it cannot be sliced eiother

NOTE: These operations do **not apply** to **sets** (because they are unordered) or to **dictionaries** in the same way. Dictionaries use **keys** for lookup, not indexes.

- **Indexing:**

```
my_list = ['a', 'b', 'c', 'd']
print(my_list[0])   # 'a' (first element)


print(my_list[-1])  # 'd' (last element)


nested_list = [["apple", "banana"],["carrot", "dates"],["eggplant", "fig"]]
print(nested_list[1][0])  # 'carrot'
print(nested_list[2][1])  # 'fig'
```

- **Slicing:**

```
my_list = ['a', 'b', 'c', 'd', 'e']


# Basic slicing: from index 1 to index 3 (excluding index 3)
print(my_list[1:3])  # ['b', 'c']


# Omitting the start (defaults to 0)
print(my_list[:3])   # ['a', 'b', 'c']


# Omitting the end (defaults to the end of the list)
print(my_list[2:])   # ['c', 'd', 'e']


# Full copy
print(my_list[:])    # ['a', 'b', 'c', 'd', 'e']


# Slicing with step (every second element)
print(my_list[::2])  # ['a', 'c', 'e']


# Reversing the list using slicing
print(my_list[::-1]) # ['e', 'd', 'c', 'b', 'a']
```

Charateristics of Data Structures

Mutability/Inmutability:

- **Mutability:** You can **modify the object "in place"** without needing to reassign it to a new variable. The object **keeps the same memory address.**
  Examples: `list`, `set`, `dict`

```
my_list = [1, 2, 3]
print(id(my_list))  # e.g., 140022048


my_list.append(4)   # Modifies in place
print(id(my_list))  # Same memory address (140022048)
```

- **Inmutability:** You **cannot change the object itself**. Any "change" creates a **new object**, requiring **reassignment to a new variable**. The memory address **changes**.
  Examples: `tuple`, `str`, `int`, `float`

```
my_string = "Hello"
print(id(my_string))  # e.g., 140033456


my_string += " World"  # Creates a new string
print(id(my_string))  # Different memory address (140033600)
```

Ordered/unordered

- **Ordered:** The **position of elements matters**. You can access items by **index/position** (starting at 0). Repeating the same elements keeps the **same sequence.**

Examples: `list`, `tuple`, `str`, `range`

```
my_list = [10, 20, 30]
print(my_list[0])  # 10
```

- **Unordered:** The **position does not matter**. Items are **not accessed by index**, but by **key** or **presence**. The storage order is **not guaranteed conceptually**, even if it appears ordered in Python 3.7+.
  Examples: `set`, `dict` (Ordered as of Python 3.7+, but conceptually still accessed by key, not position)

```
my_set = {"apple", "banana", "cherry"}
for item in my_set:
    print(item)  # Order not guaranteed

my_dict = {"name": "Tyron", "age": 30}
print(my_dict["name"])  # Access by key, not position
```

NOTE: unordered object canno be indexed, therefore cannot be sliced

Allows repetition
It describes whether the **same value can appear multiple times** in a data structure **without being automatically removed or rejected**.
Allowing repetition is useful when **duplicates are meaningful**, like in shopping lists, logs, or sequences where order and quantity matter.

- **Allows repetion:** `list`, `tuple`, `str`, `dictionary values`
- **Unique:** `set, dictionary keys`

---

3) SEQUENCES

3.1) Strings (str)

A string is a sequence of characters used to store and represent text in Python.
This data type defined by **single**, **double**, or **triple quotes**, and they are **immutable**, meaning their content cannot be changed after creation.

- **Ordered**: The position of items matters, and items are **indexed starting from 0**.
- **Immutable**: Once created, **you cannot add, remove, or change** items.
- **Allows Duplicates**: Tuples can contain **repeated values**.

Strings may not seem like sequences in the everyday sense, but in Python, they belong to this group because they store an ordered series of characters and support indexing, slicing, and iteration. However, since they represent text and act as a single value in many contexts, they are sometimes treated as "primitive" for simplicity

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

3.1.1) String purpose:

Store a **sequence of characters**, **immutable** and **ordered**. Ideal for **text processing**, **pattern searching**, and **displaying messages**.

3.1.2) Creation of a string:

- **Direct assignment/manually**

```
s = "Hello"
```

- **Concatenation (joining strings with +)**

```
s = "Hello" + " " + "World"  # "Hello World"
```

- **converting an objest using str()**

```
num = 42
s = str(num)  # "42"
```

- **Repetition (repeating with *)**

```
s = "Ha" * 3  # "HaHaHa"
```

- **Joining lists into strings (join)**

```
words = ["Python", "is", "fun"]
s = " ".join(words)  # "Python is fun"
```

- **Formatted or f-string**

```
Name = "Tyron"
s = f"Hello, {name}"  # "Hello, Tyron"
```

3.1.3) Operations with strings (all the following blocks need print to get the result):

- **Indexing (accessing characters by position)**

```
s = "Python"
s[0]  # 'P'
s[-1] # 'n'
```

- **Slicing (extracting substrings)**

```
s[0:3]  # 'Pyt'
s[2:]   # 'thon'
s[0::2] # 'Pto'
```

- **Length check with len()**

```
len("Python")  # 6
```

- **Membership check with in**

```
"Py" in "Python"  # True
```

- **Changing case (upper, lower, capitalize)**

```
"hello".upper()      # 'HELLO'
"HELLO".lower()      # 'hello'
"python".capitalize() # 'Python'
```

- **Replacing substrings (replace)**

```
"Hello World".replace("World", "Python")  # 'Hello Python'
```

- **Stripping spaces (strip)**, removes the spaces from the beginning and end but not in the middle

```
"  hello  ".strip()  # 'hello'
```

- **Splitting into lists (split)**, splits the string into parts whenever it finds a comma (or other characters), returning a list of the parts

```
"a,b,c".split(",")  # ['a', 'b', 'c']
```

3.2) Lists

A **List** in Python is an **ordered**, **mutable** (modifiable) collection of items, which can hold **elements of any type** (numbers, strings, other lists, etc.).

- **Ordered**: The position of items matters, and items are **indexed starting from 0**.
- **Mutable**: Once created, **we can add, remove, or change** items.
- **Allows Duplicates**: Tuples can contain **repeated values**.

Lists are **indexed**, starting at 0, and can **contain duplicates**.

3.2.1) List purpose

Stores an **ordered sequence** of **any items**, allowing **duplicates** and **frequent modifications**.
Ideal when **order matters** and the data needs to **grow or change**.

```
my_list = [1, 2, 3]
print(id(my_list))  # Example memory address

my_list.append(4)   # Modifi
es the original list
print(my_list)      # [1, 2, 3, 4]
print(id(my_list))  # Same memory address
```

3.2.2) Creation of a list:

- **Direct Assignment/manually, with square parentheses** [ ]

```
my_list = [1, 'Apple', 3.14, 4, True]
```

- **Using the list() constructor**

```
# From a string (splits into characters)
char_list = list("hello")  # ['h', 'e', 'l', 'l', 'o']

# From a tuple
tuple_data = (1, 2, 3)
list_from_tuple = list(tuple_data)  # [1, 2, 3]

# From another iterable like range
```

```python
list_from_range = list(range(1,6))  # [0, 1, 2, 3, 4,5]
```

- **List comprehension (efficient and Pythonic)**

```python
# Squares of numbers from 0 to 4
squares = [x**2 for x in range(5)]  # [0, 1, 4, 9, 16]

# Conditional list comprehension
even_numbers = [x for x in range(10) if x % 2 == 0]  # [0, 2, 4, 6, 8]
```

- **Using .split() on strings**

```python
# Splitting by spaces
words = "hello world".split()  # ['hello', 'world']

# Splitting by a custom separator
csv_line = "a,b,c".split(",")  # ['a', 'b', 'c']
```

- **Multiplying to prefill list**

```python
zeros = [0] * 5  # [0, 0, 0, 0, 0]
```

- **Nested list**

```python
nested = [[1, 2], [3, 4], [5, 6]]
print(nested[0])      # [1, 2]
print(nested[0][1])   # 2
```

3.2.3) Operations with lists (mylist = [1, ‘Apple, 3.14, 4 True])

- **Indexing (accessing characters by position)**

```python
print(my_list[0])  # first element
```

- **Slicing (extracting substrings)**

```python
print(my_list[:2])  # [1, "apple"]
```

- **Length check with len()**

```python
print(len(my_list))  # Number of elements in the list = 5
```

- **Membership check with in**

```python
print("banana" in my_list)  # True or False
```

- **Modifying elements**

```python
my_list[0] = "updated value"
```

- **Adding elements**

```python
# Append to the end
my_list.append("new item")

# Insert at a specific position
my_list.insert(1, "banana")
```

- **Removing elements**

```
# Remove by value
my_list.remove("apple")

# Remove by index
del my_list[0]

# Remove and return the last item
last_item = my_list.pop()
```

- **Looping over a list**

```
for x in my_list:
   print(x)
```

3.2.3) Nested list

3.3) Tuples

A **tuple** is an **ordered**, **immutable** (unchangeable) data container that can **hold multiple items** of **any data type**, including mixed types.

- **Ordered**: The position of items matters, and items are **indexed starting from 0**.
- **Immutable**: Once created, **one cannot add, remove, or change** items.
- **Allows Duplicates**: Tuples can contain **repeated values**.

The main reasons why one could prefer to use Tuples intead of list, besides the obvius diffrence in “mutability”, are

- **Data Integrity**: Use tuples when **data should not be changed** (e.g., geographic coordinates).
- **Performance**: Tuples are **faster** and **use less memory** than lists.
- **Hashable**: Tuples can be used as **dictionary keys** or **set elements**, unlike lists.

NOTE: Python doesn’t allow tuple comprehension because it **already uses parentheses `()` for generator expressions**, which look very similar to what "tuple comprehension" would look like. On top or data, comprehension allows to build collections dynamically (often growing them element by element), but tuples are immutable. So this wouldn’t make sense, concetually.

3.3.1) Tuples purpose:

Store an **ordered, fixed collection** of **any items**, allowing **duplicates** but **not modifiable**. Ideal for **fixed groupings** like **coordinates** or **configurations** you **don’t want to change** and data integrity.

3.3.2) creation of a tuple:

- **Direct Assignment/manually, with normal parentheses ()**

```
my_tuple = (1, "Apple", 3.14, False, 5, 5)
```

- **Without parenthesis (just commas, Python automatically stores grouped values as tuples)**

```
my_tuple = 10, "Apple", 3.14, False, 5, 5
```

- **Single element tuple**

```
single_item_tuple = (1,)  #it correctly generates a single item tuple
single_item_tuple = 1,    #also correct without parenthesis
not_a_tuple =(1)   #generates a variable with 1 element only
print(not_a_tuple)
```

- **Using tuple contructor**

```
#From an iterable, like a list
mytuple = tuple(list(range(1,11)))
print(mytuple)

mytuple1 = tuple([1,2,3,4,5,6,7,8,9,10])
print(mytuple1)

#From a range
tpl = tuple(range(1,11))
print(mytuple2)

# From a string (splits into character
str_tuple = tuple("hello")
print(str_tuple)
```

- **Tuple unpacking, from an existing tuple**

```
my_tuple = (1, 2, 3)
a, b, c = my_tuple  # Unpacking
```

3.3.3) Operations with tuples: (my_tuple = (1, 'Apple, 3.14, 4 True0))
Given the immutability of tuples (no add, remove or change), many of the operations that is possible to perform with list will raise an error if used for tuples: for example, **.append()**, **.insert()**, **.remove()**, **.pop()** and **item reassignment** like tpl[0] = x all **raise an error**. Therefore, tupleshave fewer methods

```
tpl = (1, 2, 3)
# t[0] = 4       # ❌ Error: 'tuple' object does not support item assignment
# t.append(4)    # ❌ Error: 'tuple' object has no attribute 'append'
```

- **Indexing (accessing characters by position) (my_tuple = (1, "Apple", 3.14, False, 5)**

```
print(my_tuple[1])  # 'Apple'
```

- **Slicing (extracting subtuples)**

```
My_tuple = (1, "Apple", 3.14, False))


print(my_tuple[1:4])  # ("Apple", 3.14, False)
print(my_tuple[:3])    # (1, "apple", 3.14)
print(my_tuple[::2])   # (1, 3.14, 5)
```

- **Length check with len()**

```
print(len(mytuple)) #5
```

- **Membership check with in**
- **Counting occurrence of x**

```
tpl = (1, 2, 2, 3)
print(t.count(2))   # 2
print(t.index(2))   # 1
```

- **Returning the indes of first occurrence of x**

```
tpl = (1, 2, 2, 3)
print(t.index(2))   # 1
```

- **Hashability (can be used as key for dictionary or set elements)**

```
my_dict = {(1, 2): "value"}  # Valid dictionary key
print(my_dict[(1, 2)])       # "value"
```

3.3.4) Tuple unpacking

Tuple unpacking is an important charatteristic of tuples: it means **assigning the elements of a tuple** to **multiple variables in one step**

```
# Example tuple
person = ("Tyron", 30, "Engineer")

# Tuple unpacking into separate variables
name, age, profession = person

print(name)       # Tyron
print(age)        # 30
print(profession) # Engineer


###

mytuple = (1,2,5,6,7)

a,b,c,d,e = mytuple
print(a,b,c,d,e)
```

- The **number of variables** must **match** the **number of items** in the tuple.
- We can use `*` **to capture remaining items** (advanced use).

```
data = (1, 2, 3, 4, 5)
first, *middle, last = data

print(first)   # 1
print(middle)  # [2, 3, 4]
print(last)    # 5
```

4) MAPS

4.1) Dictionaries

A **dictionary** is an **unordered**, **mutable** data container that stores **key-value pairs**, where **keys must be unique and immutable**, but **values can be of any data type**, including mixed types.
Since dictionaries are unordered mapping, **slicing is not supported**

- **Unordered**: Items are **not accessed by position**, but by **keys**. *(Note: Since Python 3.7, insertion order is preserved, but conceptually you access by keys.)*
- **Mutable**: one can **add, remove, or update** key-value pairs **in place** without creating a new object.
- **Keys Must Be Unique**: Duplicate keys **are not allowed**; adding the same key again **overwrites the previous value**.
- **Values Can Repeat**: Different keys **can have the same value**.

4.1.1) Dictionary purpose:

Store **key-value pairs** with **unique keys** and **any type of values**. Ideal for **mapping relationships** like **names to phone numbers** or **field-value pairs**.

4.1.2) creation of a dictionary:

- **`Direct Assignment/manually:`** with curly braces {}

```
my_dict = {"name": "Tyron", "age": 30, "job": "Engineer"}
```

Output:

```
{'name': 'Tyron', 'age': 30, 'job': 'Engineer'}
```

- **`Dict():`** creates a dictionary with variable = key and data = value

```
my_dict = dict(name="Tyron", age=30, job="Engineer")
```

Output:

```
{'name': 'Tyron', 'age': 30, 'job': 'Engineer'}
```

- **`dict(list_of_tuples):`** creates a dictionary {key: value} = (value 1, value 2) of a tuple

```
pair_list = [(1,1), (2,4), (3,9), (4,16), (5,25), (6,36), (7,49), (8,64), (9,81), (10,100)]
mydic = dict(pair_list)
print(mydic)
```

Output:

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

- **`.zip(keys,values):`** merges two lists (keys and values) into a dictionary

```
keys = [1,2,3,4,5,6,7,8,9,10]
values = [1,2,9,16,25,36,49,64,81,100]

mydict = dict(zip(keys,values))
print(mydict)
```

- `dictionary comprehension:`

```
my_dict = {x: x**2 for x in range(1,11)}
print(my_dict)

###
#Inverrting keys and values from an existinf dictionary
new_dict = {mydict[key]:key for key in mydict}
print(new_dict)
```

4.1.3) Operations with dictionary:

- **Indexing (my_dict = {“name”: “Tyron”, “age”: 30, “job”: “Engineer”}**

```
print(my_dict["name"])  # "Tyron" NOTE: will raise an error if key doesn't exist

#Safer option with .get()
print(my_dict.get("name"))  # "Tyron"
print(my_dict.get("height", "Not found"))  # "Not found"
```

- **Adding or updating a key-value pair**

```
my_dict["country"] = "Netherlands"  # Adds new pair
my_dict["age"] = 31                 # Updates existing key
```

- **Removing key-value pair**

```
del my_dict["job"]            # Remove by key
value = my_dict.pop("age")      # Remove and return value
```

- **Checking if key exists**

```
if "name" in my_dict:
    print("Key exists!")
```

- **Getting all keys, values, items**

```
print(my_dict.keys())    # dict_keys(['name', 'country'])
print(my_dict.values())  # dict_values(['Tyron', 'Netherlands'])
print(my_dict.items())   # dict_items([('name', 'Tyron'), ('country', 'Netherlands')])
```

- **Looping thorugh dictionary**

```
for key in my_dict:
    print(key)          # Prints the key (e.g., "name", then "age")
    print(my_dict[key])  # Prints the corresponding value ("Tyron", 30, “Engineer”)
```

- **Looping through items**

```
#Allows to loop through both keys and values
for key, value in my_dict.items():
print(f"{key}: {value}")
```

5) SETS

5.1) Set

A **set** is an **unordered**, **mutable** data container that stores **unique elements** of any data type.
Sets are **useful for membership testing, removing duplicates**, and **mathematical set operations** like union and intersection.

- **Unordered**: Items have **no guaranteed order**, and **cannot be accessed by index**.
- **Mutable**: You can **add or remove items in place**.
- **Unique Elements Only**: **Duplicates are not allowed**. Adding the same item again **has no effect**.

NOTE: Sets don’t suppored indexing, therefore slicing, since they are unordered

5.1.1) Set Purpose:

Store an **unordered collection of unique items** with **no duplicates allowed**. Ideal for **membership tests**, **de-duplication**, or **mathematical set operations**.

5.1.2) creation of a set:

- **Direct Assignment/manually, with curly braces {}**

```
myset = {1,2,3,4,5,6,7,8,9,10}
print(myset)
```

- **Using the set() constructor**

```
my_set = set(range(1,11))

#from a list
my_set_list = set(list(range(1,11)))

myset_list = set([1,2,3,4,5,6,7,8,9,10])

#from a tuple
my_set_tpl = set(tuple(range(1,11)))

myset_tpl  = set((1,2,3,4,5,6,7,8,9,10))

#from a string
myset_str = set("Hello") #{H,e,l,l,o}
```

- **Set comprehension (with filtering)**

```
new_set = {x for x in my_set if x % 2 == 0}
```

5.1.3) Operations with sets:

- **Adding elements**

```
my_set.add(4)
```

- **Removing elements**

```
my_set.remove (2)   # Raises error if not found
print(my_set)      # {1,3,4,5,6,7,8,9,10}

my_set.discard(2)  # No error if not found
print(my_set)      # {1, 3, 4, 5, 6, 7, 8, 9, 10}



my_set.pop()      # Removes a random element
print(my_set)     # {1, 3, 4, 5, 6, , 8, 9, 10}

my_set.clear()    # Empties the set
print(my_set)     # set()
```

- **Membership testing**

```
if 3 in my_set:
   print("3 is in the set")
```

- **Looping through a set**

```
for item in myset:
   print(item)
```

5.1.4) Set operators

Set operators are **special methods and symbols** that perform **mathematical set operations**, such as **combining, comparing, or filtering** sets based on their elements.

- **Union**: Combines both sets by **adding all elements**, but since sets only store **unique elements**, it **keeps one instance of each distinct element**, even if they appear in both sets.
- **Intersection**: **Keeps only the elements that are common to both sets**, storing **a single instance** of each shared element.
- **Difference**: **Keeps only the elements that are in set a but not in set b**, excluding all elements that also appear in b.
- **Symmetric Difference**: **Keeps the elements that are unique to either a or b, excluding all elements that are common to both**.

NOTE: Only work with sets. To perform this operation on lists and tuple we got to convert them to lists

```
s = set(range(1,6))
s1 = set(range(0,11,2))
print(s)
print(s1)

s_total = s | s1  #combines all unique elements from both sets
print(f"union is {s_total}")

s_intersection = s & s1  #keeps only the elements tha exist in both sets
print(f"the interection is{s_intersection}")

s_difference = s-s1  #Keeps elements in a but not in b
```

```
s_difference1 = s.difference(s1)
print(f"difference is{s_difference}")
#print(f"difference is{s_difference1}")


s_sym_difference = s ^ s1  #Keeps elements in either a or b, but not in both
s_sym_difference1 = s.symmetric_difference(s1)
print(f"the symmetric difference is {s_sym_difference}")
#print(f"the symmetric difference is {s_sym_difference1}")
```

5.2) frozensets

A **frozenset** is an **immutable version of a set**, meaning its **elements cannot be added, removed, or changed** after creation.
It **stores unique, unordered elements**, just like a normal set.

- **Unordered**: No guaranteed order, no indexing or slicing.
- **Immutable**: **Cannot** add, remove, or change elements after creation.
- **Unique Elements Only**: **No duplicates allowed**.

5.2.1) Key uses:

- **Hashable**: Can be used as **keys in dictionaries** or **elements of sets**.
- **Safe to share**: Guarantees the content **cannot change**.

```
my_f_set = frozenset(range(1,11)) # it’s like a tuple for sets
print(my_f_set)
```

LOGICAL OPERATORES

Logical operators are simple but fundamental objects in Python

- Work with **boolean values** (`True`, `False`).
- Often used in **`if` statements** and **conditionals**.
- Evaluate **left to right**, with **`not`** having **highest precedence**.

1) AND

- Returns `True` **only if both** conditions are **True**.
- Example: `True and True` → `True`; `True and False` → `False`.

```
x = 7
print(x > 5 and x < 10)  # True
```

2) OR

- Returns `True` **if at least one** of the conditions is **True**.
- Example: `True or False` → `True`; `False or False` → `False`.

```
x = 7
print(x < 5 or x > 10)   # Fals
```

3) NOT

- **Reverses** the result of the condition.
- Example: `not True` → `False`; `not False` → `True`.

```
x = 7
print(not x > 5)         # False
```

CONDITIONAL STATEMENTS: if, elif and else

if, **elif**, and **else** are **conditional statements** used to control the **flow of a program** based on whether conditions are **True or False**.

- **if** checks the first condition.
- **elif** ("else if") checks additional conditions if the previous ones were False.
- else runs if **none of the previous conditions** were True.

They are often use inside loops, where at every iteration they check for a condition

```
x = 10


if x > 10:
   print("Greater than 10")
elif x == 10:
   print("Exactly 10")
else:
   print("Less than 10")
```

Nested conditional statements
Sometimes, an if, elif, or else block may contain **another conditional structure inside it**.
This is called **nesting**, and it’s used in situations where, once a condition is met, it **leads to its own set of if, elif, or else statements** —
in other words, when a **second decision** depends on the **first one being true**.

```
age = 18
has_id = True


if age >= 18:
   if has_id:
       print("Access granted.")
   else:
       print("Access denied. Please show ID.")
else:
   print("You must be at least 18.")
```

CYCLES: For and while loop

In python, cycles (commonly called loops) are structures that repeat/iterate a certain block of code until a condition is met or all items in a sequence are processed.
Inside a single cycle there might be additional conditions to be verified at every iteration (if-else statements) which we are going to discuss in the next chapter.

Break, Continue, Pass

These are **three control flow keywords** used inside loops,each with a **different effect on how the loop behaves**.
They are **often used inside if, elif, or else blocks**, so that when a condition is met, the keyword takes effect — but **they do not require** an **if** to be used.

- **`Break`** is used to immediately stop the current loop**,** skipping any remaining iterations, and continuing with the first statement after the loop.
  In the case of nested loops, break only stops the innermost loop—that is, the closest enclosing loop where the break appears. The outer loop continues as normal unless explicitly broken as well.

```
for i in range(2):              # Outer loop: i = 0, 1
    print(f"Outer loop i = {i}")

    for j in range(5):          # Inner loop: j = 0 to 4
        if j == 2:
            print("  Breaking inner loop")
            break               # Exits only the inner loop
        print(f"  Inner loop j = {j}")

    print("End of outer iteration\n")
```

Output:

```
Outer loop i = 0
  Inner loop j = 0
  Inner loop j = 1
  Inner loop j = 2 => Breaking inner loop
Going to the next outer iteration

Outer loop i = 1
  Inner loop j = 0
```

```
 Inner loop j = 1
 Inner loop j = 2 => Breaking inner loop
Going to the next outer iteration
```

- **`Continue:`** skips the rest of the current iteration and moves to the next one, without stopping the loop entirely. Outer loops in case of nested loops are not really affected as this command only affects the one it's written in— the outer loop continues as usual, unless explicitly told otherwise.

```
for x in range(5):
    if x == 2:
        continue  # Skip this iteration when x is 2
    print(f"x = {x}")
```

Output

```
x = 0
x = 1
x = 3
x = 4
```

- **`Pass:`** does nothing; acts as a placeholder when a statement is required syntactically but no action is needed yet.

```
for x in range(5):
    if x == 2:
        pass  # Do nothing here, but continue as usual
    print(f"x = {x}")
```

Output

```
x = 0
x = 1
x = 2
x = 3
x = 4
```

For loop

This type of loop is used when we want to perform **finite and controlled iterations**.
A **'for' loop** repeats over a **fixed sequence** (like a list, range, or set) with a **known number of iterations**. In fact, **'for' loops** iterate over **collections** like **lists, tuples, dictionaries, or sets**,which are, by definition, **objects with a finite and known number of elements**.

'For' loop with iterations given by a specific number

- **Over a certain number (direct iteration).**

```
for x in range(1,6):            # 5 iterations, print the “*” 5 times
    print("*","*","*","*","*")
```

```
for x in range(6):
    print(x)                # 0,1,2,3,4,5 (in a column)
```

Most common iterations for lists:

- **`Direct iteration:`** here x represents the items, not the indices (value-based iteration)

```
my_list = [1, 'Apple', 3.14, 4, True]

for x in my_list:
    print(x)          # 1, Apple, 3.14, 4, True (in a column)
```

- **`Iterate over pairs indices/values of a list`**

```
fruit =[]

for i,x in enumerate(my_list):
    if my_list[i] == "Apple" or my_list[i] == "kiwi": #my_list[i] = value x at index i
        fruit.append(x)
print(fruit)          # prints the final, complete list

###

new_list =[]

for i,x in enumerate(my_list):
    if i % 2 == 0 :              # if index i is even, then append x
        new_list.append(x)
    print(new_list)              # prints new_list at every iteration
```

- **`index-based iteration:`** iterate over indices

```
for i in range(len(my_list)):
    if i >= 3:                   # if index >= 3
        print(my_list[i])          # print the value = my_list[i] for every index

###

# in this case, the presence of i + 1 will make the iteration go out of range when I = last element. To avoid this, the – 1
form is used so that i + 1 - 1 = i
my_list = ["a", "b", "c"]

for i in range(len(my_list) - 1):  # Runs for i = 0, 1 only
    print(my_list[i], my_list[i + 1])
```

Most common iterations for string:
Same iteration structure as lists

Most common iterations for tuples:
Same iteration structure as lists

Most common iterations for dictionaries:

- **Iteration over keys (default)**

```
for key in my_dict:
    print(key)
    print(my_dict[key])  #prints the value
```

- **Iteration over values**

```
for value in my_dict.values():
    print(value)
```

- **Iterations over pairs key/value**

```
for key, value in my_dict.items():
    print(key, value)
```

Most common iterations for set

- **Iteration over items**

```
for item in my_set:
    print(item)
```

Nested 'for' loop: in this situation, for every single iteration of the first loop, python is gonna run all the iteration of the nested loop (recursively depending on how many for are nested). The two, or more, loops can be completely unrelated or dependant on each other.

```
# related nested for loops
my_list = [(1,2), (3,4), (5,6)]
new_list =[]

for tpl in my_list:   #this for goes through every tuple in my_list
    for item in tpl:  #this for goes through every element of every tuple
         new_list.append(item)
        print(item)   # prints every item
    print(tpl)        # prints every tuple
print(new_list)       # prints the total new_list

# unrelated nested for loops
fruits = ["apple", "banana"]
colors = ["red", "green", "yellow"]

for fruit in fruits:  # Outer loop over fruits
    print(f"Fruit: {fruit}")
    for color in colors:  # Inner loop over colors
        print(f" - Color option: {color}")


Output


Fruit: apple
 - Color option: red
 - Color option: green
 - Color option: yellow
Fruit: banana
 - Color option: red
 - Color option: green
 - Color option: yellow
```

Else in for loops…

While

When we want to itearate until a certain condition is met but we don’t necessarely need to keep track of the numbers or step of iterations or using a predefined sequence.

A **while loop** repeats **as long as a condition stays True**, without needing a container or predefined sequence.

An important diffrence between ‘while’ and ‘for’ loop is that **in a while loop**, we **need to manually control the iteration** at each step by **updating the variable ourselves**.

For this reason, **augmented assignment operators** like += or −= are **commonly used** to **increment or decrement** the variable and ensure the loop eventually stops.

Most common ways to start and stop a while loop:

- **With a boolean condition/comparison operator**

```
x = 0
while x < 5: #atutomatically stops when x=5, including this value
    print(x)
    x += 1   #increases x (that starts from 0) by one at every iterations
```

- **Boolean flag (variable = True ; variable = False)**

```
running = True
    user_input = input("Type 'stop' to exit: ")
    if user_input == "stop": # when the if condition is triggered (input = “stop”)
        running = False      # running is evaluated to False and the loop stops
```

- **`break:`**

```
while True:
    user_input = input("Type 'exit' to stop: ")
    if user_input == "exit":
        break
```

- `return` (in functions)

```
def ask_until_yes():
    while True:
        answer = input("Say 'yes' to continue: ")
        if answer == "yes":
            return  # Exits the loop and the function
```

- `exit():` terminates the entire Python program immediately, no matter where it is called.

```
while True:
    user_input = input("Type 'exit' to terminate the program: ")
    if user_input.lower() == "exit":
        print("Exiting the program...")
        exit()
    else:
        print(f"You typed: {user_input}") # this code is unreacheable
```

BUILT-IN ATTRIBUTES, METHODS AND FUNCTIONS

The difference between a function and a method is **mostly conceptual and syntactic**, not functional — especially when you're using pre-built libraries like NumPy or pandas.

Functions and methods are **built-in tools provided by Python** that let us perform common tasks **without writing the logic from scratch**.

Attributes…

Functions

A **function** is a **named block of reusable code** that performs an action or returns a result.
It is **called using its name**, followed by parentheses.
Examples: `print()`, `len()`, `type()`, `range()`

A function is indipendent and applies to values we give it:

- A **function** exists **on its own**, not tied to a specific object.
- You pass the **data into it** as an **argument**.
- It works **externally** on whatever you give it.

```
name = "Tyron"
print(len(name))     # len() is a function → you pass 'name' into it
```

Most common functions:

- **The range() function creates an iterable sequnce of numbers based on the input parameters**

```
range(1,6)  #creates the seguence 1,2,3,4,5
range(5)    #creates the sequence 0,1,2,3,4
```

Methods

A **method** is a **function that is attached to an object** and **acts on that object**. It is **called with dot notation**: `object.method()`.
Examples: `"hello".upper()`, `my_list.append(4)`, `my_dict.get("key")`

A method is dependent on the object is called on:

- A **method belongs to a specific object type** (like strings, lists, or dictionaries).
- It's called **using dot notation**, and it **knows the object it's working on** because it's already attached to it.
- You don't pass the object in — it's **implicitly understood**.

```
name = "Tyron"
print(name.upper())  # .upper() is a method → called *on* the string object
```

Most common methods:

BUILDING A FUNCTION

- **Reusability**: Write code once, use it multiple times without repeating logic.
- **Clarity and readability**: Break complex code into smaller, named blocks that are easier to understand.
- **Avoid repetition**: Follow the DRY principle ("Don't Repeat Yourself").
- **Modularity**: Organize code into logical units, making it easier to structure and collaborate on.
- **Easier testing and debugging**: Test individual functions in isolation to find and fix issues quickly.
- **Maintainability**: Update functionality in one place instead of tracking down repeated code.
- **Flexibility through parameters**: Make functions dynamic by accepting different inputs and returning results.

How to build a function

First of all we use the **def keyword**, which tells python that we are about to build a function, followed by the name of the function and round brackets. Inside the brakets, we are going put the variables, that at this point are called parameters, that the function.

```
def greet(name):
```

After this, and at one level of indentation, we have the body of the function, which is the code we want to run every time we call the function: this code can be anything.

```
print(f"Hello, {name}!")
```

At this point, whenever we need to use the function we can call it with the related parameters, that are noiw called arguments of the function.

```
greet("Tyron")  # Calling the function with an argument
```

Output:
Hello, Tyron!

Defining parameters during definition:

- **Positional parameters**: `def func(a, b)`
- **Default values**: `def func(a, b=10)` → `b` is optional
- **Variable-length positional args**: `def func(*args)` → accepts any number of arguments
- **Keyword arguments / kwargs**: `def func(**kwargs)` → accepts named arguments as a dictionary

Passing arguments during function call:

- **Positionally**: `func(1, 2)`
- **By keyword**: `func(a=1, b=2)`
- **Mix of both** (positional first): `func(1, b=2)`
- **Unpacking sequences or dictionaries**:
  - `func(*[1, 2])`
  - `func(**{"a": 1, "b": 2})`

return command: the return statement is used to **exit a function** and optionally **send back a result** to the caller.
When return is executed, the function **immediately ends**, and any code **after it is ignored**.
It can return **any value** — a number, string, list, boolean, or even another function.

Example of a function:

```
def find_even(numbers):
  for num in numbers:
     if num % 2 == 0:
            return f"First even number found: {num}"
    else:
        return "No even number found in the list."

result = find_even([1, 3, 5, 8, 9])
print(result)
# Output: First even number found: 8
```

MODULES
OOP (Object Oriented Programming)

NUMPY

NumPy is a powerful mathematical library in Python that provides support for **efficient operations on arrays, vectors, and matrices**.
It allows for **fast numerical computations** and includes a wide range of **functions for linear algebra, statistics, and other mathematical operations** on these data structures.
It's also very useful for deep learning and data science in general!

The first step is to install and import the library

```
!pip install numpy
```

An import it

```
import numpy as np
```

Array in Numpy

An **array** is the **basic building block** in NumPy -- a grid-like data structure used to store elements of the same type. **This is the native Numpy's data container**.
It supports efficient mathematical operations and broadcasting, making it ideal for storing and manipulating numerical data for scientific computing across one or more dimensions.
Arrays can be 1D, 2D, or multi-dimensional.

Dimension of an array

The dimension (also rank or ndim) of a Numpy array refers to the number of axis it has. We can picture the axis as normal cartesian axis with **`x = rows, y = columns and z = depth/layer`**:

- A **0D array** is a scalar — just a single value with no axes.
- A **1D array** (e.g. shape `(x,)`) is a vector, with one axis (like a line).
- A **2D array** (e.g. shape `(x, y)`) is a matrix, with two axes: rows and columns.
- A **3D array** (e.g. shape `(x, y, z)`) is a stack of matrices — like a cube — with three axes: height, width, and depth.

The number of dimensions is also determined by the level of **nesting** in the original list structure. For example, `np.array([[1, 2, 3, 4, 5]])` is a **2D array** with shape `(1, 5)` because it contains **one outer list** (defining one row), which itself contains **a list of five elements** (the columns).
Even though it has only one row, it still requires **two indices** to access a value — one for the row and one for the column — which is what makes it 2D. We can think of it like a **table/matrix with 1 row and 5 columns,** just like a spreadsheet.

NOTE: every dimension has it's own size. A 1D array has shape `(n,)` — one dimension with `n` elements. A 2D array (matrix) has shape `(rows, columns)`, where axis 0 is rows and axis 1 is columns. `.shape` gives sizes per axis; `.ndim` gives the number of dimensions.

Creating a 1D array from a list

```
#create arrays with Numpy

#create a 1D array

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

```
arr1 = np.array[to_be_arr]


#or


arr2 = np.array([1,2,3,4,5])
print(arr1)
print(type(arr1))
print(arr1.shape)
```

Output:

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
(5,) (this is the shape)
```

We can see that a **NumPy array is not the same as a Python list** — for example, when printed, the array **doesn't show commas** between elements, and its **type is explicitly `numpy.ndarray`**.
Additionally, the output `(5,)` refers to the array's **shape**:

- The **single number `5`** means it is a **1D array** with **5 elements**.
- The **comma** after the `5` is part of Python's syntax for a **tuple**, indicating that the shape is `(5,)` and **not just a scalar**. 5 indicates the number of elements inside the array

NOTE: this is not a 1 row 5 columns objects, since is 1D, it's just an array with 5 elements.

Reshaping the first array into a 2D object (from a list)

```
# 1D array
arr2 = np.array([1,2,3,4,5])
arr2.reshape(1,5) # the '1,5' means 1 row and 5 columns
```

Output:

```
array([[1, 2, 3, 4, 5]])
```

The two sets of square brackets in the output mean that now this is a 2D array (a 3D object would have 3 sets of square brackets). **The number inside the reshape function has to match the number of elements in the original array** : if the array is a 1x5 object (1 row, 5 elements), the reshape as to be .reshape (1,5)

Creating a 2D array from the original array by nesting the original list

```
# .array() converts a list (elements separated by commas) into an array(elements not separated
# by commas) . It works with list created with a COMPREHENSION as well!
arr2 = np.array([[1,2,3,4,5]])
print(arr2)
```

Output:

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

If we check the shape of the previous array we'll obtain
(1,5)

Meaning that the array has **1 row and 5 columns**, unlike the 1D array which has shape `(5,)`, the **presence of both dimensions** (no blank after the comma) confirms that this is a **2D object** — essentially a row vector.

Creating a 2D array through a double list

```
# 2D array
arr2 = np.array([[1,2,3,4,5],[2,3,4,5,6]])
print(arr2)
print(arr2.shape)
```

Output:

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

This means that there are **2 rows and 5 columns**.
It is still considered a **2D array** because we have **two parallel lists** (each representing a row), both **nested inside a single outer list** — which gives us **two levels of structure**, i.e., a **2D array.**

Creating an array with the built-in function

- **`.array(list)`:** creates a 1D array from a list

  ```
  array = np.array(list(range(0,6)))
  print(array)
  ```

  Otuput:

  ```
  [0 1 2 3 4 5]
  ```

- **`.arange(start, end, stepsize)`:** creates an array with the elements sorted

  ```
  # 1D array with built-in function arange
  np.arange(0,10,2)
  ```

  Otuput:

  ```
  array([0, 2, 4, 6, 8])
  ```

- **`.reshape(# of rows, # of columns)`:** creates a 1D array by reshaping with

  ```
  np.arange(0,10).reshape(10,) # (10,) means a 1D array with 10 elements
  ```

  Otuput:

  ```
  array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
  ```

- **`.reshape(# of rows =1D, # of columns =2D)`:** creates a 2D array by reshaping

  ```
  np.arange(0,10,2).reshape(1,5) # (1,5) means 1 row 5 columns
  ```

  Output:

  ```
  array([[0, 2, 4, 6, 8]])
  ```

  ```
  np.arange(0,10,2).reshape(5,1) # (5,1) means 5 rows 1 column
  ```

  Output:

```
array([[0],
       [2],
       [4],
       [6],
       [8]])
```

Note: `.reshape(...)` changes the shape of an array to the desired number of dimensions by specifying the number of elements along each axis — such as rows, columns, or depth.
The **number of values inside `.reshape()` defines the number of dimensions**, while the **values themselves define how many elements to include along each axis**

- **`.randint(start, end, size = (x, y)):`** creates a 2D array with random integers

```
arr3 = np.random.randint(1,21, size = (5,5))
print(arr3)
```

Output:

```
[[ 4  4 14 18 12]
 [20  7 18  8  2]
 [ 6 14 14 16  1]
 [ 5  2 11 10 16]
 [18 13 18 14 16]]
```

NOTE: numpy has it’s own randint method thorugh the random sub-module

- **`List comprehension:`** creating a 1D array (vector)

```
array = np.array([np.random.randint(1,20) for x in range(5)])
print(array)
```

Output:

```
[18  8  1 18  6]
```

- **`List comprehension:`** creating a 2D array (matrix)

```
array = np.array([[x for x in range(4)] for y in range(4)])
print(array)
```

Output:

```
[[0 1 2 3]
 [0 1 2 3]
 [0 1 2 3]
 [0 1 2 3]]
```

Built-in functions and methods in Numpy

Let’s take a look at the most useful built-in functions in Numpy

- **`.concatenates(([,],[,])):`** concatenates elements from previous array into a new one

```
border_elements2 = np.concatenate((arr1[0,:], arr1[-1,:],arr1[1:-1,0], arr1[1:-1,-1]))
print(border_elements2)
```

Output:

```
These are the border elements
[ 2 10  9  5  4  2 11  7  6 10]
```

- **`.sum(array, axis)`**: sums all elements in a row or in a column

```
# Compute the row-wise and column-wise sum
row_sum = np.sum(array, axis=1)
print('this the the row sum:', row_sum)
column_sum = np.sum(array, axis=0)
print('this the the column sum:',column_sum)
```

Output:

```
this the the row sum: [10 26 42 58]
this the the column sum: [28 32 36 40]
```

- **`.prod(array)`**: mutliplies element sin an array (or list)

```
arr = np.random.randint(1,5, size = (2,2))
np.prod(arr)
```

Output:

```
[[4 1]
 [4 4]]
np.int64(64)
```

- **`.ndim(array)`**: retrieves the dimension of an array with .ndim(array)

```
array = np.array([rd.randint(1,20) for x in range(5)])
print(array)
np.ndim(array)
```

Output:

```
[16  6  5 20 17]
1
```

- **`.size(array)`**: retrieving the number of elements of an array

```
arr = np.random.randint(1,5, size = (2,2))
print(arr,'\n')
np.size(arr)
```

Output:

```
[[4 1]
 [4 4]]

4
```

- **`.reshape(new shape x, new shape y)`**: reshapes an array into a new one

```
arr = np.arange(1,10).reshape(3,3)

reshaped_arr1 = arr.reshape(9,1)          #reshaping to (1,9) array (1 row, 9 columns
print(reshaped_arr1
reshaped_arr2 = reshaped_arr1.reshape(1,9) #reshaping to (9,1) array (9 row, 1columns)
print(reshaped_arr2)
```

Output:

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

- **`.flatten(order)`** : flattening an array into a vector with .flatten(order = number)

```
arr = np.random.randint(1, 10, size = (5,5))
print(arr)

flat_arr = arr.flatten()
print(flat_arr)
```

Output:

```
[[5 9 4]
 [3 1 1]
 [4 6 7]]

[5 9 4 3 1 1 4 6 7]
```

NOTE: the order in flatten allows to decide the output of the array. Here two examples:

- **Order = ‘C’:** reshapes the orignal array by reading it row-wise (1st row, 2nd row, etc.)
- **Order = ‘F’:** reshapes the original array by reading it column wise (1st col, 2nd col, etc.)

Attributes of array

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print("Array:\n", arr)
```

Output:

```
Array:[[1 2 3]
        [4 5 6]]
```

- **`.shape`**: retruns the shape of the array

```
print("Shape:", arr.shape)
```

Output:

```
Shape: (2, 3)
```

- **`.ndim:`** returns the dimension od the array

```
print("Number of dimensions:", arr.ndim)
```

Output:

```
Number of dimensions: 2
```

- **`.size:`** returns the number of elements of an array

```
print("Size (number of elements):", arr.size)
```

Output:

```
Size (number of elements): 6
```

- **`.dtype:`** returns data type (int32 but it may vary base on the platform)

```
print("Data type:", arr.dtype)
```

Output:

```
Data type: int32
```

- **`.itemsize:`** returns the size of bytes of the element in the array

```
print("Item size (in bytes):", arr.itemsize)
```

Output:

```
Item size (in bytes): 4
```

Vectorized operations in Numpy

These operations and functions operate on **each element of an array without writing explicit loops**, making your code faster and cleaner

```
arr1=np.array([1,2,3,4,5])
arr2=np.array([10,20,30,40,50])
```

Output:

```
[1 2 3 4 5]
[10 20 30 40 50]
```

- **Simple mathematical operations (element wise)**

```

Element-Wise addition
print("Addition:", arr1+arr2) #1+10, 2+20, etc

Element-Wise Substraction
print("Substraction:", arr1-arr2) #1-0, 2-20, etc

```

```
Element-wise multiplication
print("Multiplication:", arr1 * arr2) #1*10, 2*20, etc

Element-wise division
print("Division:", arr1 / arr2) #1/10, 2/20, etc
```

Output:

```
Addition: [11 22 33 44 55]

Substraction: [ -9 -18 -27 -36 -45]

Multiplication: [ 10  40  90 160 250]

Division: [0.1 0.1 0.1 0.1 0.1]
```

- **`Universal functions (ufuncs)`:** functions that apply to the entire array

```
arr=np.array([2,3,4,5,6])

square root
print(np.sqrt(arr)) # calculates the root square of each element in the array

Exponential
print(np.exp(arr)) # calculates e^x (x=element) the of each element in the array

Sin
print(np.sin(arr)) # calculates the sin of each element in the array

Natural log
print(np.log(arr)) #calculates log of each element in the array
```

Matricial operations in Numpy

```
matrix = np.random.randint(1, 11, size=(3, 3))
print("Original matrix:")
print(matrix)
```

- **`.ones(i, j)`:** creates i, j dimension matrix with all 1

```
np.ones((3,4))  #array with 3 rows and 4 columns with all 1
```

Output:

```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

- **`.eye(N)`:** creates N dimension identity matrices

```
np.eye(3) # 3 is the dimension of the matrix
```

Output:

```
array([[1., 0., 0.],
       [0., 1., 0.],
        [0., 0., 1.]])
```

- `.det()`: retrieves the determinant of a matrix

  ```
  determinant = np.linalg.det(matrix)
  print("Determinant:", determinant)
  ```

- `.inv()`: retrieves the inverse of a matrix

  ```
  inverse = np.linalg.inv(matrix)
  print(inverse)
  ```

- `.eigvals()`:retrieves the eigenvalues of a matrix

  ```
  eigenvalues = np.linalg.eigvals(matrix)
  print(eigenvalues)
  ```

Array indexing and slicing

Indexing works as usual with square brackets next to the variable; rows are indexed in the square bracket before the comma and columns are indexed after the comma (`[1:2 = rows, 2:3 = columns]`). Indeces start from 0 as usual.

The the double bracket syntax (`[1:2][2:3]`) doesn't work here!

The stepsize is the last element (`[x:x:stepsize, y:y:stepsize]`)

```
arr=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]]) # 2D array with 3 rows and 4 columns
print("Array : \n", arr)
```

Output:

```
Array :
 [[ 1  2  3  4]
  [ 5  6  7  8]
  [ 9 10 11 12]]
```

- **`Selecting rows:`**

  ```
  arr [0] # selects the first row
  ```

  Output:

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

- **`Selecting an element:`** from a row and then from a column

  ```
  print(arr[0][1]) # selects the first element of the second column
  ```

  Output:

  ```
  2
  ```

- **Selecting all the rows from a starting point:**

```
arr [1:] # taking all the rows from the second one (index 1) onward(:)
```

Output:

```
array([[ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

- **Selecting rows and columns from a staring to an ending point:**

```
print(arr[1:,1:3]) # taking elements from column 2 (,1) to column 3 not included, aka      column 2, (:3), from the
second row onward ([1:)
```

Output:

```
[[ 6  7]
 [10 11]]
```

NOTE:

- **Modifying array elements**

```
#substituting element from row 1 column 1 (aka the first element)
arr[0:1, 0:1]=100
print(arr)
```

Output:

```
[[100   2   3   4]
 [  5   6   7   8]
 [  9  10  11  12]]
```

```
#substituting all elements from row 2 ([1:) onward(:]) with 100
arr[1:]=100
print(arr)
```

Output:

```
[[  1   2   3   4]
 [100 100 100 100]
 [100 100 100 100]]
```

Statistical operators in Numpy (applied to the entire array)

- **Normalization (statistical concept)**

```
# to have a mean of 0 and standard deviation of 1
data = np.array([1, 2, 3, 4, 5])

# Calculate the mean and standard deviation
mean = np.mean(data)
```

```
std_dev = np.std(data)

# Normalize the data
normalized_data = (data - mean) / std_dev
print("Normalized data:", normalized_data)
```

Output:

```
Normalized data: [-1.41421356 -0.70710678  0.          0.70710678  1.41421356]
```

- **Statistical Functions**

```
data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Mean
mean = np.mean(data)
print("Mean:", mean)

# Median
median = np.median(data)
print("Median:", median)

# Standard deviation
std_dev = np.std(data)
print("Standard Deviation:", std_dev)

# Variance
variance = np.var(data)
print("Variance:", variance)
```

Output:

```
 Mean: 5.5
Median: 5.5
Standard Deviation: 2.8722813232690143
Variance: 8.25
```

NOTE: 1) Min-Max normalization

- Rescales values to fit within a defined range (commonly 0 to 1).
- Used to make different features (like age and income) comparable.
- Example: Transforming ages from `[18, 36, 72]` into `[0.0, 0.33, 1.0]`.

2) Z-score Normalization (standardization)

- Centers values around 0 with standard deviation 1.
- Useful when comparing features with different units and spreads.
- Example: Test scores `[50, 70, 90]` become something like `[-1, 0, 1]`.

3) L2 Normalization (Unit vector)

- Scales a vector so that its total length is 1.
- Used to preserve direction while standardizing magnitude.

- Example: Vector `[3, 4]` becomes `[0.6, 0.8]` — same proportions, smaller scale.

P.S.: these concepts will be discussed further in the statistical guide

Logical operators

```
# Logical operation
data=np.array([1,2,3,4,5,6,7,8,9,10])
```

- **`Comparing array with value (<, >, ==, !=, <=, >=):`**

```
data > 5 # Compares all the elements of the array with 5
```

Output:

```
array([False, False, False, False, False,  True,  True,  True,  True, True])
```

- **`Using & (= and):`**

```
# If we wanted to retrieve the elements that are >=5 and <=8 we should use indexing
data[(data >= 5) & (data <= 8)] # ’&’ here is ‘AND’
```

Output:

```
array([5, 6, 7, 8])
```

- **`Using | (= or):`**

```
# If we wanted to retrieve the elements that are >=5 or <=8 we should use indexing
data[(data >= 5) | (data <= 8)] # ‘|’ here is ‘OR’
```

Output:

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

Fancy and Boolean indexing

These are both powerful features in NumPy that make it easier to work with arrays.
Fancy indexing allows to retrieve specific elements from an array using lists or arrays of indices.
Boolean indexing is used to filter elements based on conditions, returning only those that meet the criteria, and it’s based on concept of “boolean mask”.
Boolean mask: A **boolean mask** is a NumPy array of `True` and `False` values used to **select or filter elements** from another array. `True` means "keep this element"; `False` means "ignore it."

```
arr = np.random.randint(1,26, size = (5,5))
```

Output:

```
[[ 6 13 14  6 17]
 [13  1 11  9  3]
 [ 3  3  6 15 15]
 [ 2 20 16 10 19]
 [13 13 10  2 11]]
```

- **`Fancy indexing:`** retrieving the corner elemtens of an array by using list of indices

```
corner = arr[[0, 0, 4, 4],[0, 4, 0, 4]]
print(corner)
```

Output:

```
[ 6 17 13 11]
```

- **`Boolean mask:`** filtering array and retrieving all element > 10

```
mask = arr > 10
filtered = arr[mask]
print(filtered)
```

Output:

```
[13 14 17 13 11 15 15 20 16 19 13 13 11]
```

- **`Boolean indexing:`** masking/replacing elements > 10 with the element 10

```
arr[arr > 10] = 10
print(arr)
```

Output:

```
[[ 6 10 10  6 10]
 [10  1 10  9  3]
 [ 3  3  6 10 10]
 [ 2 10 10 10 10]
 [10 10 10  2 10]]
```

Structured Arrays

A **structured array** in NumPy is a 1D array where each element is like a row in a table, and each row can contain **multiple named fields** of different data types (e.g., strings, ints, floats). It allows you to organize heterogeneous data in a structured way, similar to a **lightweight DataFrame** or a database record.

- **`Structured array:`**

```
data_type = [('x', 'i4'), ('y', 'i4')]
data = np.array([(1, 2), (3, 4), (5, 6)], dtype=data_type)
print("Original array:")
print(data)
```

Output:

```
[(2, 3) (4, 5) (6, 7)]
```

- **`Structured array:`** sorting elements in structured array by ascedent value

```
data_type = [('Name', 'U10'),('Age', 'i4'),('Weight', 'f4')]
data = np.array([('Tyron',37,79.3), ('Eline',34,57.8),('Jack',31,75)], dtype = data)
print(data)
```

```
sorted_data = np.sort(data, order = 'Age')
print(sorted_data)
```

Output:

```
Data: [('Tyron', 37, 79.3) ('Eline', 34, 57.8) ('Jack', 31, 75. )]
Sorted: [('Jack', 31, 75. ) ('Eline', 34, 57.8) ('Tyron', 37, 79.3)]
```

NOTE: the key-words 'U10', 'i4', 'f4' refers to the **data types used in NumPy structured arrays**, respectively: string with up to 10 chars; 4-byte integer; 4-byte floating-point number.

Masked Arrays

A **masked array** in NumPy is an array that allows you to **hide or ignore specific elements** during computations by using a **boolean mask**. The masked elements are treated as "missing" or "invalid," and NumPy functions automatically skip them in operations like `sum()`, `mean()`, etc.

```
arr1 = np.random.randint(1,21,size=(3,3))
print(arr1)
```

Output:

```
[[ 7  6 19]
 [ 4 14 19]
 [ 6  3  1]]
```

- **`Masked array:`** filtering elements > 10 and replace them with the su of the unmasked ones

```
filtered = arr1[arr1 > 10]
sum_filtered = sum(filtered)
print(sum(filtered))
```

Output:

```
52 #=19+14+19
```

- **`Masked array:`** masked elements on the diagonal and replace them with the mean of the ones (unmasked) on the daiagonal

```
import numpy.ma as ma

# Create a masked array of shape (3, 3) with random integers
masked_array = ma.masked_array(arr1, mask=np.eye(3, dtype=bool))
print("Original array:")
print(arr1)
print("Masked array:")
print(masked_array)

# Replace the masked elements with the mean of the unmasked elements
mean_unmasked = masked_array.mean()
masked_array = masked_array.filled(mean_unmasked)
print("Modified masked array:")
print(masked_array)
```

Output:

```
Original array:
[[ 9  6 19]
 [ 4  9 19]
 [ 6  3  9]]

Masked array:
[[-- 6 19]
 [4 -- 19]
 [6 3 --]]

Modified masked array:
[[ 9  6 19]
 [ 4  9 19]
 [ 6  3  9]]
```

PANDAS

Pandas is a powerful Python library used for **data manipulation and analysis.** It makes easier to **read, clean, filter, analyze, and export** data efficiently, supporting operations like indexing, grouping, and statistical analysis with just a few lines of code

The first step is importing installing the library

And install it

```
import Pandas
```

Then, of course we want to have a table to work on. We can therefore import the csv document and assign it to a variable (in this case, ‘data’)

data = pandas.read_csv(“name_of_the_file.csv”)

NOTE: the csv file we want to open must be in the same directory as the python script or the path to the directory with the file must be given.
Also, this process can be shortened by naming pandas with a shorter name when importing it

```
import pandas as pd
```

data = pd.read_csv(“name_of_the_file.csv”)

Pandas data type: DataFrame and Series

Dataframe and series are Pandas' native data containers.
We can now check for the type of the data

```
Print(type(data))
```

Output: <class 'pandas.core.frame.DataFrame'>

The two main types of data in a csv file in pandas are series and data_frames

1) Series

DEF: A Series is a 1D object (an array) which contains data and has an index for each data. If it comes from a dataframe, it corresponds to either a column or a row (see note)**.**
Pandas considers by default the first row of the series/column as the relative header.
We can access a series/column from a data_frame just like we would access a value in a dictionary by using the key

Dictionary: value = dict["key"] Data_frame: series = data["temp"]

with the difference that the key gives back one value in the form of a list or another dictionary or whatever value is stored by that key whereas the data_frame will always give back a series of values.

NOTE: For simplicity, we often say that a **Series** is a **column** (or a **row**) from a DataFrame.
In practice, a **Series** is simply a **1-dimensional labeled array** of values.

- When a Series **comes from a DataFrame**, it is indeed a **column** or a **row**.
- A **column or a row is always a Series**, but
- A **Series is not always** a column or row.

For example, a **custom combination of values** from different rows or columns can also be represented as a Series, as long as it has an index and a value for each entry.

Convert a series to a list

A series, given its 1D nature is comparable to a list. We can in fact convert a series to a list with the 'to_list()' method (this works only with series and conserves the original data type of the series):

```
data_list = data["temp"].to_list()
```

If we wanted to convert the entire data_frame to a list, we have two options:

1) Use the 'values' attribute (an array in NumPy) and the 'to_list()' method in conjunction.

This will convert all columns or all rows to list:

```
data.values.to_list()
```

Output:

```
['Amsterdam', 21], ['Paris', 25], ['Rome', 19],
```

2) Use the '.to_list('list')' to convert each column into a list inside a dictionary. Here list is a parameter that controls the format of the dictionary that is returned:

Output:

```
{'city': ['Amsterdam', 'Paris', 'Rome'], 'temp': [21, 25, 19]}
```

2) Data_frame:

DEF: a DataFrame is a collection of Series objects (often described as a dictionary of Series).
It is a **mutable**, **heterogeneous**, **two-dimensional** data structure — made up of **rows and columns** — and corresponds to a **table or dataset**.
We can think of a data_frame as a dictionary, or even better, as dictionaries nested inside a dictionary where the keys of the outer dictionary are the columns labels, the keys of the inner dictionaries are the indices of evry row and the related values are the actual data (or, in a more dataset/table sense, the outer keys are the columns and inner keys are the rows, with the inner values being the data)

```
{
  'city': {0: 'Amsterdam', 1: 'Paris', 2: 'Rome'},
  'temp': {0: 21, 1: 25, 2: 19},
}
```

Converting a data_frame to a dictionary
A data_frame can be converted to a dictionary with the method 'to_dict()' (we can convert the entire data_frame, aka all the series in it (1), or just a specific series (2)):

```
1)  data_dict = data.to_dict()
2)  data_dict = data['temp'].to_dict()
    print(data_dict)
```

Output:

```
1)  {'city': {0: 'Amsterdam', 1: 'Paris', 2: 'Rome'}, 'temp': {0: 21, 1: 25, 2: 19}}
2)  {0: 21, 1: 25, 2: 19}
```

Create a series

We can create a series in different ways by using the `.Series()` class from pandas

- `.Series(list)`: creates a series manually from a list with premade indices

```
data = [1,2,3,4,5]
series = pd.Series(data)
print(series)
```

Output:

```
0   1
1   2
2   3
3   4
4   5
dtype: int64
```

- `.Series(dict)`: converts a series into a dictionary with keys as indices and values as values

```
data = {'a': 1, 'b': 2, 'c': 3} #the keys are the indices of the series
series_dict = pd.Series(data)
print(series_dict)
```

Output:

```
a    1
b    2
c    3
dtype: int64
```

- **`.Series(list, index):`** creates a series from list of values and custom indices

```
data = [10,20,30]
index = ['a', 'b', 'c'] #the values of this list are the indices of the series

pd.Series(data, index = index)
```

Output:

```
a    10
b    20
c    30
dtype: int64
```

Create a dataframe

We can create a dataframe through the **`.DataFrame()`** class

- **`.DataFrame(dict):`** creates a dataframe from a dictionary of lists with keys as headers

```
data_dict = {
    "students": ["Amy", "James", "Angela"],
    "scores": [76, 56,65]
}

data = pandas.DataFrame(data_dict)
print(data)
```

Output:

```
     students   scores
0       Amy       76
1      James       56
2     Angela       65
```

- **`.DataFrame(dict):`** creates a dataframe from a list of dictionaries with keys as headers

```
data = [
   {'Name': 'Tyron', 'Age': 37, 'City': 'Rome'},
   {'Name': 'Eline', 'Age': 34, 'City': 'Amsterdam'},
   {'Name': 'Felipe', 'Age': 65, 'City': 'Lima'},
   {'Name': 'Luisa', 'Age': 58, 'City': 'Salamanca'}
]

df = pd.DataFrame(data)
print(df)
```

Output:

```
    Name  Age      City
0  Tyron   37      Rome
1   Eline   34  Amsterdam
2  Felipe   65      Lima
3   Luisa   58  Salamanca
```

- **`pd.DataFrame(values,col_labels,index_labels)`, `.randint()`:** creates a DataFrame from scratch with random integers, col_labels and index labels

```
df = pd.DataFrame(np.random.randint(1,50,size = (2,3)),columns = ['A','B','C'],index = ['X','Y']) #columns and index are
optional
print(df)
```

Output:

```
   A   B   C
X  7  39   4
Y  15   1  19
```

Creating a dataframe from multiple prexisitng Series

We can create a dataframe from prexisting Series in 3 different ways

- **`Dictionary of Series:` creating a dataframe by using a dictionary of the Series**

```
date_rng = pd.date_range(start = '2022-01-01', end = '2022-1-10', freq = 'D')
df = pd.DataFrame({
   'Date':np.random.choice(date_rng, size = len(date_rng)),
   'Product':np.random.choice(['A','B','C'], size = len(date_rng)),
   'Revenues':np.random.randint(10,101,size = len(date_rng)),
})
```

- **`Create the dataframe by adding Series`**

```
df = pd.DataFrame()
df['Date'] = pd.Series(pd.date_range(start ='2017-01-01',end= '2019-12-01',freq= 'M'))
df['Product'] = pd.Series(np.random.choice(['A','B','C'], size = len(date_rng)))
df['Revenues']= (np.random.randint(10,100,size = (len(date_rng))))
```

- **`.zip()` : creating a dataframe by zipping together multiple Series**

```
date = pd.Series(pd.date_range(start = '2017-01-01', end = '2019-12-01', freq = 'M'))
categories = ['A','B','C']
product = pd.Series(np.random.choice(categories, size = len(date_rng)))
revenues = (np.random.randint(10,100,size = (len(date_rng))))
d =pd.DataFrame(list(zip(date_rng,product,revenues)), columns=['Date','Product', 'Revenues'])
```

NOTE(mixed methods): To create a DataFrame from multiple Series, one must structure them using a dictionary or a zipped list of tuples. You can't pass separate Series directly into `pd.DataFrame()` without the aformentioned structures.
We cannot directly pass multiple Series as separate arguments without these structures—for example:

```
df = pd.DataFrame(date_rng, category, columns=['Date', 'Category']) # ❌ Wrong syntax
```

is invalid.
Once the DataFrame is created, new columns can only be added separately.

Converting the dataframe to csv

There's a specific method we can use to do so

```
data.to_csv("new_data.csv")
```

And the output here will be a a new csv file in the same folder we are working in.

Index/slicing and Accessing the dataframe (csv file)

`Indexing` in Pandas referes to the act of retrieving either a single element (scalar, cell) for `Series`, or a scalar, row or column when the data container is a `DataFrame`. `Slicing` referes to accessing a sub-Series (a sequence of values) in a `Series` or a subset of rows, columns or both rows and columns in a `DataFrame`.

- **`Accessing/reading` the csv file**

  ```
  df = pd.read_csv('data_pandas.csv')
  ```

  Output:
  'the entire dataset will be displayed'

- **`Direct indexing:`**

  ```
  df = pd.DataFrame(np.random.randint(1,50,size = (3,3)),columns = ['A','B','C'],index =['X','Y','Z'])
  print(df)


  print(df['B']['Y']).  #direct indexing to retrieve a scalar  in column B row Y
  ```

  Output:

  ```
  13
  ```

- **`.head(n):` retrieving the first n rows (5 rows default)**

  ```
  df.head(5) # .head() = first 5 rows
  ```

  Output:

	Date	Category	Value	Product	Sales	Region
0	2023-01-01	A	28.0	Product1	754.0	East
1	2023-01-02	B	39.0	Product3	110.0	North
2	2023-01-03	C	32.0	Product2	398.0	East
3	2023-01-04	B	8.0	Product1	522.0	East
4	2023-01-05	B	26.0	Product3	869.0	North

- **`.tail(n)`: retrieving the last n rows (5 rows default)**

```
df.tail(5) # .tail() = last 5 rows
```

Output:

	Date	Category	Value	Product	Sales	Region
45	2023-02-15	B	99.0	Product2	599.0	West
46	2023-02-16	B	6.0	Product1	938.0	South
47	2023-02-17	B	69.0	Product3	143.0	West
48	2023-02-18	C	65.0	Product3	182.0	North
49	2023-02-19	C	11.0	Product3	708.0	North

- **Retrievieng an entire column/series.** This can be done through a quicker syntax than the one introduced before: in fact, Pandas converts every column heading of the series into an attribute. Then, these two following codes are equivalent

```
data={
    'Name':['Krish','John','Jack'],
    'Age':[25,30,45],
    'City':['Bangalore','New York','Florida']
}
df = pd.DataFrame(data)
print('This is the dataframe:')
print(data)

df['Name']
df.Name  # Python converst every columns heading into an attribute
```

Output:

```
This is the dataframe:
   Name  Age      City
0  Krish   25  Bangalore
1   John   30   New York
2   Jack   45    Florida


   0    Krish
   1     John
   2     Jack
   Name: Name, dtype: object
```

- **`Df.loc[row]`: retrieving an entire row (with row-header)**

```
df.loc[0]
```

Output:

```
Name       Krish
Age           25
City    Bangalore
Name: 0, dtype: object
```

- **`.loc[row_label,col_label]`;`.iloc[row_index,col_index]`:indexing a value**

```
df.loc[1][2]
df.iloc[1][2]
```

Output:

```
'New York'
```

NOTE: Even though this two commands give the same result in the previous example (The label of the row here coincide with the index because the rows are labled through numbers), they are different in terms of how they perform the action:

- **`.loc[]` is based on label rather than integers**, as it would be in normal indexing, and it includes the stop value.
  The way this works is by first returning the first row as a series (the first [1]) and then it accesses the label 2 in that series (the second [2])
- **`.iloc[] (=index location)` is based on integer as in normal indexing** and it doesn't include the stop value.
  This works by returning row at position 1 as a Series (the first [1]), then access the element at position 2 in that row (the second [2])

- **`Indexing` a set of value with standard syntax `for .loc[]`**

```
df.loc[2, 'Name']      # Row with index label 2, column "name"
```

Output:

```
'Jack'
```

- **`Slicing` a set of value with standard syntax for `.loc[]`**

```
df.loc[1:3, 'Age']      # Rows 1 to 3 (inclusive), column "age"
df.loc[df['Age'] > 30]  # Filter with boolean condition
```

Output:

```
1    30
2    45
Name: Age, dtype: int64
```

	Name	Age	City
2	Jack	45	Florida

- **`Indexing` a set of values with standard syntax for `.iloc[]`**

```
df.iloc[0, 1]         # First row, second column
```

Output:

```
25
```

- **`Slicing` a set of values with standard syntax for `.iloc[]`**

```
df.iloc[1:3]          # Row positions 1 and 2 (exclusive end)
df.iloc[:, 0]         # All rows, first column
```

Output:

	Name	Age	City
1	John	30	New York
2	Jack	45	Florida

```
0   Krish
1    John
2    Jack
   Name: Name, dtype: object
```

- **`.at[row_label,col_label]`;`.iat[row_index,col_index]`: retrievies a single value**

```
df.at[1,'City']
df.iat[1,2]
```

Output:

```
'New York'
```

NOTE: The main difference between `loc()/iloc()` and `at()/iat()` is that the second two don’t allow slicing, meaning that we can only get one value at a times with them

What you want	Use	Index type	Slicing	Start/Stop
Single value by label	`.at[]`	Label-based	❌	No Slicing => No Start/Stop
Single value by position	`.iat[]`	Position-based	❌	No Slicing => No Start/Stop
Flexible access by label	`.loc[]`	Label-based	✅	Includes stop
Flexible access by position	`.iloc[]`	Position-based	✅	Doesn’t include stop

Mathematical operation on the dataframe

```
df = pd.DataFrame(np.random.randint(1,10,size=(5,3)))
print(df)
```

Ouput:

```
  A  B  C
X  8  1  6
Y  6  9  2
```

- **Element-wise operation (via broadcasting) between either a row or columns and a scalar. The results will be displayed in a new added row/column**

```
df['Col_A+2'] = df['A’] + 2      #adds 1 to all the elements of column 0
print(df)
```

```
df['Col_A-2'] = df[’A’] - 2     #subtracts 2 from every element of the column

df.loc[’X’] = df.loc[’X’] * 2    #multiplies 2 to every element of the row

df.iloc[’X’] = df.iloc[0] / 2    #divides by 2 all the elements of the row
```

Output:

```
  A  B  C  Col_A+2
X  8  1  6       10
Y  6  9  2        8
```

- **Element-wise operations (via broadcasting) between columns. The results will be displayed in a new added column the result as a new column at the right-end of the DataFrame.**

```
df['Col_sum'] = df[’A’] + df[’B’]   #new column Col_sum, sum of col A and B
print(df)

df['Col_diff'] = df[’A’] - df[’B’]  #new column Col_diff, difference of col A and B

df['Col_prod'] = df[’A’] * df[’B’]  #new column Col_diff, product of col A and B

df['Col_divis'] = df[’A’] / df[’B’] #new column Col_diff, division of col A and B
```

Output:

```
  A  B  C  Col_sum
X  8  1  6       9
Y  6  9  2      15
```

- **Element-wise operations (via broadcasting) between rows. The results will be displayed in a new added row at the bottom of the DataFrame.**

```
df.loc['Row_sum'] = df.loc[’X’] + df.loc[’Y’]     #creates a new row called Row_sum
print(df)

df.loc['row_diff'] = df.iloc[0] - df.iloc[1]      #creates a new row called Row_diff

df.loc['row_prod'] = df.loc[’X’] * df.iloc[1]     #creates a new row called Row_prod

df.loc['row_divis'] = df.iloc[0] / df.loc[’Y’]    #creates a new row called Row_divis
```

Output:

```
        A   B  C
X       6   9  2
Y       9   2  3
```

```
Row_sum  15  11  5
```

NOTE: we can perform operations between two or more `.loc[]`, two or more `.iloc[]` or between a mix of `.loc[]` and `.iloc[]`

- **All the previous operations can be performed in place, without generating a new column but instead returning the original dataframe with the updated Series. The difference is that here we use the name or the index on an already existing row/column**

```
df = pd.DataFrame(np.random.randint(1,10,size=(2,3)))
print(df)

df['A']=df['A']+2
print(df)

df.loc['X'] = df.loc['X'] - df.loc['Y']

df.iloc[0] = df.iloc[0] * df.iloc[1]

df.iloc[0] = df.iloc[0] / df.loc['Y']

df['A'] += 1              #add 1 to column A

df.loc['X'] *= 2           #multiplies row X by 2
```

__Ouput:__

```
  A  B  C
X  8  1  6
Y  6  9  2

  A  B  C
X  10  1  6
Y   8  9  2
```

- **Returns the result of the operation, works with either to the row or to the column**

```
row_sum = df.sum(axis = 1)   #sums elements in a row, aplies to each row (axis=1)

col_prod = df.prod(axis = 0) #multiplies the elements in a column, every column

row_diff = df.diff(axis = 1) #subtracts each element of a row from the element from the previous column
print('sum rows')
print(row_sum)
```

__Output:__

```
difference rows
   0  1  2
0 NaN -3  1
```

```
1 NaN  1  1
```

NOTE: the NaN value (not a number) is the result of an undefined mathematical operation. In this case come from the fact that there not other column on the left of the first one!

- **Applies the operation between a single Series and all the other Series**

```
df.add(df[0], axis=1)       #adds column 0 to every column, row by row


df.mul(df.loc[0], axis=0)    #multiplies row 0 to every row, column by column


df.subtract(df[1], axis=1)   #subtracts column 1 from every column, row by row


df.divide(df.iloc[1], axis=0) #divides row 1 from every row, column by column (iloc = index)
print(df)
```

Manipulating the dataframe (csv file)

- **Adding a column**

```
df['Salary'] = [50000, 60000, 70000]
df
```

Output:

	Name	Age	City	Salary
0	Krish	25	Bangalore	50000
1	John	30	New York	60000
2	Jack	45	Florida	70000

- **Change column's name with .rename(columns = {:})**

```
df = df.rename(columns = {'Date': 'Sales_Date'})  # always with a dictionary
df.head()
```

Output:

	Sales_Date	Category	Value	Product	Sales	Region	Sales_fillNA
0	2023-01-01	A	28.0	Product1	754.0	East	754.0

- **`.drop(col_label, axis, permanent)` : deleting a column**

```
df.drop('Salary', axis = 1, inplace = True)
```

Output:

	Name	Age	City
0	Krish	26	Bangalore
1	John	31	New York
2	Jack	46	Florida

NOTE: The `.drop()` method in pandas **does not make permanent changes by default**. It returns a **new DataFrame** with the specified rows or columns removed, unless you explicitly set:

  - inplace=True → to apply the change directly to the original DataFrame
  - axis=0 → to drop **rows** (default)
  - axis=1 → to drop **column**

- **`.drop()`: deleting a column with**

```
df.drop(0, inplace=True)
df
```

Output:

	Name	Age	City
1	John	30	New York
2	Jack	45	Florida

NOTE:
- Since we are **not specifying the `axis` parameter**, pandas uses the **default**:
  - axis=0 → which means we are targeting **rows**.
- The `0` refers to the **row label**, not the position (unless your row labels are 0, 1, 2...).
- Setting inplace=True makes the deletion **permanent** on the original DataFrame.

- **`.apply(); lambda function`: applies a function to a single row/column (i.e. x => 2x)**

```
df['New_Valeu'] = df['Value'].apply(lambda x: x*2)
df.head()
```

Output:

	Sales_Date	Category	Value	Product	Sales	Region	Sales_fillNA	Value_new	New_Valeu
0	2023-01-01	A	28.0	Product1	754.0	East	754.0	28	56.0
1	2023-01-02	B	39.0	Product3	110.0	North	110.0	39	78.0

- **`.apply()`: creates a new column as the sum of the other ones**

```
df = pd.DataFrame(np.random.randint(1,51,size = (6,3)))


df['Sum']= df.apply(lambda row: row.sum(), axis =1)
print(df)
```

- **`.applymap()`: applies a function to the entire dataframe (multiplies every cell per 2)**

```
df_doubled = df.applymap(lambda x: x * 2)
```

Data analysis methods with Pandas

- **`.dtypes` : retrieving data type (attribute)**

```
# Display the data types of each column
print("Data types: \n", df.dtypes) #in the form of a normal print (outup example)
#or
df['Value'].dtypes #in the form of a table
```

Output (in the form of a print):

```
Date        object
```

```
Category     object
Value        float64
Product      object
Sales        float64
Region       object
dtype: object
```

- **`.describe()` : retrieving statistical information about the dataframe (method)**

```
# Describe the DataFrame
print("Statistical summary:\n", df.describe()) #in the form of a print

#or

df.describe() #in the form of a table (output example)
```

Output:

	Value	Sales
count	47.000000	46.000000
mean	51.744681	557.130435
std	29.050532	274.598584
min	2.000000	108.000000
25%	27.500000	339.000000
50%	54.000000	591.500000
75%	70.000000	767.500000
max	99.000000	992.000000

- **`.groupby()` : group by a column and perform an aggregation (.mean()) (methods)**

```
grouped = df.groupby('Category')['Value'].mean()
print(grouped)
```

Output:

```
Category
A    48.357143
B    44.142857
C    59.842105
Name: Value, dtype: float64
```

- **`.any()` : checks for non-zero or (explicitely) True (method)**

```
df.any()
```

Output:

```
Date        True
Category    True
Value       True
Product     True
Sales       True
```

```
Region     True
dtype: bool
```

```
df.any(axis=1)
```

Output:

```
0    True
1    True
2    True
3    True
4    True
...
```

NOTE: The .any() method checks whether **at least one truthy value** exists along a specified axis in a DataFrame. It returns a **Series of booleans**, not a single value

- When **no axis is specified** or **axis=0** (default):
  .any() (rows) evaluates each **column**, looking **down every row per column**
  → Returns True for a column if **any row** in that column is truthy
  → Returns False if **all rows** in that column are falsy
- When **axis=1**:
  .any(axis=1) (columns) evaluates each **row**, **across all the columns:**
  → Returns True for a row if **any column** in that row is truthy
  → Returns False if **all values** in the row are falsy

This is how .any() evaluates values:

- Evaluates to True: "Hello", 123, True
- Evaluates to False: 0, False, None, Nan, ""(empty string)

NOTE: `.any()` on its own is rarely helpful, it's mostly useful in combination with `.isnull()`

▪ `.isnull()` : handling missing values with (method)

```
df.isnull()
```

Output:

15	False	False	True	False	False	False
16	False	False	False	False	False	False
17	False	False	True	False	False	False

NOTE: The **.isnull()** method evaluates the **entire DataFrame or Series**, and returns a structure of the **same shape**, where:

- **Missing values (`NaN`, `None`) are marked as True**
- **All other values are marked as False**

So it **flags missing values** as True and everything else as False.

▪ Handling missing values with `.isnull()` and `.any()` or a certain ammount of rows

```
# handling missing values # evaluates the rows = every row for each column (Output 1)
df.isnull().any()
df.isnull().any(axis = 1).head(5) # evaluates the first 5 rows (Output 2)
df.isnull().iloc[10:16] # It evaluates rows from 10 to 15 (Output 3)
df.isnull().head(15).tail(10) # same as the previous but less efficient and readable
```

Output:

1)

```
 Date       False
Category    False
Value       True
Product     False
Sales       True
Region      False
dtype: bool
```

2)

```
0    False
1    False
2    False
3    False
4    False
```

3)

	Date	Category	Value	Product	Sales	Region
10	False	False	False	False	False	False
11	False	False	False	False	True	False
12	False	False	False	False	False	False
13	False	False	False	False	False	False
14	False	False	False	False	False	False
15	False	False	True	False	False	False

- **Finding the number of null values per column**

```
df.isnull().sum()
```

Output:

```
Date       0
Category    0
Value      3
Product     0
Sales      4
Region      0
dtype: int64
```

- **`.fillna(n)` : filling missing values (None, Nan) with the desired value (method)**

```
df.fillna(0)
```

NOTE: here we are filling missing values with 0 just for clarity. **In a real scenario, we want to fill a missing values with the distribution/column's mean (for normal distributions), with median (for skewed distribiutions or with outliers) or mode (for categorical variables)!**

Output:

…

10	2023-01-11	B	7.0	Product1	842.0	North
11	2023-01-12	B	60.0	Product2	0.0	West
12	2023-01-13	A	70.0	Product3	628.0	South

…

- **Filling missing values with mean of the column**

```
# Step 1: Compute the mean of the 'Sales' column
average = df['Sales'].mean()
# Step 2: Create a new column where missing values are replaced by the mean
df['Sales_fillNA'] = df['Sales'].fillna(average)


# Same thing, just more concise — directly apply the mean in the fillna method
df['Sales_fillNA'] = df['Sales'].fillna(df['Sales'].mean())
```

Output:

Mean = 51.744680851063826

Date	Category	Value	Product	Sales	Region	Sales_fillNA

…

10	2023-01-11	B	7.0	Product1	842.0	North	842.000000
11	2023-01-12	B	60.0	Product2	NaN	West	557.130435
12	2023-01-13	A	70.0	Product3	628.0	South	628.000000

…

- **`.astype()`: change datatypes of an entire column with**

```
# creating a new column from Value where we filled al Nan with the mean and changed the data to int
df['Value_new'] = df['Value'].fillna(df['Value'].mean()).astype(int)
df.head()
```

Output:

	Sales_Date	Category	Value	Product	Sales	Region	Sales_fillNA	Value_new
0	2023-01-01	A	28.0	Product1	754.0	East	754.0	28

'Value' = float64 => 'Value_new' = int64

Grouping and Aggregating Data

Similarly to SQL, performing a grouping in pandas requires an aggregation (or a custom function, like a lambda) so that for each different value of the grouped-by column we receive a meaningful output. Grouping without aggregation would return a GroupBy object (`<pandas.core.groupby.generic.DataFrameGroupBy object at 0x...>`), which is just a container and not a visible result. Without an aggregation, groupby wouldn't make sense because the system wouldn't know what to attach for every distinct element of the grouped column.

- **Grouping by one category on an aggregation with .groupby()**

```
grouped_mean = df.groupby('Product')['Value’].mean()
grouped_mean
```

Output:

```
  Product
Product1    46.214286
Product2    52.800000
Product3    55.166667
Name: Value, dtype: float64
```

- **Grouping by two categories on a aggregation with .groupby()**

```
grouped_sales_region_sum = df.groupby(['Product', 'Region'])['Value'].sum()
grouped_sales_region_sum # Output
###
grouped_sales_region_mean = df.groupby(['Product', 'Region'])['Value'].mean()
grouped_sales_region_mean
```

Output:

```
  Product   Region
Product1  East      41.714286
          North     4.500000
          South     50.000000
          West      82.000000
Product2  East      28.000000
          North     63.500000
          South     60.333333
          West      53.500000
Product3  East      50.500000
          North     40.600000
          South     71.666667
          West      62.166667
Name: Value, dtype: float64
```

- **Grouping by one category on multiple aggregattion with .groupby() and .agg([,])**

```
grouped_agg = df.groupby('Region')['Value'].agg(['mean','sum','count'])
grouped_agg
```

Output:

	mean	sum	count
Region			
East	42.307692	550.0	13
North	37.666667	339.0	9
South	62.000000	496.0	8
West	61.588235	1047.0	17

Merging and Joining Dataframes

Let’s start by creating two Dataframes to showcase these operations

```
df1 = pd.DataFrame({'key': ['A', 'B', 'C'], 'Value1': [1,2,3]})
df2 = pd.DataFrame({'key': ['A', 'B', 'D'], 'Value1': [4,5,6]})
```

Output:

df1

	key	Value1
0	A	1
1	B	2
2	C	3

df2

	key	Value1
0	A	4
1	B	5
2	D	6

- **Merging two Dataframes on a key with .merge(): inner join**

```
pd.merge(df1,df2, on='key',how='inner')
```

Output:

	key	Value1_x	Value1_y
0	A	1	4
1	B	2	5

NOTE: .merge(how = ‘inner’) is equivalent to inner joins in SQL: they only return rows with keys that are common to both tables.
Rows without matching keys in either table are discarded.

- **Merging two Dataframes on a key with .merge(): outer join**

```
pd.merge(df1,df2, on='key',how='outer')
```

Output:

	key	Value1_x	Value1_y
0	A	1.0	4.0
1	B	2.0	5.0
2	C	3.0	NaN
3	D	NaN	6.0

NOTE: .merge(how = 'outer') is equivalent to an outer join in SQL: it returns all rows from both tables, filling in `NaN` where there are no matching values.

- **Merging two Dataframes: left outer join**

```
pd.merge(df1,df2, on='key',how='left')
```

Output:

	key	Value1_x	Value1_y
0	A	1	4.0
1	B	2	5.0
2	C	3	NaN

NOTE: The left outer join prioritizes the left table, meaning it includes all rows from the left table and fills in `NaN` for non-matching rows from the right table.

- **Merging two Dataframes: right outer join**

```
pd.merge(df1,df2, on='key',how='right')
```

Output:

	key	Value1_x	Value1_y
0	A	1.0	4
1	B	2.0	5
2	D	NaN	6

NOTE: The left outer join prioritizes the left table, meaning it includes all rows from the left table and fills in `NaN` for rows with no matching values in the right table.

Datestamps in Pandas

`.data_range(start = ‘YYYY-MM-DD’,end,period,frequency= ‘D’,‘M’,’Y’ ):`

```
date =pd.date_range(start = '2017-01-01', end = '2019-12-01', freq = 'M')
###
date = pd.Series(pd.date_range(start = '2017-01-01', end = '2019-12-01', freq = '3M'))
```

Multindexing (hierarchicla index)

MultiIndexing in pandas allows us to use multiple columns as a layered index, so we can organize and navigate grouped data **without collapsing rows**.

It’s like grouping by more than one column, but instead of summarizing, it keeps the full structure and detail

of the dataset
It helps organize data in layers (like a tree), so you can easily group, filter, and navigate complex datasets without losing detail.

For example: if you group data by both "Year" and "Country", you can use MultiIndexing to keep both levels as the index — instead of flattening or aggregating them.

```
# Sample data
data = {
    'Year': [2020, 2020, 2021, 2021],
    'Country': ['USA', 'Canada', 'USA', 'Canada'],
    'Value': [100, 80, 110, 90]
}
# Create DataFrame
df = pd.DataFrame(data)
# Set MultiIndex
df_multi = df.set_index(['Year', 'Country'])
print(df_multi)
```

Output:

```
             Value
Year Country
2020 USA       100
     Canada     80
2021 USA       110
     Canada     90
```

- **`.MultiIndex.from_arrays(arr,col_names=(,))` : creates a multiindex dataframe**

```
array = [['A','A','B','B'],['one','two','one','two']]
index = pd.MultiIndex.from_arrays(array, names = ('Category','Subcategory'))
df1 = pd.DataFrame(np.random.randint(1,100,size = (4,3)), index = index, columns = ['Value1','Value2','Value3'])
print('Dataframe after multindexing:')
print(df1)
```

Output:

```
Dataframe after multindexing:


                      Value1  Value2  Value3
Category Subcategory
A        one              61      53      95
         two              68      66      88
B        one              26      97      27
         two              35      83      91
```

- **`Indexing and slicing in multiindex dataframe:` we can either access category and subcategory by explicitely calling them or we can acces values from the actual values by using normal indexing and slicing syntax**

```
print('Elements in Category A')
print(df.loc['A'])

print('Element in Category B, Subcategory two')
print(df.loc[('B','two')])

print('Element 1st row, 2nd column from Values')
print(df1.iloc[0,1])

print('Elements from all rows, column 2nd to 3rd from Values')
print(df1.iloc[0:,1:])

print('Elements from column Value1')
print(df1['Value1'])
```

__Output:__

```
Elements in Category A
            Value1  Value2
Subcategory
one             18       5
two             20      12

Elements in Category B, Subcategory two
Value1    9
Value2    7
Name: (B, two), dtype: int64

Element 1st row, 2nd column
4

Elements from all rows, column 2nd to 3rd
                     Value2  Value3
Category Subcategory
A        one              4      57
         two             67      10
B        one             97      12
         two              6      67

Elements from column Value1
Category  Subcategory
A         one           89
          two           29
B         one           24
          two            1
                    Name: Value1, dtype: int64
```

- **`.groupby([col_name1,col_name2,…])` : summing elements by category after group**

```
sum_category = df.groupby(['Category']).sum()
print('Sum of values for each Category')
print(sum_category)
```

Output:

```
Sum of values for each Category
        Value1  Value2
Category
A          38     17
B          29     21
C          24     27
```

Pivot Table

A **pivot table** in pandas is a tool that allows us **summarize and reorganize data** by grouping it along one or more keys (rows and columns) and applying an aggregation function (like `mean`, `sum`, etc.) to a specific value column.

For example, in a table with date, product and revenues, we could group revenues data by *product* (rows) and *date* (columns), and calculate the **average sales** for each.

```
date_rng = pd.date_range(start = '2022-01-01', end = '2022-1-10', freq = 'D')
df = pd.DataFrame({
   'Date':np.random.choice(date_rng, size =30),
   'Product':np.random.choice(['A','B','C'], size = 30),
   'Revenues':np.random.randint(10,101,size = 30)
})
```

- **`.pivot_table()` : creating a basic pivot table to calculate the sum of revenues by year and product**

```
pivot=df.pivot_table(values='Revenues',index='Date',columns='Product',aggfunc ='sum')
print('Total Revenues per product by year')
print(pivot)
```

Output:

```
Total Revenues per product by day
Product       A    B    C
Date
2022-01-02   NaN   NaN  18.0
2022-01-03   NaN  20.0   NaN
2022-01-05   NaN   NaN  99.0
2022-01-06  67.0   NaN  24.0
2022-01-08   NaN  94.0   NaN
2022-01-09   NaN   NaN  35.0
2022-01-10  37.0   NaN   NaN
```

- **`.Grouper():` class to aggregate the output by a specific range of one of the categories**

```
pivot = pd.pivot_table(df,index ='Category',columns=pd.Grouper(key='Date',freq= 'YE'), values = 'Values', aggfunc =
'mean')
print('Mean of values for category by year')
print(pivot)
```

Output:

```
Mean of values for category by year
Date      2017-12-31  2018-12-31  2019-12-31
Category
A            28.50       66.6   57.714286
B            35.00       55.0   59.500000
C            61.75       38.5   27.500000
```

Common Functions, Methods, Attributes and Classes in Pandas

As in other libraries (or on SQL), we can perform actions on the items/values inside a data_frame. For example, we can calculate mean, max and min:

- **`np.random.choice(object,size):` returns rando items from a container**

```
date_rng = pd.date_range(start = '2022-01-01', end = '2022-1-10', freq = 'D')
random_date = np.random.choice(date_rng, size = len(date_rng))
###
random_category = np.random.choice(['A','B','C'], size = len(date_rng))
```

- **`pd.concat([objects], axis):` combines dataframe or Series along a certain axis**

```
df1 = pd.DataFrame(np.random.randint(1,10, size = (2,2)), columns = ['A','B'])
df2 = pd.DataFrame(np.random.randint(1,10, size =(2,2)),columns = ['A','B'])
print(df1)

print(df2)

vertical = pd.concat([df1, df2], axis=0)  # Vertical (default)
horizontal = pd.concat([df1, df2], axis=1)  # Horizontal
print('vertical:')
print(vertical)

print('horizontal:')
print(horizontal)
```

Output:

```
   A  B
0  2  1
1  3  8

   A  B
0  5  6
```

```
1  4  7


vertical:
   A  B
0  2  1
1  3  8
0  5  6
1  4  7


horizontal:
   A  B  A  B
0  2  1  5  6
1  3  8  4  7
```

- **`np.nan`: represents a non number element (can be used to set values to nan)**

```
df = pd.DataFrame({'A': [1, 2, np.nan, 4]})
```

- **`.sum()`: sums the desired values**

```
df = pd.DataFrame(np.random.randint(1,10,size = (3,1)), columns = ['Product'])
df['Category']= np.random.choice(['A','B'], size = (3,1))
sum_category = df.groupby(['Category']).sum()
print(df)
print(sum_category)
```

Output:

```
   Product Category
0       1       B
1       4       A
2       3       A
        Product
Category
A           7
B           1
```

- **`.mean()`:**

```
average_temp1 = average(temp)
print(round(average_temp1, 2))

average_temp2 = sum(temp) / len(temp)
print(round(average_temp2,2))

average_temp3 = data["temp"].mean()
print(round(average_temp3,2))
```

- **`.max()/.min()`**

```
max_temp = data["temp"].max()
print(max_temp)

print(data["temp"].max())
```

As we said, we might want to retrieve a specific series from a data_frame. This can be done through a quicker syntax than the one introduced before: in fact, Pandas converts every column heading of the series into an attribute. Then, these two following codes are equivalent

```
print(data["condition"])
print(data.condition) #Python converts every column heading into an attribute
```

If we wanted to retrieve the information about a specific row, we could specify as the key of the data_frame a condition that filters the data based on one of its columns.
For example, if we wanted all the rows where the day is "Monday"**,** we would write:

```
print(data[data.day == "Monday"])
```

Output:

```
     day    temp    condition
0  Monday      12       Sunny
```

In the same way, if the row we want to retrieve is based on a calculation performed by Python, we can apply the same logic. This time, however, we include the calculation directly in the condition. For example, to get the row with the maximum or minimum temperature, we write:

```
print(data[data.temp == data.temp.max()])
print(data[data.temp == data.temp.min()])
```

Output:

```
     day   temp   condition
6  Sunday     24       Sunny


###
     day  temp    condition
0  Monday    12       Sunny
```

If we wanted to extract a specific data/value from a (filtered) row, we could store the filtered DataFrame in a variable, and then use attribute access to retrieve the desired column.
For example, if we wanted to get the condition for Monday, we could write:

```
monday = data[data.day == "Monday"]
print(monday.condition)
```

Output:

```
0   Sunny
```

If then we wanted to convert Monday’s temperature from Celsius to Fahrenheit we could just use the following code

```
monday = data[data.day == "Monday"]
print((monday.temp)*9/5+32)
```

or

```
monday = data[data.day == "Monday"]
monday_temp = monday.temp
print(Monday_temp * 9/5+32)
```

Output:

```
0   53.6
```

Using strings in Pandas

```
text_data = pd.Series(['apple', 'banana', 'cherry', 'date', 'elderberry'])
print("Original Series:")
print(text_data)
```

- `.upper():` **converting strings to upper case**

  ```
  # Convert all the strings to uppercase
  uppercase_data = text_data.str.upper()
  print("Uppercase Series:")
  print(uppercase_data)
  ```

- `Indexing/slicing:` **retrieving the first 3 characters of each string**

  ```
  # Extract the first three characters of each string
  first_three_chars = text_data.str[:3]
  print("First three characters:")
  print(first_three_chars)
  ```

  Output:

  ```
  Original Series:
  0        apple
  1       banana
  2       cherry
  3         date
  4   elderberry
  dtype: object
  First three characters:
  0   app
  1   ban
  2   che
  3   dat
  4   eld
  dtype: object
  ```

PANDSA: Reading Data From Various Sources

MATPLOTLIB: Data Visualization

Matplotlib is a popular Python library used for creating static, animated, and interactive visualizations. It's especially useful for data analysis and scientific plotting.

We can create line charts, bar plots, histograms, scatter plots, and more with simple commands.

First step is to install the library

```
!pip install matplotlib
```

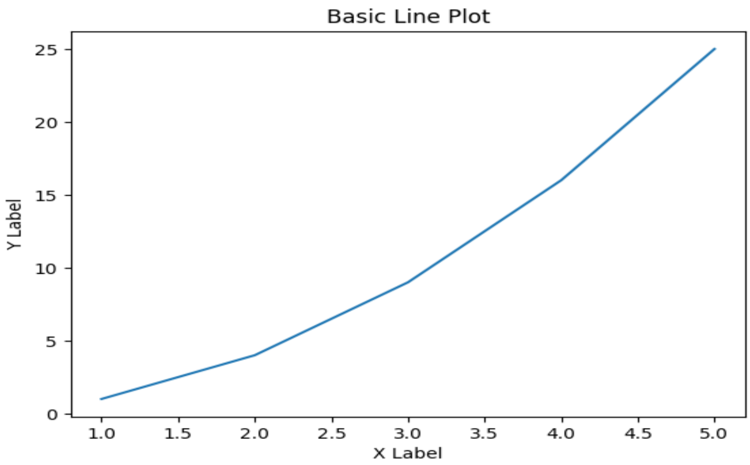
And import it

```
import matplotlib.pyplot as plt
```

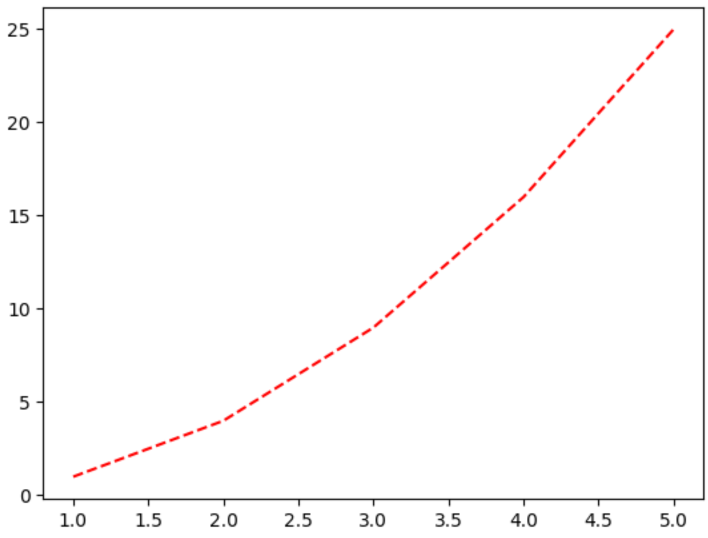
Pyplot is the most important library through which most of the function to create plots will be created

Basic line plot

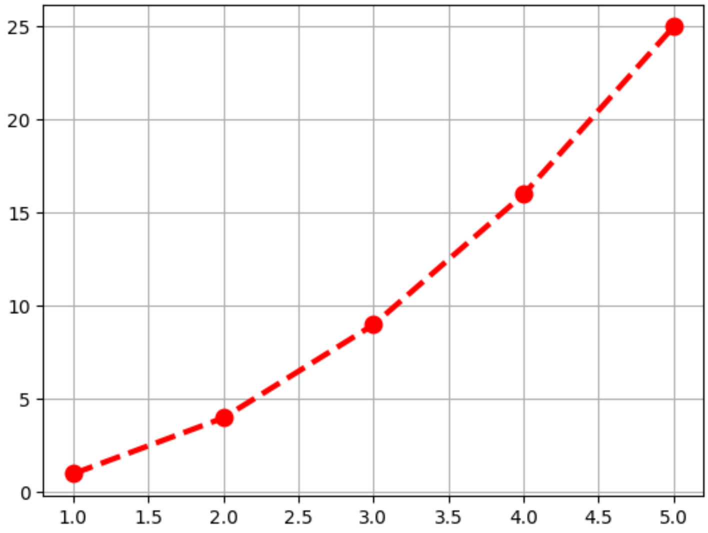
```
x = [1,2,3,4,5]   # X axis values
y = [1,4,9,16,25] # Y axis values


#creating a line plot
plt.plot(x,y)
plt.xlabel('X Label')
plt.ylabel('Y Label')
plt.title('Basic Line Plot')
```



```
plt.plot(x,y, color = 'red', linestyle = '--') #or linestyle = '-.'
```



```
plt.plot(x,y, color = 'red', linestyle = '--', marker ='o', linewidth = 3, markersize = 9)
plt.grid(True)
```



- **Setting the size of the canvas with plt.figure(figsize)**

```
plt.figure(figsize = (9,5))
```

.figure() creates a new figure object while the parameter **figsize** sets the **size of the entire figure (canvas)** — not the size of a single plot — in **inches**.

- The first number (`9`) is the **width** in inches.
- The second number (`5`) is the **height** in inches.

So `figsize=(9, 5)` means: "Create a figure that is 9 inches wide and 5 inches tall."

Multiple linear plots and multiple lines

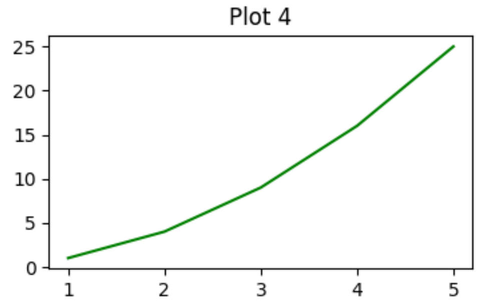
- **Multiple plots with .subplot()**

.subplot() divides the space for the output/plots in a grid. The parameters in the parenthesis works as follow:

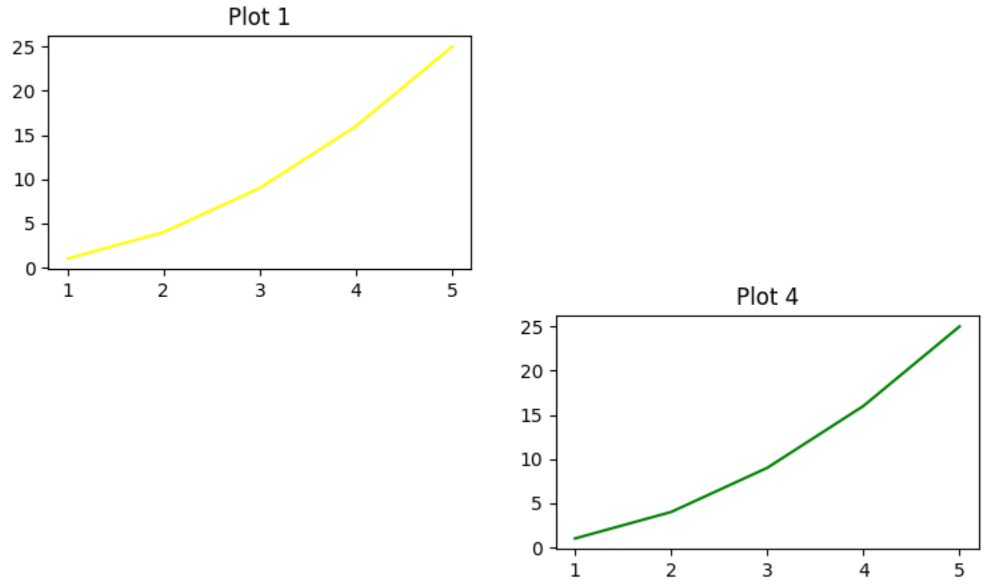
- **First parameter:** defines the number of row of the grid
- **Second parameter:** defines the number of columns of the grid
- **Third parameter:** defines the portion of the grid occupied by the current plot

Example: `(1, 2, 2)` divides the space into 1 row and 2 columns (i.e., 2 plot areas in total) and places the current plot in the **second position**, which is the **right-hand plot**.

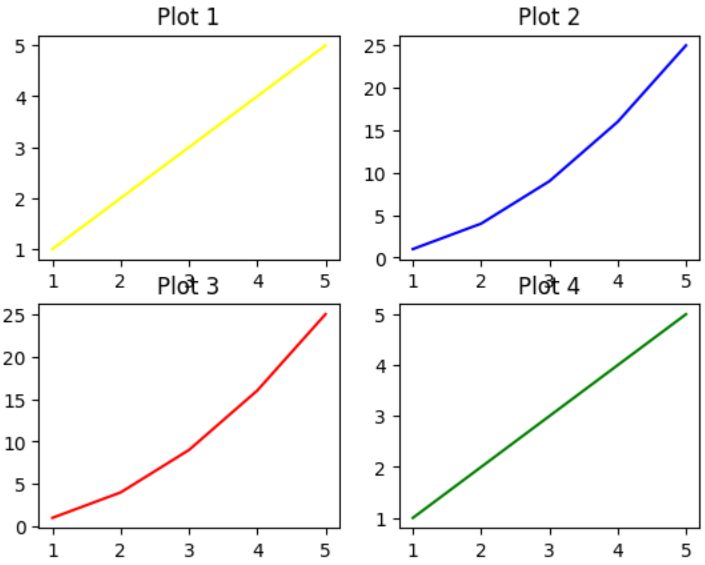
```
x = [1,2,3,4,5]
y1 = [1,4,9,16,25]
y2 = [1,2,3,4,5]


plt.figure(figsize = (9,5))
plt.subplot(1,2,1)
plt.plot(x,y1, color = 'green')
plt.title('Plot 1')
```



```
            plt.subplot(2,2,1)
plt.plot(x,y1, color = 'green')
plt.title('Plot 4')

plt.subplot(2,2,4)
plt.plot(x,y1, color = 'red')
plt.title('Plot 4')
```

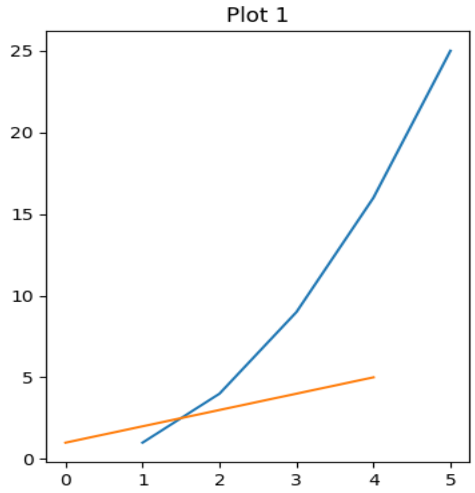


```
plt.subplot(2,2,1)
plt.plot(x,y2, color = 'yellow')
plt.title('Plot 1')

plt.subplot(2,2,2)
plt.plot(x,y1, color = 'blue')
plt.title('Plot 2')

plt.subplot(2,2,3)
plt.plot(x,y1, color = 'red')
plt.title('Plot 3')
plt.subplot(2,2,4)
plt.plot(x,y2, color = 'green')
plt.title('Plot 4')
```

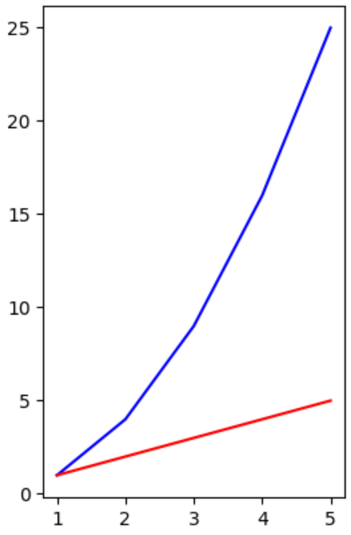


- **Multiple axis/charts**

```
x = [1,2,3,4,5]
y1 = [1,4,9,16,25]
y2 = [1,2,3,4,5]

plt.figure(figsize = (9,5))

plt.subplot(1,2,1)
plt.plot(x,y1, y2)
plt.title('Plot 1')
```



```
#if we want to choose the color of each chart we should plot each line separately
plt.subplot(1, 2, 1)

plt.plot(x, y1, color='blue', label='y1')
plt.plot(x, y2, color='red', label='y2')
```



Histograms and Barplot

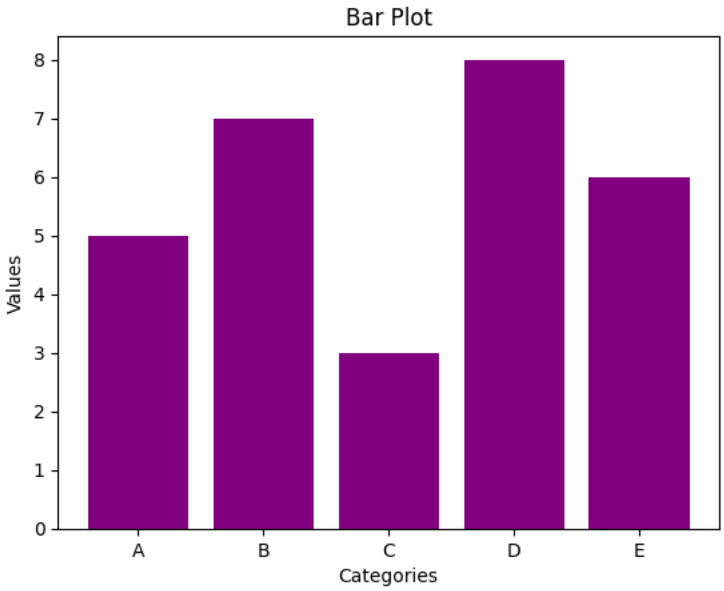
Bar Plots and **Histograms** both use bars, but they are used for different types of data.

- **Bar Plot:**
  Used to represent **categorical data** — it shows values for **distinct categories**.
  Example: sales by product type, counts by country, etc.
  Bar plots divide values into **named categories** (like "Apples", "Bananas", "Oranges").

- **Histogram:**
  Used to represent **continuous numerical data** — it shows how data is **distributed across intervals (bins)**.
  Example: distribution of ages, exam scores, temperatures.
  Histograms divide values into **numeric ranges** (like 0–10, 10–20, etc.).

  - **Simple Bar Plot using categories and .bar(x,y,…)**

```
categories = ['A','B', 'C', 'D', 'E'] # X-axis
values = [5,7,3,8,6] # Y-axis

plt.bar(categories, values, color = 'purple') #.bar(X,Y)
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Bar Plot')
plt.show() #this command is not necessary in some GUI
```

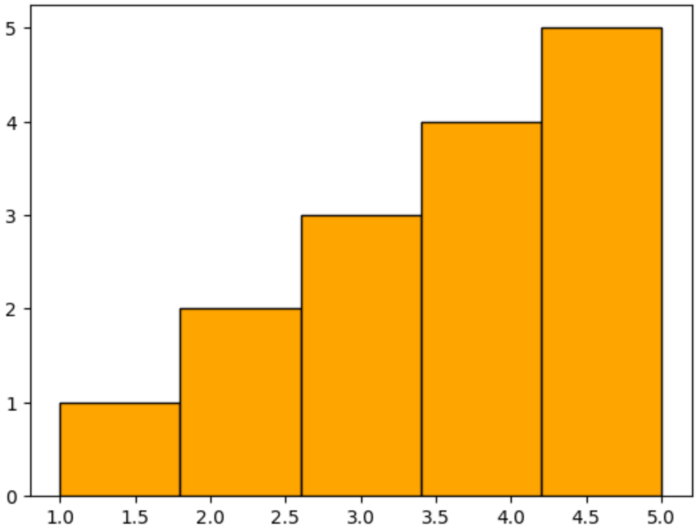


- **Simple Histogram with .hist(data, n bins,…)**

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


plt.hist(data, bins = 5, color = 'orange', edgecolor = 'black')
```

Output:

```
(array([1., 2., 3., 4., 5.]),
 array([1. , 1.8, 2.6, 3.4, 4.2, 5. ]),
 <BarContainer object of 5 artists>)
```

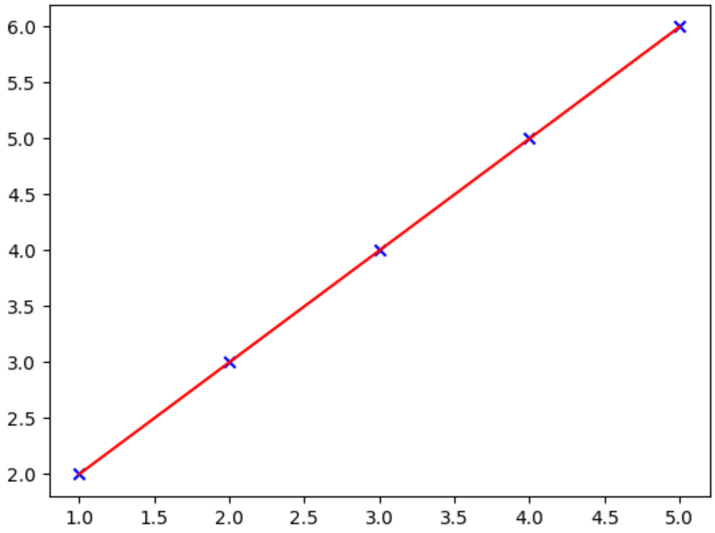


NOTE: there 5 bins for 5 different data, so Python divided the number from 1 to 5 (four units) in 5 bins. This way, the bins edges are at a distance from each other of 0,8 (<1). If we had reuqested bins = 4, the edges would be right on the ticks (1,2,3,4,5) of the x-axis

Scatter Plot

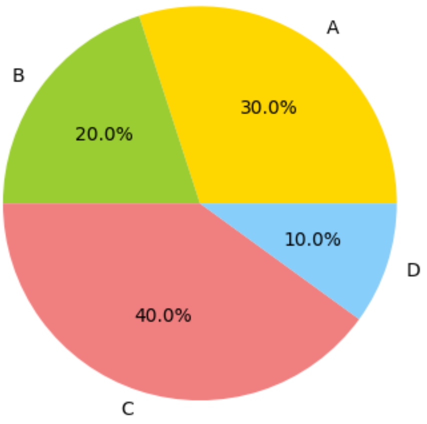
Scatter plot with .scatter(x,y,…)

```
x = [1,2,3,4,5]
y = [2,3,4,5,6]

plt.scatter(x,y, color = 'blue', marker = 'x')
plt.plot(x,y, color = 'red')
```



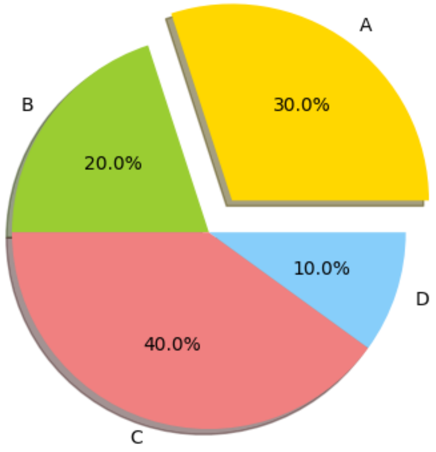
Pie Chart

```
labels = ['A', 'B', 'C', 'D']
sizes = [30,20,40,10]
colors = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue']

plt.pie(sizes, labels = labels, colors = colors, autopct = "%1.1f%%")
```



```
labels = ['A', 'B', 'C', 'D']
sizes = [30,20,40,10]
colors = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue']
explode = (0.2, 0, 0, 0) ##move out the first slice
```

```
plt.pie(sizes,explode = explode,labels = labels,colors = colors,autopct = "%1.1f%%",shadow = True)
```



Practical Example

1) First step: create the data frame from the csv file

```
import pandas as pd


df = pd.read_csv('data_pandas.csv')
```

Output (df.head(5)):

	Date	Category	Value	Product	Sales	Region
0	2023-01-01	A	28.0	Product1	754.0	East
1	2023-01-02	B	39.0	Product3	110.0	North
2	2023-01-03	C	32.0	Product2	398.0	East
3	2023-01-04	B	8.0	Product1	522.0	East
4	2023-01-05	B	26.0	Product3	869.0	North

2) Second step: Organize the data to get the sum of the sales for each category of product

```
total_sales_by_product = df.groupby('Category')['Sales'].sum()
total_sales_by_product
```

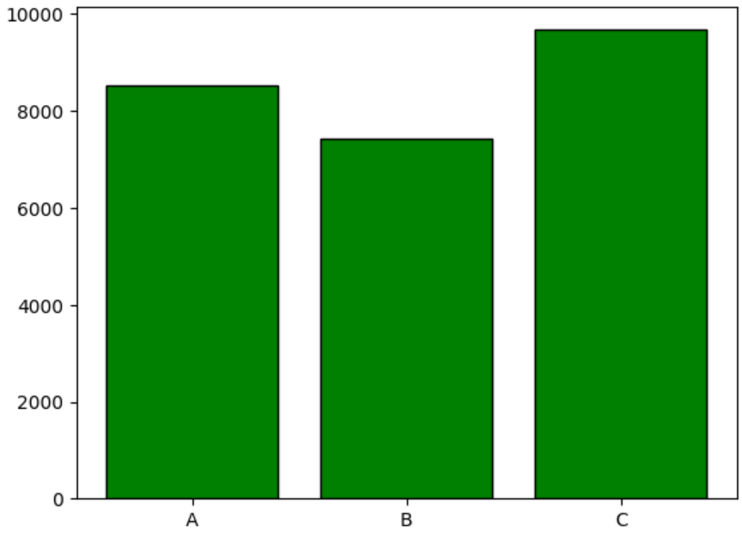
Output:

```
  Category
A   8527.0
B   7425.0
C   9676.0
Name: Sales, dtype: float64
```

3) Building the plot: bar plot

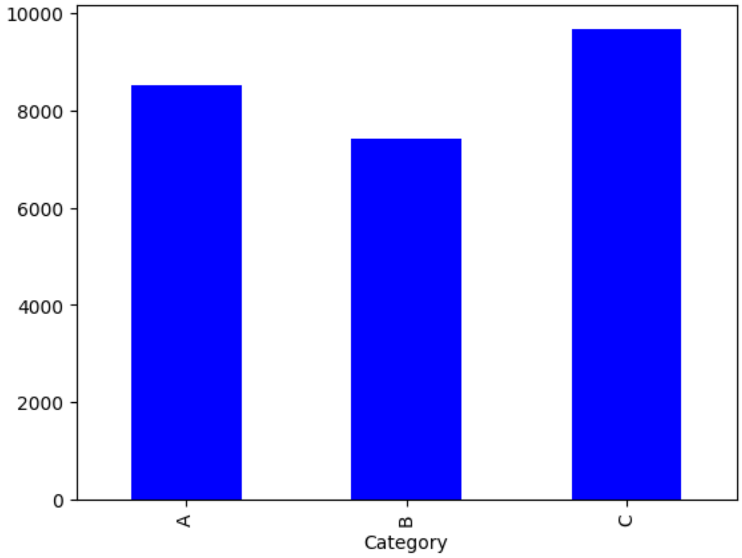
```
categories = df['Category'].unique()
values = df.groupby('Category')['Sales'].sum()
```

```
plt.bar(categories, values, color = 'green', edgecolor = 'black')
plt.show()
```



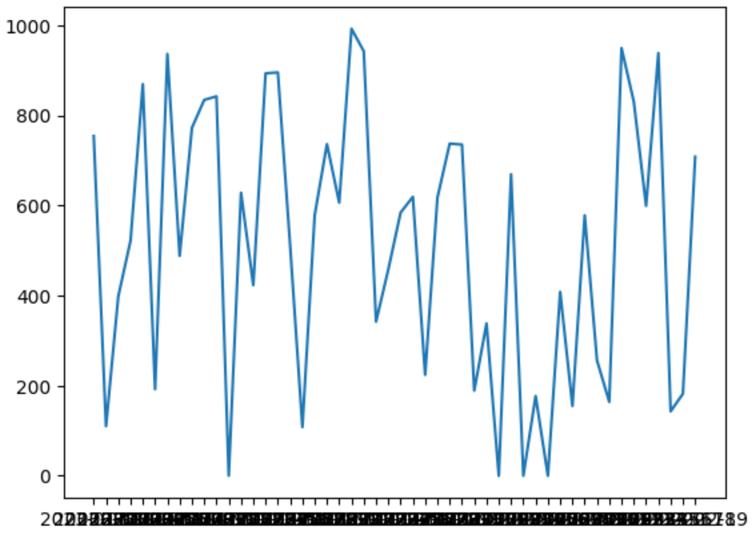
OR

```
total_sales_by_product.plot(kind='bar', color = 'blue')
```



4) Retrieving the trend of sales over time

```
sales_overtime = df.groupby('Date')['Sales'].sum().reset_index()


plt.plot(sales_overtime['Date'], sales_overtime['Sales'])
```


1000
800
600
400
200
0

SEABORN: Data Visualization 2

Seaborn is a Python data visualization library built on top of **matplotlib** (seaborn provides **high-level functions** for beautiful, statistical plots but, under the hood, it uses matplotlib to draw everything). It provides a high-level interface for creating attractive and informative statistical graphics. It helps creating complex visualization with just a few lines of code.

First step is to install Seaborn library using pip

```
%pip install seaborn
```

And import it

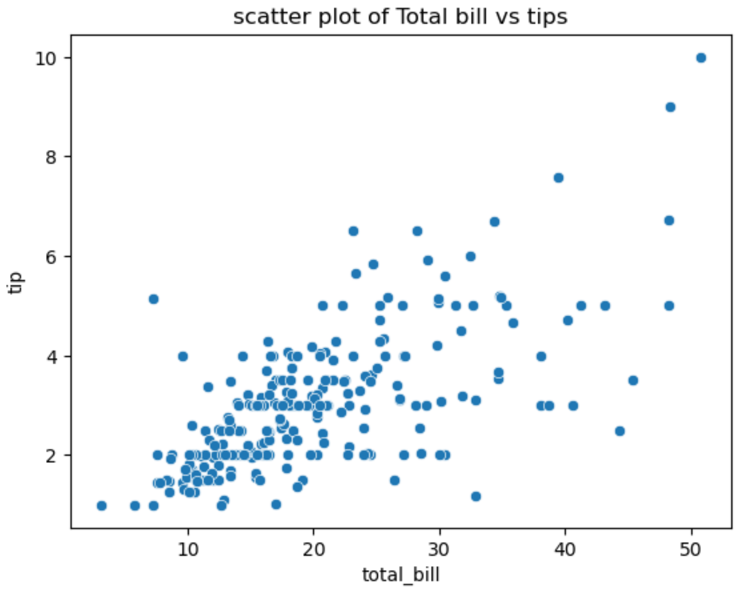
```
import seaborn as sns
```

NOTE: since seaborn uses matplotlib for the actual creation of the graphics, it's common use to import also matplotlib for better customization, which seaborn doesn't fully provide.

Basic plotting

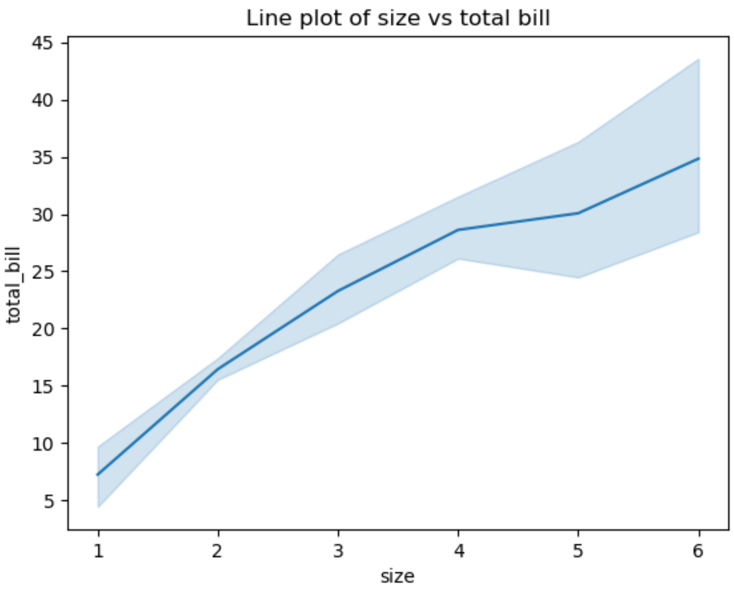
- **Scatter plot with seaborn (and matplotlib for customization)**

```
import matplotlib.pyplot as plt


sns.scatterplot(x = 'total_bill', y ='tip', data = tips)
plt.title('scatter plot of Total bill vs tips')
```



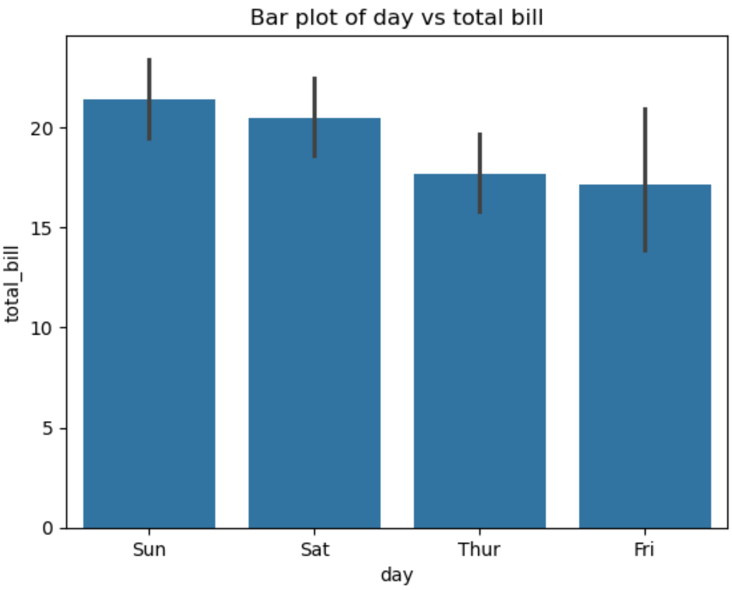
- **Line plot with seaborn**

```
sns.lineplot(x = 'size', y ='total_bill', data = tips)
plt.title('Line plot of size vs total bill')
```



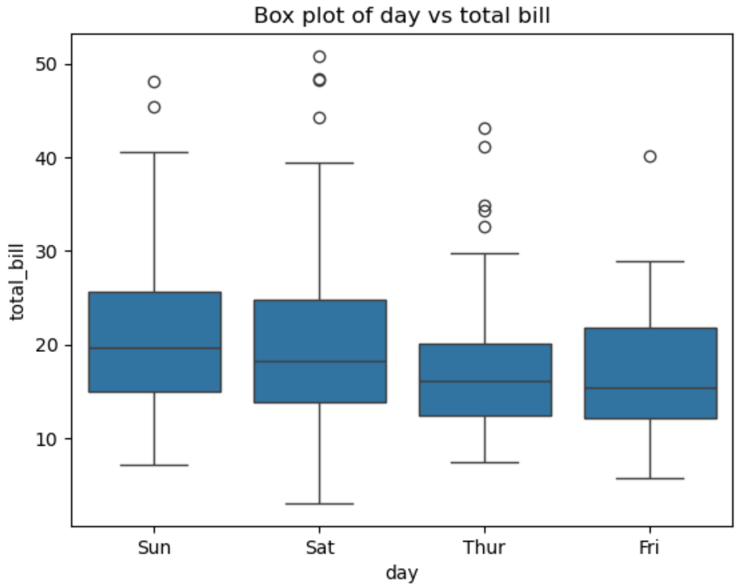
- **Categorical plot with seaborn (a bar plot with categories instead of values)**

```
sns.barplot(x = 'day', y = 'total_bill', data = tips)
plt.title('Bar plot of day vs total bill')
```



- **Boxplot with seaborn (bar plot with categories and percentiles)**

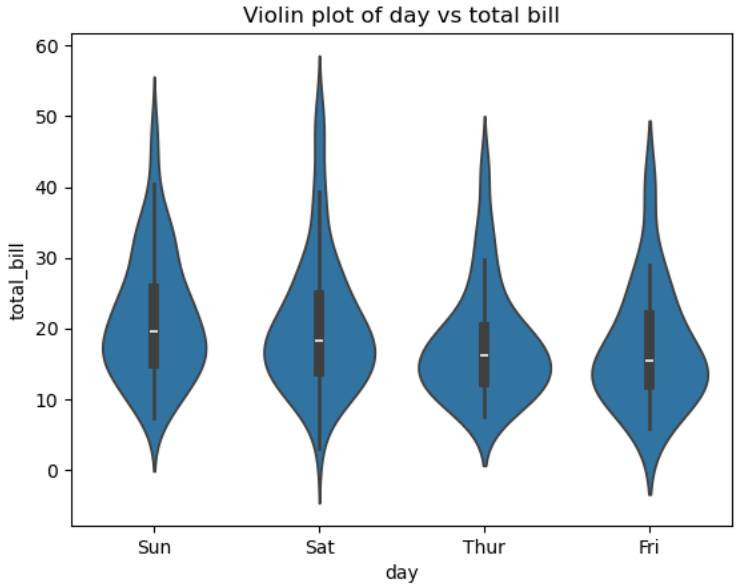
```
sns.boxplot(x = 'day', y = 'total_bill', data = tips)
plt.title('Box plot of day vs total bill')
```



NOTE: the horizontal lines of every "box" here represent thepercentiles of data, starting from zero (the lower horizontal line outside the box), then 25th percentile, 50th percentile, 75th percentile and 100th percentile. The circles outside the boxes are outliers.

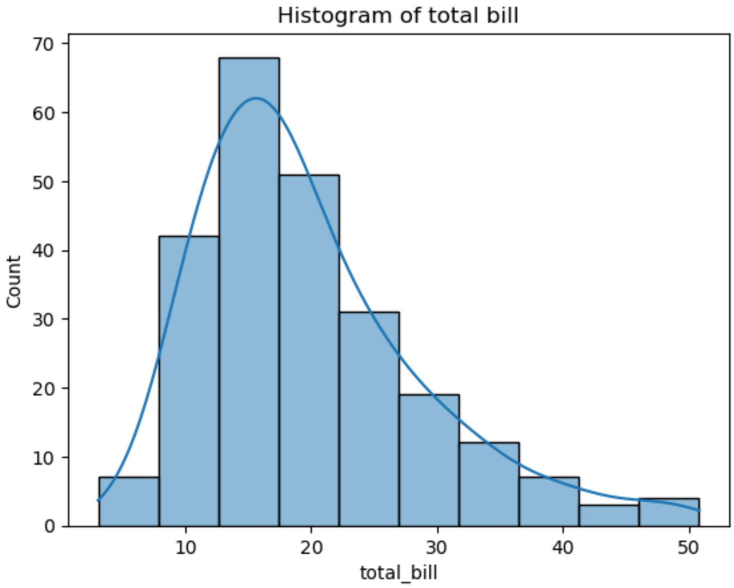
- **Violin plot with seaborn**

```
sns.violinplot(x = 'day', y = 'total_bill', data = tips)
plt.title('Violin plot of day vs total bill')
```



- **Histogram with categories, frequency of the categories and shape of the distribution**

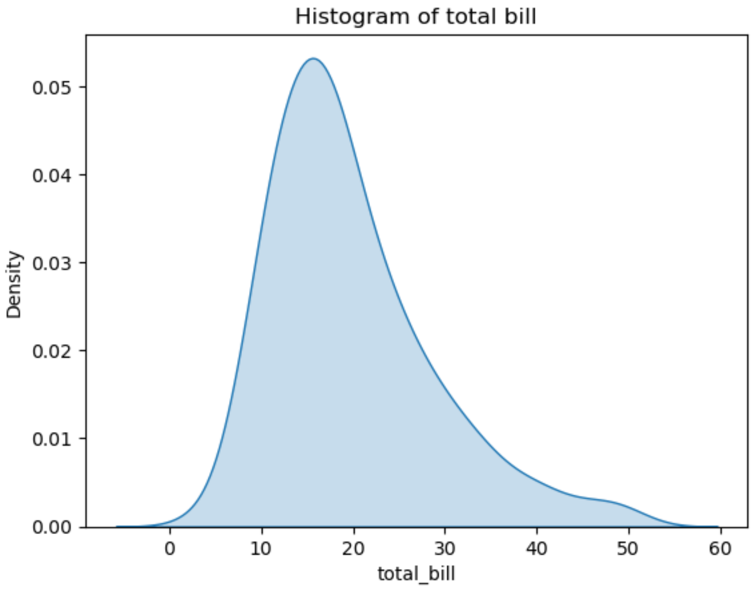
```
sns.histplot(tips['total_bill'], bins = 10, kde = True)
plt.title('Histogram of total bill')
```



NOTE: setting kde (stands for kernel density estimator) to True also gives back the shape of the distribution while setting it to False only gives back the distribution.

- **Distribution of categories**

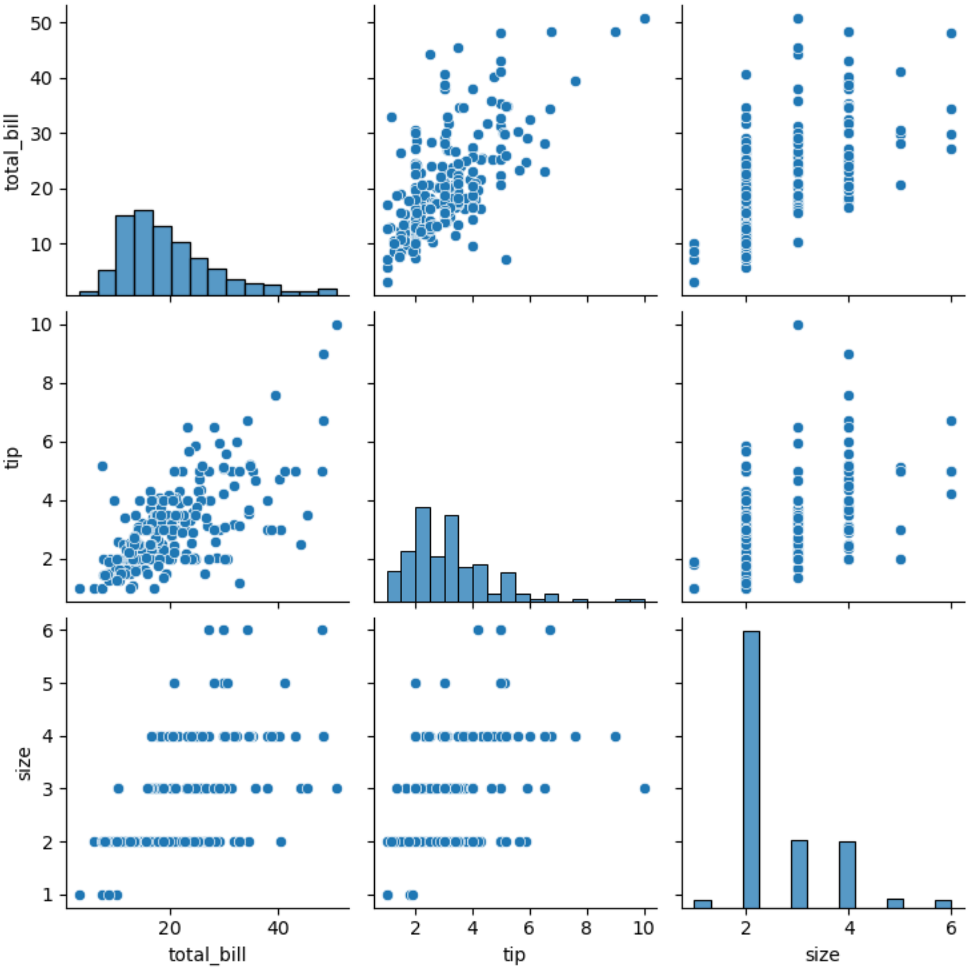
```
sns.kdeplot(tips['total_bill'], shade = True)
plt.title('Histogram of total bill')
```



NOTE: setting shade to True gives a colored area under the distribution.

- **Pair function, it pairs the combination of all the columns of the dataset in one output**

```
sns.pairplot(tips)
```



NOTE: this function pairs up in a single output and gives the relationships between the columns with data values in the dataset.

Advanced plotting

- **Correlation between data columns**

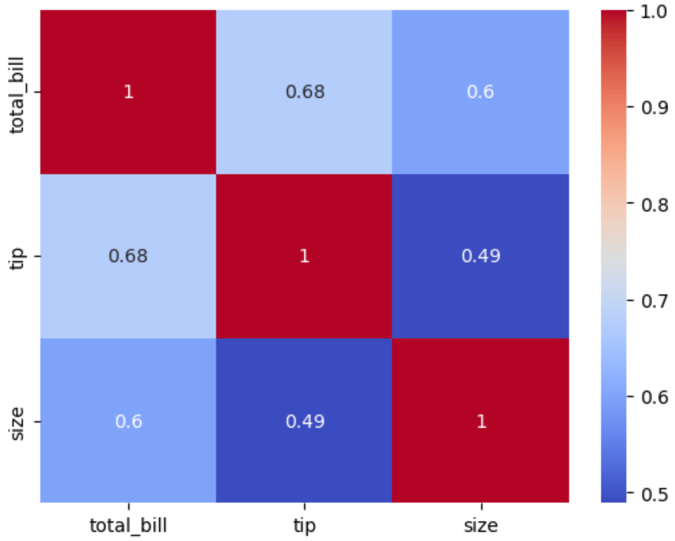
```
corr = tips[['total_bill', 'tip', 'size']].corr()
corr
```

Output:

	total_bill	tip	size
total_bill	1.000000	0.675734	0.598315
tip	0.675734	1.000000	0.489299
size	0.598315	0.489299	1.000000

- **Heatmap of correlation between columns presenting values**

```
sns.heatmap(corr, annot = True, cmap = 'coolwarm')
```

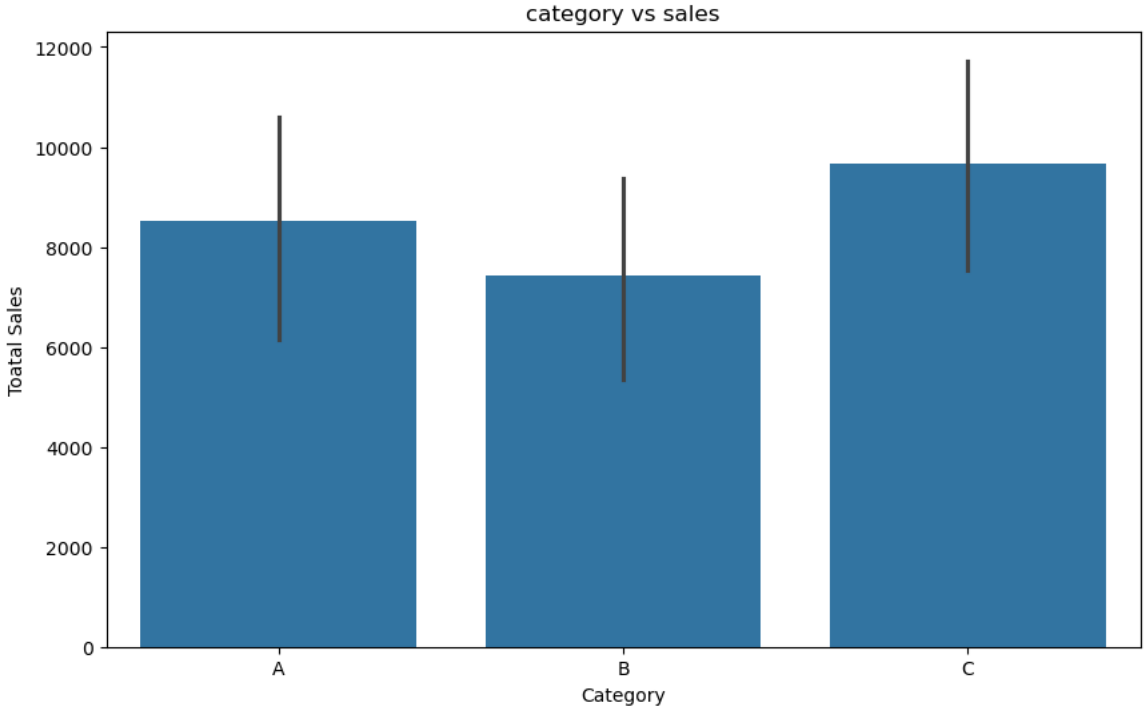


NOTE: as we can see from the heatmap, data with highest correlation are in warmer colors (correlation value of the same data is of course 1, therefore it has a red color), while data with less correlation are presented in cooler colors

Using seaborn with pandas

- **Plotting sum of sales by category with .barplot()**

```
plt.figure(figsize = (10,6))
sns.barplot(x = 'Category', y = 'Sales', data = df, estimator = sum)
plt.title('category vs sales')
plt.xlabel('Category')
plt.ylabel('Toatal Sales')
```



NOTE: in this section we use the functionalities of pandas and seaborn together. In the .barplot() function we used the aggregator ‘sum’ in the estimator functionality