



Python Revision Notes

Core Python • NumPy • Pandas
Quick-reference revision guide for students

1. Python Core

1.1 Data Types

A data type tells Python what kind of value a variable holds — a number, text, or true/false. Python is dynamically typed, meaning you never have to declare the type yourself; Python figures it out automatically when you assign a value.

💡 Tip: Use `type()` on any variable to instantly check its data type.

Integer (int)

Integers are whole numbers without a decimal point. You can reassign a variable to a new value at any time — Python updates the type automatically.

```
a = 10
a = 20          # re-assigning
print(a)        # 20
print(type(a))  # <class 'int'>
▶ Output: 20 | <class 'int'>
```

Float (float)

Floats are numbers that include a decimal point. Use floats whenever you need precision — such as prices, measurements, or scientific calculations.

```
b = 10.5
print(b)          # 10.5
print(type(b))    # <class 'float'>
▶ Output: 10.5 | <class 'float'>
```

String (str)

Strings store text and can be defined with single, double, or triple quotes. Triple quotes are ideal for multi-line text. All three produce the same str type.

```
name  = 'Hi AIBees'      # single quotes
name2 = "Welcome to AIBees" # double quotes
name3 = '''Multi-line text''' # triple quotes
```

💡 Tip: Triple quotes are great for long messages or text that spans multiple lines.

Boolean (bool)

Booleans represent only two values: True or False. Under the hood, True equals 1 and False equals 0, so they can be used in arithmetic expressions.

```
is_active = True
is_logged_in = False
print(int(is_active))    # True → 1
print(int(is_logged_in)) # False → 0
► Output: 1 | 0
```

1.2 Lists

A list is an ordered, mutable collection of items inside square brackets `[]`. Mutable means you can add, remove, or change items after the list is created. Python indexing always starts from 0, so the first element is at index 0.

- Use `append()` to add a new element at the end.
- Use `list[index] = value` to update an existing element.
- Lists can hold mixed data types — int, str, bool, even other lists.

```
fruits = ['apple', 'banana', 'mango']
fruits.append('orange')    # add element at end
fruits[0] = 'grapes'       # update first element
print(fruits)
# ['grapes', 'banana', 'mango', 'orange']

# Heterogeneous list — mixed types are fine
mixed = [10, 20.5, 'Hello', True, [1, 2, 3]]
► Output: ['grapes', 'banana', 'mango', 'orange']
```

💡 **Tip:** `fruits[-1]` accesses the last element. Negative indexing counts from the end.

1.3 Tuples

A tuple is like a list but immutable — once created, its values cannot be changed. Defined with round brackets `()`, tuples are used when data should remain constant, such as coordinates or fixed configuration values.

- Attempting to change a tuple value raises a `TypeError`.
- Tuples are faster than lists for read-only data.

```
fruits = ('apple', 'banana', 'cherry')
print(fruits[0])    # apple

# This will CRASH — tuples are immutable!
# fruits[0] = 'orange' → TypeError

# Mixed-type tuple
mixed_tuple = (10, 20.5, 'Hello', True, [1, 2, 3])
```

💡 **Tip:** If your data won't change, prefer a tuple over a list — it's safer and slightly faster.

1.4 Dictionary

A dictionary stores data as key-value pairs inside curly braces `{}`. Think of it like a real dictionary — you look up a word (key) to find its meaning (value). Keys must be unique and are usually strings; values can be any data type.

- Access a value with `dict['key']`.
- Add or update a key with `dict['key'] = new_value`.

```

user = {
    'name'      : 'John Doe',
    'age'       : 30,
    'is_active': True,
    'skills'    : ['Python', 'ML', 'Data Science'],
}
print(user['name'])    # John Doe
print(type(user))     # <class 'dict'>
► Output: John Doe | <class 'dict'>

```

1.5 Control Flow (if / elif / else)

Control flow lets your program make decisions. Use `if` to check a condition, `elif` for additional checks, and `else` as a final fallback. Python uses indentation (4 spaces) to define code blocks — there are no curly braces.

- `if` runs when the condition is `True`.
- `elif` is checked only when all previous conditions were `False`.
- `else` is the final fallback — runs when nothing else matched.

```

score = 85

if score > 90:
    grade = 'A'
elif score >= 80:    # checked only if score <= 90
    grade = 'B'
else:                # everything below 80
    grade = 'C'

print(grade)        # B
► Output: B

```

💡 **Tip:** Indentation is not optional in Python — incorrect indentation causes an `IndentationError`.

1.6 Loops

Loops let you execute a block of code repeatedly. Use a `for` loop when you know the number of iterations, and a `while` loop when you want to repeat until a condition becomes `False`.

for Loop

A `for` loop iterates over every item in a sequence (list, tuple, range, string) one by one, executing the block for each item.

```

numerical_list = [10, 20, 30, 40, 50]
for num in numerical_list:
    print(num)
► Output: 10 20 30 40 50 (each on a new line)

```

while Loop

A `while` loop keeps running as long as its condition is `True`. Always ensure the condition will eventually become `False` — otherwise you create an infinite loop that never stops.

```

x = 0
while x < 5:
    print(x)
    x += 1    # increments x; loop stops when x == 5
► Output: 0 1 2 3 4 (each on a new line)

```

💡 **Tip:** Always include an update statement (like `x += 1`) inside a `while` loop to avoid infinite loops.

1.7 Functions

A function is a reusable block of code that performs a specific task. Define it once with `def`, then call it as many times as needed. Functions can accept parameters (inputs) and return results using `return`.

- `def` keyword creates a new function.
- Parameters are placeholder names for values passed at call time.
- `return` sends a value back to the caller; without it the function returns `None`.

```
def greet(name):  
    print('Hello, ' + name + '!')  
  
greet('AIBees')    # Hello, AIBees!  
greet('Alice')     # Hello, Alice!  
  
def check_even_odd(number):  
    if number % 2 == 0:  
        return 'Even'  
    else:  
        return 'Odd'  
  
print(check_even_odd(10))    # Even  
print(check_even_odd(7))    # Odd  
► Output: Hello, AIBees! | Even | Odd
```

💡 **Tip:** Keep each function focused on one task. Functions that do too many things are harder to test and debug.

1.8 Exception Handling

Exceptions are runtime errors. Instead of letting the program crash, wrap risky code in a `try` block — if an error occurs, the `except` block catches it and handles it gracefully, keeping your program running.

- `try` — the code that might raise an error.
- `except` — what to do if that error occurs.
- Use `except Exception as e` to catch any error and read its message.

```
def even_or_odd(number):  
    try:  
        if number % 2 == 0:  
            return 'Even'  
        else:  
            return 'Odd'  
    except Exception as e:  
        return f'Error: {e}'  
  
# Division by zero example  
try:  
    result = 10 / 0  
except ZeroDivisionError:  
    print('Cannot divide by zero.')  
► Output: Cannot divide by zero.
```

💡 **Tip:** Always catch specific exceptions (`ZeroDivisionError`, `TypeError`) first before the generic `Exception`.

1.9 List Comprehension

List comprehension is a concise one-line way to build a new list by applying an expression to each item in an iterable. It replaces a 3–4 line `for` loop with a single readable statement. Syntax: `[expression for item in iterable]`.

```

numbers = [1, 2, 3, 4, 5]

# Traditional loop approach
for num in numbers:
    print(f'{num} squared is {num * num}')

# List comprehension — same result, one line
squares = [num * num for num in numbers]
print(squares)    # [1, 4, 9, 16, 25]
► Output: [1, 4, 9, 16, 25]

```

💡 **Tip:** Add a condition at the end: `[x for x in numbers if x % 2 == 0]` returns only even numbers.

1.10 Lambda Functions

A lambda is an anonymous (nameless) function written in a single line. Use lambdas for short, simple operations — especially when passing a function as an argument (e.g. to `sorted()` or `apply()`). Syntax: `lambda arguments : expression`.

```

# Regular function
def add(x, y):
    return x + y

# Equivalent lambda — same result, one line
add = lambda x, y: x + y
print(add(5, 3))    # 8
► Output: 8

```

💡 **Tip:** Lambdas are best for short, simple operations. For complex logic, use a regular `def` function instead.

1.11 Classes and Object-Oriented Programming (OOP)

A class is a blueprint for creating objects. Each object is an independent instance of that class with its own data (attributes) and behaviours (methods). OOP helps organise code into logical, reusable units that mirror real-world entities.

- **class** — the blueprint (e.g. `BankAccount`, `LLM`)
- `__init__` — the constructor, runs automatically when an object is created
- `self` — refers to the specific object being used; links attributes to the instance
- **Method** — a function defined inside a class that acts on the object's data

Simple LLM Class Example

```

class LLM:
    def __init__(self, model_name): # called on creation
        self.model_name = model_name

    def summarize(self, text):
        return f'Summarized by {self.model_name}: {text}'

gemini = LLM('Gemini Pro')    # create an object
print(gemini.summarize('Sample text.'))
► Output: Summarized by Gemini Pro: Sample text.

```

BankAccount Class — Practical Example

This example shows a real-world class with multiple methods. Each BankAccount object has its own independent balance — changing Alice's account never affects Bob's account.

```
class BankAccount:
    def __init__(self, owner_name, initial_balance=0):
        self.owner_name = owner_name
        self.balance = initial_balance


    def deposit(self, amount):
        if amount > 0:
            self.balance += amount

    def withdraw(self, amount):
        if amount > self.balance:
            print('Insufficient funds!')
        else:
            self.balance -= amount

    def get_balance(self):
        return f'Balance: {self.balance} for {self.owner_name}'

alice = BankAccount('Alice', 500)
alice.deposit(150)      # balance = 650
alice.withdraw(100)     # balance = 550
print(alice.get_balance())

► Output: Balance: 550 for Alice
```

 **Tip:** Each object (alice, bob) has its own independent copy of balance. They never interfere with each other.

2. NumPy

2.1 What is NumPy?

NumPy (Numerical Python) is the foundation of scientific computing in Python. It provides the ndarray — a fast, memory-efficient multi-dimensional array. Unlike regular Python lists, NumPy applies maths operations to all elements at once (vectorised), making it significantly faster for large datasets.

- NumPy arrays hold elements of the same data type — this makes them faster than lists.
- Operations like +, -, * apply element-by-element without any loops.
- NumPy is the backbone of Pandas, Scikit-learn, and TensorFlow.

```
import numpy as np
```

2.2 Creating Arrays

NumPy provides several shortcut functions to create common arrays instantly, so you don't have to build them manually from scratch. These are the most frequently used ones.

```
# From a Python list
arr1 = np.array([10, 20, 30])
arr2 = np.array([1, 2, 3])

# Filled arrays
np.zeros(10)      # [0. 0. 0. ... 0.] — 10 zeros
np.ones(10)       # [1. 1. 1. ... 1.] — 10 ones
np.ones(10) * 5   # [5. 5. 5. ... 5.] — 10 fives
```

```
# Integer ranges
np.arange(10, 51)      # integers 10 to 50
np.arange(10, 51, 2)   # even numbers 10-50 (step = 2)

# Reshape 1-D into a 2-D matrix
np.arange(9).reshape(3, 3)
# [[0 1 2]
#  [3 4 5]
#  [6 7 8]]
```

💡 **Tip:** `reshape(rows, cols)` changes shape without changing data. `rows × cols` must equal the total elements.

2.3 Element-Wise Operations

NumPy applies arithmetic operators (+, -, *, /) to matching elements of two arrays simultaneously — no loops needed. This is called vectorisation and is far faster than iterating with a Python for loop.

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

print(arr1 + arr2)  # [11 22 33] - adds matching pairs
print(arr1 * 2)     # [20 40 60] - scales every element
▶ Output: [11 22 33] | [20 40 60]
```

💡 **Tip:** Broadcasting: when shapes differ, NumPy automatically stretches the smaller array (e.g. `arr * 2` scales every element by 2).

2.4 Indexing and Slicing

NumPy uses `[row, col]` notation to select elements from 2-D arrays. Slicing with `:` lets you grab a range of rows or columns in one go — like a powerful Excel range selector, but in code.

```
mat = np.arange(1, 26).reshape(5, 5)
# [[ 1  2  3  4  5]
#  [ 6  7  8  9 10]
#  [11 12 13 14 15]
#  [16 17 18 19 20]
#  [21 22 23 24 25]]

mat[0]          # first row      → [ 1  2  3  4  5]
mat[4, :]       # last row       → [21 22 23 24 25]
mat[3, 4]       # single element → 20

mat[2:, 1:]     # bottom-right block (rows 2-4, cols 1-4)
# [[12 13 14 15]
#  [17 18 19 20]
#  [22 23 24 25]]

mat[:3, 1:2]    # rows 0-2, col 1 only → [[ 2] [ 7] [12]]
mat[3, 1:5]     # row 3, cols 1-4     → [17 18 19 20]
```

💡 **Tip:** Slice syntax `start:stop` — start is inclusive, stop is exclusive. Omit start for 'from beginning'; omit stop for 'to end'.

3. Pandas

3.1 What is Pandas?

3.2 Loading Data

`read_csv()` is the most common way to bring data into a `DataFrame`. It reads a comma-separated values file and automatically uses the first row as column headers.

```
sal = pd.read_csv('Salaries.csv')  
# Creates a DataFrame named 'sal' from the CSV file
```

💡 **Tip:** Always check the file path. If the CSV is not in the same folder as your notebook, provide the full path.

3.3 Exploring the DataFrame

Before analysing data, always explore it first to understand the shape, column names, data types, and whether any values are missing. These commands form your standard starting checklist for any new dataset.

```
sal.shape           # (rows, columns) e.g. (148654, 13)  
sal.head()          # first 5 rows — quick preview  
sal.head(2)         # first 2 rows  
sal.tail(2)         # last 2 rows — check end of data  
sal.columns         # list all column names  
sal.columns[0]      # name of the first column  
sal.info()          # dtypes, non-null counts per column  
sal.describe()      # count, mean, min, max for numeric cols  
sal.describe(include='all') # stats for ALL columns
```

💡 **Tip:** `sal.info()` is crucial — it reveals missing values when the non-null count is less than the total row count.

3.4 Selecting & Filtering Data

Use square brackets to select a column by name (returns a `Series`). To filter rows, pass a condition inside square brackets — Pandas returns only the rows where that condition is `True`. This works exactly like a `WHERE` clause in SQL.

Statistical Queries

```
sal['BasePay'].mean() # average BasePay across all rows  
sal['OvertimePay'].max() # highest OvertimePay value
```

Filtering Rows by Condition

```
# Job title of a specific employee  
sal[sal['EmployeeName'] == 'JOSEPH DRISCOLL']['JobTitle']  
  
# Their total pay including benefits  
sal[sal['EmployeeName'] == 'JOSEPH DRISCOLL']['TotalPayBenefits']  
  
# Full row of the highest-paid employee  
sal[sal['TotalPayBenefits'] == sal['TotalPayBenefits'].max()]
```

💡 **Tip:** The condition `sal['col'] == value` creates a boolean `Series` (`True/False`). Wrapping it in `sal[...]` returns only the `True` rows.

3.5 GroupBy

groupby() splits the DataFrame into groups by a column's unique values, applies an aggregation function to each group, and combines the results. It is Pandas' version of SQL's GROUP BY — ideal for computing per-category statistics.

```
# Average BasePay broken down by Year
sal.groupby('Year')['BasePay'].mean()

# Returns a Series: Year as index, mean BasePay as values
```

💡 **Tip:** Pattern: `df.groupby('group_col')['value_col'].func()` — use `.mean()`, `.sum()`, `.max()`, `.count()` etc.

3.6 Value Counts

value_counts() counts how many times each unique value appears in a column and returns the result sorted from most to least frequent. It's the fastest way to understand the distribution of any categorical column.

```
sal['JobTitle'].value_counts()      # all job titles
sal['JobTitle'].value_counts().head(5)  # top 5 only
```

💡 **Tip:** `value_counts()` sorts descending by default. Add `.reset_index()` if you need it as a regular DataFrame.

3.7 Apply & Lambda

apply() runs a custom function on every value in a Series. Combining it with a lambda avoids defining a separate function. This is powerful for transforming or categorising data based on any custom logic — especially when built-in methods aren't enough.

```
# Count employees whose job title contains 'chief'

def chief_string(title):
    return 'chief' in title.lower()  # case-insensitive check

# apply() + lambda runs chief_string on every row
count = sum(sal['JobTitle'].apply(lambda x: chief_string(x)))
print(count)  # number of employees with 'chief' in title
```

💡 **Tip:** `title.lower()` converts to lowercase first, so 'Chief', 'CHIEF', and 'chief' are all matched correctly.