Python Revision Notes

---

1. Using Python as a Calculator

Basic Arithmetic Operators

Operator	Meaning	Example	Result
`+`	Addition	`2 + 3`	`5`
`-`	Subtraction	`5 - 2`	`3`
`*`	Multiplication	`8 * 12`	`96`
`/`	Division	`100 / 7`	`14.285...`
`//`	Floor Division (drops decimals)	`100 // 7`	`14`
`%`	Remainder (Modulus)	`100 % 7`	`2`
`**`	Exponent	`5 ** 3`	`125`

Operator Precedence

- `**`
- `*`, `/`, `//`, `%`
- `+`, `-`
- Use `()` to control the order.

Example:
`((2 + 5) * (17 - 3)) / (4 ** 3)` → `1.53125`

---

2. Jupyter Notebook Basics

Creating Cells

- Insert cell below → **+ button** or **Esc + B**
- Change type:
  - Code → **Esc + Y**
  - Markdown → **Esc + M**

Running cells

- **Shift + Enter**
- Toolbar “Run”

Editing cells

- Double-click to edit Markdown
- `Help > Keyboard Shortcuts` to see all shortcuts

---

3. Solving Problems Using Python

You can directly type math expressions in a code cell and run them.
Examples:

- Population, cost, total, etc.
  (Your provided solved answers are all straightforward arithmetic.)

---

4. Variables

What is a variable?

A name that stores a value.

Example:

```
cost_of_ice_bag = 1.25
profit_margin = 0.2
number_of_bags = 500
```

Using variables

```
profit_per_bag = cost_of_ice_bag * profit_margin
total_profit = number_of_bags * profit_per_bag
```

Checking a variable

Just type its name in a cell.

Undefined variable error

Happens when you try to use a variable before defining it.

---

5. Comments

Uses

- Explain code
- Improve readability

Types

- Single-line: `#`
- Multi-line:
- `"""`
- `This is a comment`
- `spanning multiple lines`
- `"""`

---

6. print() Function

Used to display output:

```
print("Total profit is", total_profit)
```

---

7. Comparison Operators

Operator	Meaning
`==`	equal
`!=`	not equal
`>`	greater than
`<`	less than
`>=`	greater or equal
`<=`	less or equal

Result is always: `True` or `False`.

Example:

```
3 + 6 <= 9   # True
```

You can store results:

```
is_expensive = cost_of_ice_bag >= 10
```

---

8. Logical Operators

and

True only if both are True.

or

True if at least one is True.

not

Reverses True ↔ False.

Combining

Use parentheses to control logic:

```
(2 > 3 and 4 <= 5) or not (1 < 0)
```

---

9. Markdown Basics (for text cells)

Headers

```
# Header 1
## Header 2
### Header 3
```

Lists

- Bulleted:
  `* item`
- Numbered:
  `1. item`

Text Styling

- Bold → `**text**`
- Italic → `*text*`
- Link → `[title](url)`

Python Branching & Looping — Revision Notes

---

1. if, elif, else (Branching)

if statement

Runs code **only if condition is True**.

```
if condition:
    statements
```

- Indentation (4 spaces) is required.
- Example:

```
if a_number % 2 == 0:
    print("Even")
```

---

else statement

Executes when the if condition is **False**.

```
if condition:
    statements
else:
    statements
```

---

elif (else if)

Used for **multiple conditions**.
Only the **first True** condition runs, the rest are skipped.

```
if cond1:
    ...
elif cond2:
    ...
elif cond3:
    ...
else:
    ...
```

Example:

```
if n % 2 == 0:
    print("Divisible by 2")
elif n % 3 == 0:
    print("Divisible by 3")
elif n % 5 == 0:
    print("Divisible by 5")
```

Difference:

if + elif chain

- Stops after the first True condition.

Multiple if statements

- All are checked independently.

Using logical operators

You can combine conditions using:

- `and`
- `or`
- `not`

Example:

```
if n % 3 == 0 and n % 5 == 0:
    print("Divisible by 3 and 5")
```

Non-Boolean Conditions

Python converts values to boolean automatically:

Falsy values
→ `0`, `''`, `[]`, `{}`, `None`

Truthy values
→ Everything else

Example:

```
if "":
    print("True")
else:
    print("False")   # runs
```

Nested if

An if inside another if.

```
if n % 2 == 0:
    if n % 3 == 0:
        print("Divisible by 2 and 3")
```

Try to avoid heavy nesting for readability.

Shorthand if (Ternary Operator)

```
x = true_val if condition else false_val
```

Example:

```
parity = "even" if n % 2 == 0 else "odd"
```

Statements vs Expressions

- **Statements** = instructions (if, loops, assignment)
- **Expressions** = produce a value (math, function calls, variables)

Rule: only expressions can appear on RHS of `=`.

---

pass statement

Used when a block must not be empty.

```
if n > 0:
    pass
```

---

2. while Loops

Syntax

```
while condition:
    statements
```

Repeated until the condition becomes False.

Example: factorial

```
result = 1
i = 1
while i <= 100:
    result *= i
    i += 1
```

Common mistake → Infinite Loop

Happens when:

- condition never becomes False
- variable inside loop doesn't update

Stop using Kernel → Interrupt.

---

3. break and continue

break

Stops the loop immediately.

```
if i == 42:
    break
```

continue

Skips remaining statements in this iteration and jumps to next.

```
if i % 2 == 0:
    continue
```

---

4. Logging

Using print statements inside loops/conditions to check values during execution.

---

5. for Loops

Syntax

```
for value in sequence:
    statements
```

Works with: lists, tuples, strings, dictionaries, ranges.

Example:

```
for day in days:
    print(day)
```

Looping over dictionary

Default: loops over **keys**.

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

Looping over **values**:

```
for v in person.values():
    print(v)
```

Looping over **key-value pairs**:

```
for key, val in person.items():
    print(key, val)
```

---

6. range()

Used to create sequences of numbers.

- `range(n)` → 0 to n-1
- `range(a, b)` → a to b-1
- `range(a, b, step)`

Example:

```
for i in range(5):   # 0 to 4
    print(i)
```

---

7. enumerate()

Gives index + value together.

```
for i, val in enumerate(list):
    print(i, val)
```

---

8. break, continue, pass (in for loops too)

Same behavior as while loops.

---

9. Nested Loops

A loop inside another loop.

Example:

```
for day in days:
    for fruit in fruits:
        print(day, fruit)
```

Python Variables & Data Types

---

1. Variables

What is a variable?

A container that stores data.

Assignment

```
x = 10
color = "blue"
```

Multiple assignment

```
a, b, c = 1, 2, 3
```

Same value to many variables

```
x = y = z = 5
```

Reassigning

New value replaces old:

```
x = 10
x = x + 1     # 11
x += 1        # shorthand
```

---

2. Rules for Naming Variables

✓ Must start with a letter or _
✓ Can contain letters, digits, underscores
✓ Case-sensitive
✕ Cannot start with a number
✕ No spaces / special symbols (`$`, `-`, etc.)

---

3. Syntax Errors

Invalid variable names → `SyntaxError`

Example:

```
a variable = 3   # ✕
my-fav = 2       # ✕
3names = 3       # ✕
```

---

4. Built-in Data Types

Primitive Types

- **int**
- **float**
- **bool**
- **None**
- **str**

Non-Primitive / Containers

- **list**
- **tuple**
- **dict**

Check type:

```
type(x)
```

---

5. Integers (int)

- Whole numbers (positive/negative)
- Unlimited size
  Example:

```
a = 2024
b = -23423423423423423
```

---

6. Floats (float)

- Numbers with decimals:

```
pi = 3.14
a = 4.0
```

- Scientific notation:

```
1e-2   # 0.01
```

- Conversions:

```
float(5)    # 5.0
int(3.8)    # 3
```

- `/` always returns float
- `//` for integer division

---

7. Boolean (bool)

Values: **True**, **False**

Often from comparisons:

```
5 > 3   # True
```

Arithmetic:

```
True → 1
False → 0
```

Falsy values:
`False, 0, 0.0, None, "", [], (), {}, set(), range(0)`
Everything else → truthy.

---

8. None

Represents “no value”.

```
x = None
```

---

9. Strings (str)

Creating strings

```
"x"         # double quotes
'y'         # single quotes
"""multi
line"""
```

Escape characters

```
\"   # inside double quotes
\n   # newline
```

Length

```
len(s)
```

Indexing

```
s[0]
s[3]
s[2:5]    # slicing
```

Membership

```
"day" in "Saturday"
```

Concatenation

```
"Hello " + name
```

String methods

- `lower()`
- `upper()`
- `capitalize()`
- `replace(a, b)`
- `split(",")`
- `strip()`
- `format(...)`

format example

```
"Hello {}".format(name)
```

Convert to string

```
str(23)
```

---

10. Lists

Ordered, mutable collection.

Create list

```
fruits = ['apple', 'banana', 'cherry']
```

Indexing

```
fruits[0]
fruits[-1]
```

Slicing

```
fruits[1:3]
```

Modify list

```
fruits[1] = "mango"
```

Add elements

```
fruits.append("kiwi")
fruits.insert(1, "pear")
```

Remove elements

```
fruits.remove("banana")
fruits.pop(2)   # by index
fruits.pop()    # last
```

Check membership

```
"apple" in fruits
```

Concatenate lists

```
new_list = list1 + list2
```

Copying lists

```
b = a.copy()     # correct
b = a            # same list (wrong for copying)
```

Useful methods

- `reverse()`
- `sort()`
- `sort(reverse=True)`

---

11. Tuples

Ordered, **immutable** collection.

Create tuple

```
t = (1, 2, 3)
```

Or without parentheses:

```
t = 1, 2, 3
```

Single element tuple

```
(4,) or 4,
```

Access/Check values

```
t[0]
"apple" in t
```

Not allowed

✕ modify
✕ append
✕ remove

Use case

Multiple assignment:

```
x, y = (3, 4)
```

---

12. Dictionary (dict)

Unordered collection of **key–value pairs**.

Create dictionary

```
person = {
    "name": "John",
    "age": 30
}
```

Or:

```
person = dict(name="John", age=30)
```

Access values

```
person["name"]
person.get("age")
```

get() can include default:

```
person.get("address", "Unknown")
```

Add/update values

```
person["age"] = 31
person["city"] = "London"
```

Remove

```
person.pop("city")
```

Check key

```
"name" in person
```

View parts

```
person.keys()
person.values()
person.items()
```

Convert to list for indexing:

```
list(person.items())[0]
```

Copy dictionary

```
d2 = d1.copy()
```

Update from another dict

```
d1.update(d2)
```

Valid keys

✓ Strings
✓ Numbers
✓ Booleans
✕ Lists (not hashable)

✓ Python Functions

---

1. What is a Function?

A **function** is a reusable block of code that:

- Takes inputs (arguments/parameters)
- Performs operations
- Returns an output (optional)

Example (built-in):

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

---

2. Defining Your Own Function

Syntax:

```
def function_name():
    # indented body
```

Example:

```
def say_hello():
    print("Hello there!")
    print("How are you?")
```

Calling:

```
say_hello()
```

---

3. Function Arguments

Functions can accept one or more inputs.

Example:

```
def filter_even(number_list):
    result_list = []
    for number in number_list:
        if number % 2 == 0:
            result_list.append(number)
    return result_list
```

Usage:

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

---

4. Building Better Functions — Loan EMI Example

Goal: Compare EMI for two loan options.

Start simple:

```
def loan_emi(amount):
    emi = amount / 12
    print(emi)
```

---

5. Local Variables & Scope

Variables created **inside a function** are **local**.

They cannot be accessed outside the function.

Example:

```
emi = amount / duration   # inside function; not available outside
```

Trying:

```
emi
```

→ `NameError`

---

6. Returning Values

Use `return` to output a value:

```
def loan_emi(amount, duration):
    emi = amount / duration
    return emi
```

Store results:

```
emi1 = loan_emi(1260000, 96)
```

---

7. Optional Arguments (Default Parameters)

Add optional parameters with default values:

```
def loan_emi(amount, duration, down_payment=0):
    loan_amount = amount - down_payment
    return loan_amount / duration
```

If you don't pass `down_payment`, it defaults to 0.

---

8. Adding Interest (EMI Formula)

EMI formula:

```
EMI = P * r * (1+r)^n / ((1+r)^n -1)
```

(P = principal, r = monthly rate, n = months)

Implementation:

```
def loan_emi(amount, duration, rate, down_payment=0):
    loan_amount = amount - down_payment
    emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-
1)
    return emi
```

Required arguments must come **before** optional ones.

---

9. Named Arguments

Improves readability:

```
loan_emi(amount=1260000, duration=8*12, rate=0.1/12, down_payment=300000)
```

Arguments can be placed on multiple lines.

---

10. Modules & Library Functions

Modules = files containing Python code.

Import:

```
import math
```

Use module function:

```
math.ceil(1.2)  # → 2
```

Used to round EMI:

```
emi = math.ceil(emi)
```

---

11. Improving Function Using Other Functions

Updated EMI function:

```
def loan_emi(amount, duration, rate, down_payment=0):
    loan_amount = amount - down_payment
    emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-
1)
    emi = math.ceil(emi)
    return emi
```

Good practice:

- Functions should be short (<10 lines)
- Do one job only
- Reusable and readable

---

12. Real Problem Solving Using Functions

Example: Shaun's monthly loan payments (house + car):

Use the same function twice:

```
emi_house = loan_emi(...)
emi_car   = loan_emi(...)
print(emi_house + emi_car)
```

---

13. Exceptions and try–except

Errors during execution → **Exceptions**

Example: `ZeroDivisionError`
Occurs if `rate = 0`

Handle using try-except:

```
try:
    emi = complicated_formula
except ZeroDivisionError:
    emi = loan_amount / duration
```

This prevents program crash and ensures valid output.

---

14. Updated EMI Function with Error Handling

```
def loan_emi(amount, duration, rate, down_payment=0):
    loan_amount = amount - down_payment
    try:
        emi = loan_amount * rate * ((1+rate)**duration) /
(((1+rate)**duration)-1)
    except ZeroDivisionError:
        emi = loan_amount / duration
    emi = math.ceil(emi)
    return emi
```

---

15. Docstrings (Function Documentation)

Docstring = string written inside a function to document it.

Example:

```
def loan_emi(amount, duration, rate, down_payment=0):
    """Calculate EMI.

    amount: total cost including down payment
    duration: months
    rate: monthly interest rate
    down_payment: initial payment
    """
```

View docstring:

```
help(loan_emi)
```

✅ Interacting With the OS & Filesystem

---

1. The os Module

Python's `os` module allows you to:

- Interact with the Operating System
- Work with directories & files
- Query paths, create folders, list files, etc.

Import:

```
import os
```

---

2. Getting Current Working Directory

```
os.getcwd()
```

Returns the absolute path of where your Python program is currently running.

Example:

```
'/content'
```

---

3. Listing Files in a Directory

```
os.listdir('.')
os.listdir('/usr')
```

- Accepts **absolute** or **relative** paths.
- Returns a list of filenames in the directory.
- Does **not** include `.` or `..`

---

4. Creating Directories

```
os.makedirs('./data', exist_ok=True)
```

`exist_ok=True`
→ prevents error if the directory already exists.

Check if directory created:

```
'data' in os.listdir('.')
os.listdir('./data')  # should be empty initially
```

---

5. Downloading Files with urllib

```
import urllib.request
urllib.request.urlretrieve(url, './data/filename.txt')
```

Downloads data from a URL into local folder.

---

6. Opening Files

```
file = open('./data/loans1.txt', mode='r')
```

File Modes

Mode	Meaning
`'r'`	read (default)
`'w'`	write (overwrite)
`'x'`	create file only if not exists
`'a'`	append
`'b'`	binary mode
`'t'`	text mode (default)
`'+'`	read + write

---

7. Reading File Contents

```
content = file.read()
print(content)
```

Important: Close the file when done:

```
file.close()
```

Accessing after closing → error:

```
ValueError: I/O operation on closed file
```

---

8. Using with (Auto-close Files)

Preferred way:

```
with open('./data/loans2.txt') as f:
    data = f.read()
```

After the block ends, file is **automatically closed**.

Trying:

```
f.read()
```

→ `ValueError: I/O operation on closed file.`

---

9. Reading a File Line-by-Line

```
with open('./data/loans3.txt') as f:
    lines = f.readlines()
```

Result is a list — each line is a string ending with `\n`.

---

10. Understanding CSV Files

CSV = Comma-Separated Values
Each line → record
Each value separated by `,`

Example:

```
amount,duration,rate,down_payment
100000,36,0.08,20000
```

Header row → names of columns.

---

11. Processing CSV Manually

Steps to parse:

1. Read lines
2. Extract header
3. Convert values to floats
4. Combine headers + values
5. Produce list of dictionaries

Helper Function 1: Parse Headers

```
def parse_headers(line):
    return line.strip().split(',')
```

Uses:

- `.strip()` → removes whitespace + `\n`
- `.split(',')` → convert to list

Helper Function 2: Parse Values

Handles missing values (`''`) by converting to `0.0`.

```
def parse_values(line):
    values = []
    for item in line.strip().split(','):
        if item == '':
            values.append(0.0)
        else:
            values.append(float(item))
    return values
```

Helper Function 3: Combine Headers + Values

```
def create_item_dict(values, headers):
    d = {}
    for value, header in zip(values, headers):
        d[header] = value
    return d
```

`zip` pairs elements of two lists positionally.

---

12. Final Function: Reading Any CSV File

```
def read_csv(path):
    result = []
    with open(path, 'r') as f:
        lines = f.readlines()
        headers = parse_headers(lines[0])
        for line in lines[1:]:
            values = parse_values(line)
            item = create_item_dict(values, headers)
            result.append(item)
    return result
```

Generic → works with any CSV file.

---

13. Using the Loan EMI Function on CSV Data

Use `loan_emi` function to compute EMI for each loan dictionary.

Example:

```
for loan in loans:
    loan['emi'] = loan_emi(loan['amount'], loan['duration'], loan['rate']/12,
loan['down_payment'])
```

Adds new `'emi'` key to each loan.

---

14. Turning EMI Computation Into a Function

```
def compute_emis(loans):
    for loan in loans:
        loan['emi'] = loan_emi(
            loan['amount'],
            loan['duration'],
            loan['rate']/12,
            loan['down_payment']
        )
```

---

15. Writing Data Back to Files

Use write mode:

```
with open('./data/emis2.txt', 'w') as f:
    f.write("...")
```

---

16. Generic Function to Write CSV Files

```
def write_csv(items, path):
    with open(path, 'w') as f:
        if len(items) == 0:
            return
```

```
        headers = list(items[0].keys())
        f.write(','.join(headers) + '\n')

        for item in items:
            values = []
            for header in headers:
                values.append(str(item.get(header, "")))
            f.write(','.join(values) + "\n")
```

Supports:

- Any number of columns
- Writes header + values
- Converts values to strings

---

17. Complete Workflow (Automated Processing for Multiple Files)

```
for i in range(1,4):
    loans = read_csv(f'./data/loans{i}.txt')
    compute_emis(loans)
    write_csv(loans, f'./data/emis{i}.txt')
```

This:

- Reads all loan files
- Computes EMIs
- Writes results to new CSV files

---

18. Key Concepts Summary

✓ `os` module

→ interacting with filesystem

✓ `with open()`

→ auto-closing files

✓ Reading files

→ `.read()`, `.readlines()`

✓ Parsing CSV manually

→ `split`, `strip`, `zip`

✓ Writing CSV

→ `.write()` with formatted strings

✓ Automation

→ Functions like `read_csv`, `compute_emis`, `write_csv`

✓ Reusability

→ Build small reusable helper functions

✓ Working With Numerical Data -- (NumPy)

---

1. Why Numerical Data Needs NumPy

Most real-world datasets contain **numerical values**:

- Stock prices
- Sensor measurements
- Sales figures
- Statistical datasets
- Weather & climate data

Python lists are **not efficient** for numerical computation.
NumPy provides:

- Fast array operations
- Vectorized mathematical functions
- Multi-dimensional arrays
- Tools for matrix algebra, broadcasting, indexing

---

2. Using Climate Data to Predict Apple Yield

We want to model:

```
yield_of_apples = w1 * temperature + w2 * rainfall + w3 * humidity
```

Where:

- `w1`, `w2`, `w3` are known weights based on past statistical analysis.

Example:

```
w1, w2, w3 = 0.3, 0.2, 0.5
```

---

3. Basic Python Implementation (Using Lists)

Climate Data (for regions):

```
kanto = [73, 67, 43]
johto = [91, 88, 64]
hoenn = [87, 134, 58]
sinnoh = [102, 43, 37]
unova = [69, 96, 70]
```

Weight Vector:

```
weights = [0.3, 0.2, 0.5]
```

Function to calculate crop yield:

```
def crop_yield(region, weights):
    result = 0
    for x, w in zip(region, weights):
        result += x * w
    return result
```

This is the "dot product" of two vectors.

---

4. Introducing NumPy

To speed up numerical operations:

```
import numpy as np
```

Convert lists → NumPy arrays:

```
kanto = np.array([73, 67, 43])
weights = np.array([0.3, 0.2, 0.5])
```

Check type:

```
type(kanto)           # numpy.ndarray
```

Indexing:

```
kanto[2]     # 43
```

---

5. Performing Dot Product using NumPy

Method 1: Built-in dot

```
np.dot(kanto, weights)
```

Method 2: Element-wise multiply + sum

```
(kanto * weights).sum()
```

NumPy automatically performs element-wise operations.

---

6. Why Use NumPy? (Major Benefits)

✓ Vectorized Computations

No loops needed.

✓ Faster Execution

NumPy is implemented in **C and C++**, so operations run ~100x faster.

Example speed test for 1 million values:

- Python loop → ~124 ms
- NumPy dot → ~2 ms

---

7. Multi-Dimensional NumPy Arrays

Climate data for **multiple regions**:

```
climate_data = np.array([
    [73, 67, 43],
    [91, 88, 64],
    [87, 134, 58],
    [102, 43, 37],
    [69, 96, 70]
])
```

This is a **2D array (matrix)**.

Check shape:

```
climate_data.shape     # (5, 3)
weights.shape          # (3,)
```

NumPy supports **n-dimensional arrays**:

```
arr3.shape   # (2, 2, 3)
```

Check datatype:

```
climate_data.dtype    # int64
weights.dtype         # float64
```

If even one element is float → whole array becomes float.

---

8. Matrix Multiplication (Predicting Yields for All Regions)

Matrix (5×3) × Vector (3×1) → Vector (5×1)

Using:

```
np.matmul(climate_data, weights)
```

Or simply:

```
climate_data @ weights
```

---

9. Reading CSV Files Using NumPy

We download a CSV file with 10,000 rows:

```
np.genfromtxt('climate.txt', delimiter=',', skip_header=1)
```

- `delimiter=','` → split on commas
- `skip_header=1` → ignore header row

Shape:

```
(10000, 3)
```

---

10. Predicting Yields for the Whole Dataset

```
yields = climate_data @ weights
```

Shape:

```
yields.shape   # (10000,)
```

---

11. Combining Arrays Using np.concatenate

To add yields as a new column:

```
yields = yields.reshape(10000, 1)
climate_results = np.concatenate((climate_data, yields), axis=1)
```

Important:

- Use `axis=1` → concatenate column-wise
- Shapes must match except on the concatenated axis
- Reshape 1D to 2D using `.reshape()`

---

12. Saving Arrays to a File Using NumPy

```
np.savetxt(
```

```
    'climate_results.txt',
    climate_results,
    fmt='%.2f',
    header='temperature,rainfall,humidity,yield_apples',
    comments=''
)
```

Writes a CSV-like file.

---

13. Arithmetic Operations & Broadcasting

NumPy supports:

- Adding scalars
- Element-wise operations between arrays
- Broadcasting between arrays of compatible shapes

Examples:

```
arr2 + 3
arr3 - arr2
arr2 * arr3
arr2 % 4
```

Broadcasting Example:

```
arr2.shape → (3, 4)
arr4.shape → (4,)
arr2 + arr4   # VALID → arr4 is broadcast
```

Invalid broadcast example:

```
arr5.shape → (2,)
arr2 + arr5   # ERROR → shapes incompatible
```

---

14. Comparison Operators

Element-wise comparison:

```
arr1 == arr2
arr1 >= arr2
arr1 != arr2
```

Counting matches:

```
(arr1 == arr2).sum()
```

(Remember: `True → 1`, `False → 0`)

---

15. Indexing & Slicing in Multi-Dimensional Arrays

General form:

```
array[row_index, column_index]
```

Examples:

```
arr3[1, 1, 2]            # specific element
arr3[1:, 0:1, :2]        # slice
arr3[1:, 1, 3]           # mixing indices & slices
arr3[:2, 1]              # fewer indexes gives subarray
```

Error if too many indices:

```
arr3[1,3,2,1]  # IndexError
```

---

16. Creating Arrays with NumPy Functions

Zero matrix:

```
np.zeros((3,2))
```

Ones matrix:

```
np.ones([2,2,3])
```

Identity matrix:

```
np.eye(3)
```

Random values:

```
np.random.rand(5)       # uniform distribution
np.random.randn(2,3)    # normal distribution
```

Filled array:

```
np.full([2,3], 42)
```

Range values:

```
np.arange(10, 90, 3)
np.linspace(3, 27, 9)
```

---

17. Essential NumPy Categories

Mathematics

- `np.sum`, `np.exp`, `np.round`

Array Manipulation

- `np.reshape`
- `np.stack`
- `np.concatenate`
- `np.split`

Linear Algebra

- `np.dot`
- `np.matmul`
- `np.transpose`
- `np.eigvals`

Statistics

- `np.mean`, `np.median`, `np.std`, `np.max`

---

18. Finding NumPy Functions

Use:

- Google search
- NumPy docs: https://numpy.org/doc/stable/reference/routines.html
- `help(np.function_name)`

✅ PANDAS

---

1 Reading a CSV File Using Pandas

What is Pandas?

- A Python library used for working with **tabular data** (similar to Excel sheets).
- Supports reading data from many formats: **CSV, Excel, HTML tables, JSON, SQL**, etc.

About CSV Files

- CSV = *Comma Separated Values*
- Each line represents a **record**
- Each record contains **fields separated by commas**
- Mostly used for tabular numeric or text data

Example data (Italy COVID Daywise):

```
date,new_cases,new_deaths,new_tests
2020-04-21,2256,454,28095
2020-04-22,2729,534,44248
...
```

Downloading a CSV File

```
from urllib.request import urlretrieve

urlretrieve(
  'https://hub.jovian.ml/wp-content/uploads/2020/09/italy-covid-daywise.csv',
  'italy-covid-daywise.csv'
)
```

Reading the File With Pandas

```
import pandas as pd
covid_df = pd.read_csv('italy-covid-daywise.csv')
```

Understanding the Loaded DataFrame

- `type(covid_df)` → shows it's a DataFrame
- Displaying the DataFrame shows 248 rows × 4 columns:
  - **date**
  - **new_cases**
  - **new_deaths**
  - **new_tests**

Observations:

- Data from **Dec 31, 2019 → Sept 3, 2020**
- new_tests has many **NaN** values because Italy started reporting tests later.

Useful Functions

`.info()`

- Shows columns, data types, missing values, memory usage.

`.describe()`

- Summary statistics (mean, std, min, max…)

`.columns`

- List of column names.

`.shape`

- Returns `(rows, columns)`.

---

2 Retrieving Data From DataFrames

Pandas DataFrame = **Dictionary of Lists**
Example:

```
covid_data_dict = {
  'date': [...],
  'new_cases': [...],
```

```
}
```

Accessing Columns

```
covid_df['new_cases']
covid_df.new_cases   # Only works if name contains no spaces
```

- Returns a **Series**

Accessing Single Values

```
covid_df['new_cases'][243]
covid_df.at[243, 'new_cases']
```

Accessing Multiple Columns

```
cases_df = covid_df[['date', 'new_cases']]
```

⚠ This creates a **view**, not a copy.
To make a real copy:

```
copy_df = covid_df.copy()
```

Accessing Rows

```
covid_df.loc[243]     # returns entire row as a Series
```

Head/Tail/Sample

```
covid_df.head(5)
covid_df.tail(5)
covid_df.sample(10)
```

Finding First Non-Missing Value

```
covid_df.new_tests.first_valid_index()
```

---

3 Analyzing Data (Basic Calculations)

Total Cases / Deaths

```
total_cases = covid_df.new_cases.sum()
total_deaths = covid_df.new_deaths.sum()
```

Death Rate

```
death_rate = total_deaths / total_cases
```

Total Tests

Before April 19 → 935,310 tests already done.

```
total_tests = 935310 + covid_df.new_tests.sum()
```

Positive Rate

```
positive_rate = total_cases / total_tests
```

---

4 Querying & Sorting Rows

Boolean Filtering

```
covid_df[covid_df.new_cases > 1000]
```

Complex Conditions

```
covid_df[covid_df.new_cases / covid_df.new_tests > positive_rate]
```

Adding a New Column

```
covid_df['positive_rate'] = covid_df.new_cases / covid_df.new_tests
```

Deleting Columns

```
covid_df.drop(columns=['positive_rate'], inplace=True)
```

Sorting Rows

Most new cases

```
covid_df.sort_values('new_cases', ascending=False)
```

Most deaths

```
covid_df.sort_values('new_deaths', ascending=False)
```

Least cases

```
covid_df.sort_values('new_cases').head()
```

Fixing Incorrect Values

Example: −148 cases on June 20
Replace with average of neighbors:

```
covid_df.at[172, 'new_cases'] = (
  covid_df.at[171, 'new_cases'] + covid_df.at[173, 'new_cases']
) / 2
```

---

5 Working With Dates

Convert to datetime

```
covid_df['date'] = pd.to_datetime(covid_df.date)
```

Extract Year/Month/Day/Weekday

```
covid_df['year'] = covid_df.date.dt.year
covid_df['month'] = covid_df.date.dt.month
covid_df['day'] = covid_df.date.dt.day
covid_df['weekday'] = covid_df.date.dt.weekday
```

Example Aggregation (Month of May)

```
covid_df[covid_df.month == 5][['new_cases', 'new_deaths', 'new_tests']].sum()
```

Average Cases on Sundays

```
covid_df[covid_df.weekday == 6].new_cases.mean()
```

---

6 Grouping & Aggregation

Group by Month

```
covid_month_df =
covid_df.groupby('month')[['new_cases','new_deaths','new_tests']].sum()
```

Mean Aggregation

```
covid_df.groupby('month')[['new_cases','new_deaths','new_tests']].mean()
```

Cumulative Sums

```
covid_df['total_cases'] = covid_df.new_cases.cumsum()
covid_df['total_deaths'] = covid_df.new_deaths.cumsum()
covid_df['total_tests'] = covid_df.new_tests.cumsum() + initial_tests
```

---

7 Merging Data (Country Details)

Load locations.csv

```
locations_df = pd.read_csv('locations.csv')
```

Fetch Italy row:

```
locations_df[locations_df.location == 'Italy']
```

Add Location Column to covid_df

```
covid_df['location'] = 'Italy'
```

Merge

```
merged_df = covid_df.merge(locations_df, on='location')
```

Calculate Per-Million Metrics

```
merged_df['cases_per_million']  = merged_df.total_cases  * 1e6 /
merged_df.population
merged_df['deaths_per_million'] = merged_df.total_deaths * 1e6 /
merged_df.population
merged_df['tests_per_million']  = merged_df.total_tests  * 1e6 /
merged_df.population
```

---

8 Writing Data Back to Files

Select Useful Columns

```
result_df = merged_df[[
  'date','new_cases','total_cases','new_deaths','total_deaths',
  'new_tests','total_tests','cases_per_million','deaths_per_million',
  'tests_per_million'
]]
```

Save to CSV

```
result_df.to_csv('results.csv', index=False)
```

---

9 Bonus: Plotting With Pandas

Plot new cases

```
result_df.new_cases.plot()
```

Set Date as Index

```
result_df.set_index('date', inplace=True)
```

Plot multiple columns

```
result_df.new_cases.plot()
result_df.new_deaths.plot()
```

Plot death rate

```
(result_df.total_deaths / result_df.total_cases).plot()
```

Monthly Bars

```
covid_month_df.new_cases.plot(kind='bar')
covid_month_df.new_tests.plot(kind='bar')
```

📊 DATA VISUALIZATION

1️⃣ Introduction to Data Visualization

What is Data Visualization?

Data visualization is the **graphical representation of data** using charts, plots, and images.

Why it is Important

- Makes complex data easy to understand
- Reveals **patterns, trends, and relationships**
- Helps in **decision-making**
- Essential in **Data Analysis, Machine Learning, and AI**

Python Libraries for Visualization

1. Matplotlib

- Core (low-level) visualization library
- Highly customizable
- Foundation for many other libraries

import matplotlib.pyplot as plt

- `plt` is an alias for convenience

2. Seaborn

- High-level library built on Matplotlib
- Requires **less code**
- Produces **better-looking default plots**

import seaborn as sns

%matplotlib inline

%matplotlib inline

Purpose

- Ensures plots appear **inside Jupyter Notebook**
- Without it, plots may open in separate windows

---

2 Line Chart

Purpose

- Shows **change over time**
- Connects data points using straight lines

Basic Line Chart

yield_apples = [0.895, 0.91, 0.919, 0.926, 0.929, 0.931]

plt.plot(yield_apples)

- X-axis defaults to index values (0,1,2…)
- Jupyter outputs a `Line2D` object
- Add `;` to suppress output

plt.plot(yield_apples);

Customizing X-axis

years = [2010, 2011, 2012, 2013, 2014, 2015]
plt.plot(years, yield_apples)

- Makes the plot **more meaningful**

Axis Labels

plt.xlabel("Year")
plt.ylabel("Yield (tons per hectare)")

Plotting Multiple Lines

years = range(2000, 2012)
plt.plot(years, apples)
plt.plot(years, oranges)

- Used for **comparison** between datasets

Title and Legend

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges'])

- Legend identifies each line

Line Markers

Markers highlight individual data points.

plt.plot(years, apples, marker='o')
plt.plot(years, oranges, marker='x')

Common markers:

- `o` → circle
- `x` → cross
- `s` → square
- `^` → triangle

Styling Lines & Markers

Argument	Meaning
`c` or `color`	Line color
`ls` or `linestyle`	Solid / dashed
`lw`	Line width
`marker`	Marker type
`ms`	Marker size
`mec`	Marker edge color
`mew`	Marker edge width
`mfc`	Marker fill color
`alpha`	Transparency

plt.plot(years, apples, marker='s', c='b', ls='-', lw=2, ms=8, mec='navy')

---

fmt Shorthand

Format: `"[marker][line style][color]"`

plt.plot(years, apples, 's-b')
plt.plot(years, oranges, 'o--r')

Only markers:

plt.plot(years, oranges, 'or')

Figure Size

plt.figure(figsize=(12, 6))

Improving Style with Seaborn

sns.set_style("whitegrid")

Available styles:

- white
- dark
- whitegrid
- darkgrid
- ticks

Global Customization (rcParams)

```
import matplotlib
matplotlib.rcParams['font.size'] = 14
matplotlib.rcParams['figure.figsize'] = (9, 5)
```

Applies settings to **all future plots**

---

3 Scatter Plot

Purpose

- Shows relationship between **two numerical variables**
- Each point represents one observation

Dataset: Iris

```
flowers_df = sns.load_dataset("iris")
```

Basic Scatter Plot

```
sns.scatterplot(flowers_df.sepal_length,
flowers_df.sepal_width)
```

Scatter Plot with Hue & Size

```
sns.scatterplot(flowers_df.sepal_length,
flowers_df.sepal_width,
hue=flowers_df.species,
s=100)
```

Interpretation:

- Setosa → short, wide sepals
- Virginica → long, narrow sepals
- Versicolor → overlaps

Using Column Names

```
sns.scatterplot('sepal_length', 'sepal_width',
hue='species', s=100,
data=flowers_df)
```

---

4 Histogram

Purpose

- Shows **frequency distribution**
- Divides data into **bins**

Basic Histogram

```
plt.hist(flowers_df.sepal_width)
```

Controlling Bins

```
plt.hist(flowers_df.sepal_width, bins=5)
plt.hist(flowers_df.sepal_width, bins=np.arange(2,5,0.25))
```

Multiple Histograms

```
plt.hist(setosa_df.sepal_width, alpha=0.4)
plt.hist(versicolor_df.sepal_width, alpha=0.4)
```

- `alpha` controls transparency

Stacked Histogram

plt.hist([setosa_df.sepal_width,
versicolor_df.sepal_width,
virginica_df.sepal_width],
stacked=True)

---

5 Bar Chart

Purpose

- Displays **categorical comparisons**

Basic Bar Chart

plt.bar(years, oranges)

Stacked Bar Chart

plt.bar(years, apples)
plt.bar(years, oranges, bottom=apples)

Seaborn Bar Plot (Mean Values)

sns.barplot('day', 'total_bill', data=tips_df)

- Automatically calculates **mean**
- Error bars show variability

Bar Plot with Hue

sns.barplot('day', 'total_bill', hue='sex', data=tips_df)

Horizontal Bar Chart

sns.barplot('total_bill', 'day', hue='sex', data=tips_df)

---

6 Heatmap

Purpose

- Shows **2D data** using colors

Dataset

flights_df = sns.load_dataset("flights").pivot("month", "year", "passengers")

Basic Heatmap

sns.heatmap(flights_df)

Annotated Heatmap

sns.heatmap(flights_df, annot=True, fmt="d", cmap='Blues')

`Annot` -Show numbers in cells (`True`/`False`)
`Fmt`- Format of the annotations (`"d"` = integer, `"0.2f"` = 2 decimal places)
`Cmap`- Color palette for the heatmap (`'Blues'`, `'Reds'`, etc.)

---

7 Images in Matplotlib

Load & Display Image

```
img = Image.open('chart.jpg')
img_array = np.array(img)
plt.imshow(img)
plt.axis('off')
```

Cropping Image

plt.imshow(img_array[125:325, 105:305])

---

8 Subplots

Create Grid

fig, axes = plt.subplots(2, 3, figsize=(16, 8))

Access

axes[0,0].plot(...)
axes[1,2].imshow(img)

---

9 Pair Plot (Seaborn)

Purpose

- Automatic **EDA visualization**
- Shows relationships between all numeric columns

sns.pairplot(flowers_df, hue='species')

Best used for **exploratory data analysis**.

📘 Advanced Python Function Concepts

---

1 Functions with an Arbitrary Number of Arguments (*args and **kwargs)

Python allows functions to accept a **variable number of arguments**, which is useful when you do not know beforehand how many values will be passed.

◆ *args (Arbitrary Positional Arguments)

- `*args` collects **any number of positional arguments** into a **tuple**.
- Useful when you want the function to accept multiple values flexibly.

Example

```
def add_numbers(*args):
    return sum(args)

print(add_numbers(1, 2, 3))       # 6
print(add_numbers(10, 20))        # 30
```

What happens?

- All arguments become:
  `args = (1, 2, 3)`
- You can loop over them or process them as needed.

◆ **kwargs (Arbitrary Keyword Arguments)

- `**kwargs` collects any number of **keyword arguments** into a **dictionary**.
- Useful for functions that accept flexible named parameters.

Example

```
def print_details(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_details(name="Alice", age=22, role="Developer")
```

Output:

```
name: Alice
age: 22
role: Developer
```

◆ Using both together

```
def demo(a, *args, **kwargs):
    print(a)
    print(args)
    print(kwargs)

demo(5, 10, 20, 30, x=1, y=2)
```

Output:

```
5
(10, 20, 30)
{'x': 1, 'y': 2}
```

📝 **Rule:**
`a, *args, **kwargs` must appear in this order.

---

2 Defining Functions Inside Functions (Inner Functions & Closures)

Python allows defining functions **inside other functions**.
These are called:

- **Inner functions**
- **Nested functions**
- **Closures** (when inner functions remember variables from outer functions)

◆ Inner Function Example

```
def outer():
    def inner():
        print("This is inner.")
    inner()

outer()
```

Why use inner functions?

- To logically group helper functions used only inside another function.
- To implement **closures**.

◆ Closures

A **closure** is created when an inner function remembers the values of variables from its outer function **even after the outer function has finished executing**.

Example

```
def multiplier(n):
    def inner(x):
        return x * n
    return inner

double = multiplier(2)
print(double(10))  # Output: 20
```

Here:

- `inner()` remembers `n=2`
- Even though `multiplier()` has ended, the value persists

💡 Closures allow function factories, encapsulation, and decorators.

---

3 A Function That Invokes Itself (Recursion)

Recursion = a function calling itself.

Used for:

- Factorials
- Fibonacci sequence
- Tree/graph traversal
- Divide & conquer algorithms

◆ Example: Factorial

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)
```

Key Requirements for Recursion

1. **Base case** → stops recursion
2. **Recursive case** → calls itself

◆ Example: Fibonacci

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

⚠ Recursion must be used carefully

- Too deep recursion can cause `RecursionError`
- Sometimes slower than loops

---

4 Functions That Accept Other Functions or Return Functions

In Python, **functions are first-class citizens**, meaning they can be:

✓ Assigned to variables
✓ Passed as arguments
✓ Returned from functions
✓ Stored in data structures

◆ Passing a function as argument

```
def apply(func, value):
    return func(value)

print(apply(abs, -5))   # 5
```

◆ Returning a function

```
def operation():
    def add(a, b):
        return a + b
    return add

adder = operation()
print(adder(3, 4))   # 7
```

Use Cases

- Callback functions
- Event handling
- Higher-order functions
- Function factories
- Decorators

5️⃣ Functions That Enhance Other Functions (Decorators)

A **decorator** is a function that:

- Takes another function as input
- Adds some extra functionality
- Returns a modified/enhanced function

Decorators are used for:

- Logging
- Authentication
- Timing
- Input validation
- Access control
- Caching

◆ Basic Decorator Example

```
def my_decorator(func):
    def wrapper():
        print("Before function runs")
        func()
        print("After function runs")
    return wrapper
```

Using the decorator

```
@my_decorator
def greet():
    print("Hello!")
```

```
greet()
```

Output:

```
Before function runs
Hello!
After function runs
```

How Decorators Work Internally

- The `@decorator_name` syntax is shorthand for:

```
greet = my_decorator(greet)
```

◆ Decorator with Arguments

```
def debug(func):
    def wrapper(*args, **kwargs):
        print("Called with:", args, kwargs)
        return func(*args, **kwargs)
    return wrapper

@debug
def add(a, b):
    return a + b

print(add(3, 4))
```

Real-Life Uses

- `@staticmethod`
- `@classmethod`
- `@property`
- Flask route decorator: `@app.route("/")`
- Django views decorators
- Logging & timing decorators