# Python Note Summary

## 🐍 Python Notes Directory

1. input() Function
2. print() Function

---

## Lecture 6 – Collections

1. Lists
   - Creation
   - Accessing Elements
   - Methods (append, remove, etc.)
2. Tuples
3. Sets
4. Dictionaries

---

## Lecture 7 – Classes and Objects

1. Defining Classes
2. **init** Method
3. self Keyword
4. Methods
5. Creating Objects

---

## Lecture 8 – Modules and Packages

1. Importing Modules
2. Using Standard Library
3. if **name** == "**main**"

---

## Lecture 9 – File I/O

1. Reading Files
2. Writing Files
3. with Statement

---

## Lecture 10 – Advanced Topics

1. Lambda Expressions
2. List Comprehensions
3. Exception Handling (try-except)
4. Multiple Paradigms in Python (Procedural, OOP, Functional)

---

1.Introduction

## 1.1.Python

### Definition:

**Python is a high-level programming language used to write programs easily and clearly.**

#### *Feature*

- Easy to read and write.
- Can be used for many purposes (web, data, automation).

#### *Syntax*

```
print("Hello, World!")
```

#### *Example*

- Running `print("Hello, World!")` shows text on the screen.

#### *Explanation*

- Python uses simple English-like words to code.

#### *Remark*

- Created by **Guido van Rossum** in 1991.

---

## 1.2.Installation and Running

### Definition:

**Installing Python means putting it on your computer to run programs.**

#### *Feature*

- Download from **python.org**.

- Use **IDLE**, **Terminal**, or **VS Code** to run.

### *Syntax*

```
python myfile.py
```

### *Example*

- Typing `python hello.py` runs `hello.py`.

### *Explanation*

- You write code in a file and run it using `python`.

### *Remark*

- Check installation with `python --version`.

---

## 1.3. Interpreter vs Compiled Languages

### Definition

**Python is an interpreted language, meaning it runs line by line, not compiled into machine code first.**

### *Feature*

- No separate compile step needed.
- Easier for quick testing.

### *Syntax*

```python
print("This runs directly")
```

### *Example*

- Type code in **Python shell** and see results immediately.

### *Explanation*

- Compiled languages (like C++) need compilation before running. Python does not.

### *Remark*

- Interpreter reads and runs code at the same time.

---

2.Basic Syntax

## 2.1. Variables

### Definition

**Variables are names that store values in Python.**

### *Feature*

- No need to declare type.
- Created when assigned.

### *Syntax*

```
x = 5
```

### *Example*

- `name = "Alice"`

### *Explanation*

- Here, `name` stores the string `"Alice"`.

### *Remark*

- Variable names are case-sensitive.

---

## 2.2. Data Types

### Definition:

**Data types tell what kind of value a variable stores.**

### *Type Name*

| Type (Python) | Full English Name |
|---|---|
| `int` | integer |
| `float` | floating point number |
| `str` | string |
| `bool` | boolean |
| `list` | list |
| `tuple` | tuple |
| `dict` | dictionary |
| `set` | set |

---

## 2.2.1. Integer `int` **

### *Feature*

- Whole numbers.

### *Syntax*

```
x = 10
```

### *Example*

- `score = 95`

### *Explanation*

- Stores integers like 1, -5, 100.

### *Remark*

- No decimal point.

---

## 2.2.2. Floating point number `float`

### *Feature*

- Numbers with decimals.

### *Syntax*

```
x = 3.14
```

### Example

- `price = 19.99`

### Explanation

- For storing non-whole numbers.

### Remark

- Uses `.` for decimal.

---

## 2.2.3. String `str`

### Feature

- Text data.

### Syntax

```
x = "Hello"
```

### Example

- `message = 'Good morning'`

### Basic String Manipulation

- Concatenation:

```
s1 = "Hello"
s2 = "World"
s3 = s1 + " " + s2  # "Hello World"
```

- Repetition:

```
s = "ha"
s2 = s * 3  # "hahaha"
```

- Slicing:

```
s = "abcdef"
print(s[1:4])  # "bcd"
```

- Replace:

```
s = "I like Java"
s2 = s.replace("Java", "Python")  # "I like Python"
```

#### *Explanation*

- Always inside quotes.

#### *Remark*

- Can use single or double quotes.

---

### 2.2.4. Boolean `bool`

#### *Feature*

- Only **True** or **False**.

#### *Syntax*

```
x = True
```

#### *Example*

- `is_ready = False`

#### *Explanation*

- Useful for conditions.

#### *Remark*

- First letter is uppercase.

---

### 2.2.5. List `list`

#### *Feature*

- Stores many items in order.

#### *Syntax*

```
x = [1, 2, 3]
```

#### *Example*

- `fruits = ["apple", "banana", "cherry"]`

#### *Explanation*

- Items separated by commas in `[]`.

#### *Remark*

- Can change elements later.

---

### 2.2.6. Tuple `tuple`

#### *Feature*

- Like list, but cannot change items.

#### *Syntax*

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

#### *Example*

- `days = ("Mon", "Tue", "Wed")`

#### *Explanation*

- Uses `()` instead of `[]`.

#### *Remark*

- Immutable (cannot modify).

## 2.2.7. Dictionary `dict`

### *Feature*

- Stores key-value pairs.

### *Syntax*

```
x = {"name": "Bob", "age": 20}
```

### *Example*

- `student = {"id": 123, "grade": "A"}`

### *Explanation*

- Uses `{}` with keys and values.

### *Remark*

- Keys must be unique.

---

## 2.2.8. Set `set`

### *Feature*

- Unordered unique items.

### *Syntax*

```
x = {1, 2, 3}
```

### *Example*

- `letters = {"a", "b", "c"}`

### *Explanation*

- No repeated items allowed.

### *Remark*

- No index access.

---

## 2.3. Operators

### Definition

**Operators are symbols to do calculations or compare values.**

### *Feature*

- `+`, `-`, `*`, `/`, `%`, `==`, `!=`, `<`, `>`, etc.

### *Syntax*

```
x = 5 + 3
```

### *Example*

- `if age >= 18:`

### *Explanation*

- `+` adds, `>=` checks if greater or equal.

### *Remark*

- Different types of operators: arithmetic, comparison, logical.

---

## 2.4. Comments

### Definition:

**Comments are notes in code that Python ignores.**

### *Feature*

- Start with `#`.

### *Syntax*

```
# This is a comment
```

## *Example*

- `# Print greeting`

## *Explanation*

- Helps explain code to humans.

## *Remark*

- Multi-line comments use `''' ... '''` or `""" ... """`.

---

3.Control Structures

# 3.1. if-else Statements

## Definition

### Used to run code only when a condition is true.

## *Feature*

- Checks condition.
- Runs code if condition is true.
- Optionally runs other code if condition is false.

## *Syntax*

```
if condition:
    # code when true
else:
    # code when false
```

## *Example*

```
x = 5
if x > 3:
    print("x is greater than 3")
else:
    print("x is not greater than 3")
```

### *Explanation*

- If `x > 3` is true, it prints "x is greater than 3".
- Otherwise, it prints "x is not greater than 3".

### *Remark*

- Indentation (spaces) is required in Python.

---

## 3.2.for Loop

### Definition

**A `for` loop is a control structure used to repeat a block of code for each item in a sequence such as a list, tuple, string, or a generated range of numbers.**

### *Feature*

- Iterates over each item in a sequence in order.
- Automatically stops when there are no more items.
- Works with `range()` , lists, tuples, strings, sets, or dictionaries.

### *Syntax*

```
for variable in sequence:
    # code block
```

### *Example 1 – Using `range()`*

```
for i in range(3):
    print(i)
```

### *Explanation*

- `range(3)` generates numbers `0, 1, 2` .
- `i` takes each value.
- `print(i)` prints each value on a new line.

---

### *Example 2 – Iterating over a list*

```python
for i in [1, 2, 3]:
    print(i)
```

### Explanation

- `[1, 2, 3]` is a list.
- The loop goes through each element: `1`, `2`, `3`.
- `print(i)` outputs each number.

---

## Example 3 – Iterating over a tuple

```python
for fruit in ('apple', 'banana', 'cherry'):
    print(fruit)
```

### Explanation

- `('apple', 'banana', 'cherry')` is a tuple.
- The loop prints each fruit in order.

---

## Example 4 – Iterating over a string

```python
for letter in "Hello":
    print(letter)
```

### Explanation

- `"Hello"` is a string.
- The loop prints each character: `H`, `e`, `l`, `l`, `o`.

---

## Example 5 – Using `range(start, stop, step)`

```python
for i in range(1, 6, 2):
    print(i)
```

### Explanation

- `range(1, 6, 2)` means start at `1`, stop before `6`, count by `2`.
- Outputs: `1`, `3`, `5`.

### *Remark*

- The loop variable name ( `i` , `fruit` , `letter` ) can be any valid name.
- Works with any iterable object.
- Be careful with indentation: only indented code runs inside the loop.

## 3.3. while Loops

### Definition

#### Repeats code while a condition is true.

### *Feature*

- Checks condition each time.
- Stops when condition is false.

### *Syntax*

```
while condition:
    # code to repeat
```

### *Example*

```
x = 0
while x < 3:
    print(x)
    x += 1
```

### *Explanation*

- Prints 0, 1, 2.
- Stops when `x` reaches 3.

### *Remark*

- Be careful to avoid infinite loops.

4.Functions

## 4.1.Function Definitions

### Definition

**Code block that does something, can be reused by calling its name.**

#### *Feature*

- Defined with `def`.
- Can take inputs (parameters).
- Can return output.

#### *Syntax*

```python
def function_name(parameters):
    # code
    return value
```

#### *Example*

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

#### *Explanation*

- Defines a function `add` that returns the sum of `a` and `b`.

#### *Remark*

- Use functions to organize code and avoid repetition.

## 4.1.1.Calling Function Format

### Definition

**Calling a function means executing a function that has been defined previously by using its name followed by parentheses.**

#### *Feature*

- Executes the code inside the function.
- May pass arguments if required.
- May return a value if defined with `return`.

### *Syntax*

```
function_name(arguments)
```

### *Example*

```python
def greet(name):
    print("Hello, " + name)

greet("Alice")
```

### *Explanation*

- `def greet(name):` defines a function with parameter `name`.
- `greet("Alice")` calls the function with argument `"Alice"`, resulting in output `Hello, Alice`.

### *Remark*

- Function must be defined before it is called.
- Parentheses are necessary even if no arguments are passed (e.g., `my_function()`).

---

## 2. Parameters and Return

### Definition

**Parameters are inputs to functions. Return sends a value back.**

### *Feature*

- Parameters listed in parentheses.
- `return` sends result back to where function was called.

### *Syntax*

```python
def greet(name):
    return "Hello " + name
```

### *Example*

```python
message = greet("Alice")
print(message)  # Hello Alice
```

### *Explanation*

- `name` is the parameter.
- `"Hello " + name` is returned.

### *Remark*

- If no `return`, function returns `None`.

---

## 3. Type Hints

### Definition

#### Optional way to show what types inputs and outputs are.

### *Feature*

- Makes code clearer.
- Not checked by Python itself, but by tools.

### *Syntax*

```
def add(a: int, b: int) -> int:
    return a + b
```

### *Example*

```
def greet(name: str) -> str:
    return "Hello " + name
```

### *Explanation*

- `name: str` means `name` should be a string.
- `-> str` means function returns a string.

### *Remark*

- Type hints are suggestions, not rules.

---

5.Input and Output

## 5.1. `input()` Function

### Definition

**The `input()` function reads a line of text typed by the user.**

### *Feature*

- Waits for the user to type something and press Enter.
- Returns the input as a string.

### *Syntax*

```
variable = input("Prompt message")
```

### *Example*

```
name = input("Enter your name: ")
print("Hello, " + name)
```

### *Explanation*

- When run, the program shows "Enter your name:".
- It waits for user input.
- Whatever the user types is stored as text in `name`.
- Then it prints a greeting using the input.

### *Remark*

- `input()` always returns a string. If you want a number, convert it using `int()` or `float()`.

## 5.1.1. Converting input() to int or float

### Definition

**The input() function returns user input as a string. It can be converted to an integer or float using `int()` or `float()` functions.**

### *Feature*

- Converts string input to numeric types.
- Necessary for calculations with numeric inputs.

### *Syntax*

```
int_variable = int(input("Enter an integer: "))
float_variable = float(input("Enter a float: "))
```

### *Example*

```
age = int(input("Enter your age: "))
height = float(input("Enter your height in meters: "))
print("Age:", age)
print("Height:", height)
```

### *Explanation*

- `input("Enter your age: ")` takes user input as string.
- `int()` converts the string to an integer.
- `float()` converts the string to a float.
- Useful when expecting numeric input for calculations.

### *Remark*

- If user input cannot be converted (e.g., letters for int), it raises a `ValueError`.

---

## 5.2.print() Function

### Definition

#### The `print()` function shows output on the screen.

### *Feature*

- Displays text, numbers, or variables.
- Can print multiple items separated by commas.

### *Syntax*

```
print(item1, item2, ...)
```

### *Example*

```
print("The answer is", 42)
```

### *Explanation*

- Prints the items separated by spaces.
- Used to show results or messages to the user.

### *Remark*

- You can customize print with parameters like `sep` and `end`, but those are advanced topics.

---

6.Python Collections

## 6.1.Collections

### Definition

**Collections are data types in Python used to store multiple values together.**

### *Feature*

- Group and manage multiple items
- Different types for different uses

---

## 6.1.1.Lists

### Definition

**Lists are ordered collections of items, changeable and allow duplicates.**

### *Feature*

- Items are stored in order.
- Items can be added, removed, or changed.
- Can contain mixed data types.

### *Syntax*

```
my_list = [item1, item2, item3]
```

### *Example*

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])  # Output: apple
```

```python
fruits.append("orange")
fruits.remove("banana")
print(fruits)
```

### *Explanation*

- Access items by position using square brackets.
- `append()` adds an item at the end.
- `remove()` deletes the first matching item.

### *Remark*

- Lists are very flexible and often used for storing groups of data.

---

## 6.1.2.Tuples

### Definition

**Tuples are ordered collections like lists, but cannot be changed after creation.**

### *Feature*

- Items have fixed order.
- Items cannot be added or removed.

### *Syntax*

```python
my_tuple = (item1, item2, item3)
```

### *Example*

```python
coordinates = (10, 20)
print(coordinates[1])  # Output: 20
```

### *Explanation*

- Useful when you want to store data that should not change.

### *Remark*

- Tuples use parentheses, lists use square brackets.

---

### 6.1.3 Sets

#### Definition

**Sets are collections of unique items without order.**

#### *Feature*

- No duplicate items.
- Order of items is not fixed.
- Items can be added or removed.

#### *Syntax*

```
my_set = {item1, item2, item3}
```

#### *Example*

```
colors = {"red", "green", "blue"}
colors.add("yellow")
print(colors)
```

```
s = "abracadabra"
unique_chars = set(s)

print(unique_chars)
```

**Output**

```
{'d', 'c', 'b', 'a', 'r'}
```

#### *Additional Use Case with Set*

If you want to **check whether two strings have common characters**:

```
s1 = "hello"
s2 = "world"

common = set(s1) & set(s2)
print(common)  # {'o', 'l'}
```

#### *Explanation*

- Used to store unique items or to check if an item is present quickly.

### *Remark*

- Since order is not kept, you cannot access items by position.

---

## 6.1.4.Dictionaries

### Definition

**Dictionaries store pairs of keys and values.**

### *Feature*

- Each key maps to a value.
- Keys must be unique.
- Items are unordered.

### *Syntax*

```
my_dict = {key1: value1, key2: value2}
```

### *Example*

```
person = {"name": "Alice", "age": 25}
print(person["name"])  # Output: Alice
person["age"] = 26
```

Suppose you want to **count the frequency of each character** in a string.

```
s = "abracadabra"
freq = {}  # an empty dictionary

for char in s:
    if char in freq:
        freq[char] += 1
    else:
        freq[char] = 1

print(freq)
```

Output

```
{'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1}
```

### *Explanation*

- Use keys to access or change values.
- Useful to represent objects or related data.

### *Remark*

- Keys can be strings or numbers; values can be any data type.

---

7.Classes and Objects

## 7.1.Defining Classes

### Definition

**A class defines a blueprint for objects.**

### *Feature*

- Groups data and functions together.

### *Syntax*

```
class ClassName:
    # body
```

### *Example*

```
class Dog:
    pass
```

### *Explanation*

- `class Dog:` defines a class named Dog.

### *Remark*

- Class names usually start with uppercase letters.

---

## 7.2. `__init__` Method

### Definition

**Special method called when creating an object.**

### *Feature*

- Initializes object's data.

### *Syntax*

```
def __init__(self, parameters):
    # body
```

### *Example*

```
class Dog:
    def __init__(self, name):
        self.name = name
```

### *Explanation*

- Sets `self.name` when creating Dog.

### *Remark*

- `__init__` has two underscores before and after.

---

## 7.3. `self` Keyword

### Definition

**`self` is a reference to the current instance of a class and is used to access variables and methods within the class.**

### *Feature*

- Refers to the object itself.
- Required as the first parameter in instance methods.
- Allows distinction between instance variables and local variables.

### *Syntax*

```
class ClassName:
    def method_name(self, other_parameters):
        self.variable = value
```

### *Example*

```
class Person:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print("Hello, " + self.name)

p = Person("Bob")
p.greet()
```

### *Explanation*

- `self.name = name` assigns the parameter `name` to the instance variable `self.name`.
- `self` is used inside methods to refer to the current object.
- When `p.greet()` is called, it outputs `Hello, Bob`.

### *Remark*

- Although `self` is not a keyword in Python, it is a strong convention and required as the first parameter in instance methods.

---

## 7.4.Methods

### Definition

**Functions inside a class.**

### *Feature*

- Perform actions using object data.

### *Syntax*

```
def method_name(self):
    # body
```

### *Example*

```python
class Dog:
    def bark(self):
        print(self.name + " barks")
```

### *Explanation*

- `bark` uses `self.name` to print message.

### *Remark*

- Methods always belong to a class.

---

## 7.5. Creating Objects

### Definition

**Creating an object means making an instance of a class, allowing you to use its variables and methods.**

### *Feature*

- Uses the class as a blueprint to create an individual object.
- Each object has its own copy of instance variables.

### *Syntax*

```python
object_name = ClassName(arguments)
```

### *Example*

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print("Hello, my name is " + self.name + " and I am " + str(self.age) + " 
years old.")

# Creating an object of Person
p1 = Person("Alice", 20)

# Calling the greet method
p1.greet()
```

### *Explanation*

- `class Person:` defines a class named `Person`.
- `def __init__(self, name, age):` initializes `name` and `age` for the object.
- `self.name = name` sets the object's `name` variable.
- `self.age = age` sets the object's `age` variable.
- `def greet(self):` defines a method that prints a greeting using the object's data.
- `p1 = Person("Alice", 20)` creates an object `p1` with name `"Alice"` and age `20`.
- `p1.greet()` calls the method, outputting:
  `Hello, my name is Alice and I am 20 years old.`

### *Remark*

- You can create multiple objects with different data using the same class.
- Parentheses after class name are needed when creating an object to pass arguments to `__init__`.

---

8.Modules and Packages

## 8.1. Importing Modules

### Definition

#### Using code from other files or libraries.

### *Feature*

- Reuses functions and classes.

### *Syntax*

```
import module_name
```

### *Example*

```
import math
```

### *Explanation*

- Imports the math module.

### *Remark*

- Use `module_name.function()` to use functions.

---

## 8.2. Using Standard Library

### Definition

**Python comes with many built-in modules.**

### *Feature*

- No need to install separately.

### *Syntax*

```
import module_name
```

### *Example*

```
import random
print(random.randint(1, 10))
```

### *Explanation*

- Uses `random.randint` to print a random number.

### *Remark*

- Standard library is part of Python installation.

---

## 8.3. if name == "main"

### Definition

**`if __name__ == "__main__":` is a special Python construct that checks whether a Python file is being run directly or being imported as a module.**

### *Feature*

- Allows code to run only when the file is executed directly.
- Prevents certain code from running when the file is imported as a module in another script.
- Commonly used to include test code or the main execution flow.

### *Syntax*

```python
def function_name():
    # function code

if __name__ == "__main__":
    # code to execute when the file is run directly
```

### *Example*

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

if __name__ == "__main__":
    result = add(3, 4)
    print("Result:", result)
```

### *Explanation*

- `def add(a, b):` defines a function to add two numbers.
- `if __name__ == "__main__":` checks if the current file is being run directly.
- If true, it executes the code block: calls `add(3, 4)` and prints `Result: 7`.
- If this file is imported into another file using `import`, the code inside the `if __name__ == "__main__":` block will not execute automatically.

### *Remark*

- `__name__` is a built-in variable in Python.
- When a file is run directly, `__name__` is set to `"__main__"`.
- When a file is imported as a module, `__name__` is set to the module name instead.
- This is useful for writing files that can be both **reusable modules** and **standalone scripts**.

---

9.File I,O

## 9.1. Reading Files

### Definition

**Reading files means opening a file stored on disk and getting its content into the program as strings.**

### *Feature*

- Reads the entire file or line by line.

- Requires opening the file before reading.
- Must close the file after reading to free system resources.

## *Syntax*

```
file = open("filename.txt", "r")
content = file.read()
file.close()
```

## *Example 1 – Reading entire file*

```
file = open("example.txt", "r")
content = file.read()
print(content)
file.close()
```

## *Explanation*

- `open("example.txt", "r")` opens the file in **read mode ("r")**.
- `file` is a **file object** of type `_io.TextIOWrapper` (in CPython), which represents an open text file.
- `file.read()` reads the entire content as a string.
- `file.close()` closes the file.

---

## *Example 2 – Reading lines using readlines()*

```
file = open("example.txt", "r")
lines = file.readlines()
for line in lines:
    print(line.strip())
file.close()
```

## *Explanation*

- `readlines()` reads all lines and returns a list of strings.
- `line.strip()` removes the newline character at the end of each line when printing.

---

## *Example 3 – Reading line by line using readline()*

```
file = open("example.txt", "r")
line1 = file.readline()
line2 = file.readline()
print(line1.strip())
print(line2.strip())
file.close()
```

### *Explanation*

- `readline()` reads one line at a time.
- Repeated calls read subsequent lines.

---

## 9.2. Writing Files

### Definition

#### Writing files means creating a file or overwriting/appending to an existing file with new content.

### *Feature*

- Requires opening the file in write ( `"w"` ) or append ( `"a"` ) mode.
- `"w"` mode **overwrites** the file if it exists, or creates a new file if it doesn't.
- `"a"` mode **appends** to the end of the file.

### *Syntax*

```
file = open("filename.txt", "w")
file.write("Some text.")
file.close()
```

### *Example 1 – Overwriting a file*

```
file = open("output.txt", "w")
file.write("Python is easy.")
file.close()
```

### *Explanation*

- Opens `output.txt` in write mode.
- If `output.txt` exists, its original content is **deleted**, and replaced with `"Python is easy."`.
- If `output.txt` does not exist, it is created.

### *Example 2 – Appending to a file*

```python
file = open("output.txt", "a")
file.write("\nNew line added.")
file.close()
```

### *Explanation*

- Opens `output.txt` in append mode.
- Adds `New line added.` at the end without deleting existing content.

---

## 9.3. `with` Statement

### Definition

**The `with` statement simplifies file handling by automatically closing the file when the block ends, even if errors occur.**

### *Feature*

- No need to explicitly call `close()`.
- Safer and preferred method for file operations.

### *Syntax*

```python
with open("filename.txt", "r") as file:
    # operations with file
```

### *Example 1 – Reading file with with-statement*

```python
with open("example.txt", "r") as f:
    content = f.read()
    print(content)
```

### *Explanation*

- Opens `example.txt` in read mode.
- `f` is a file object used within the block.
- Automatically closes the file after the block ends.

---

### *Example 2 – Writing file with with-statement*

```python
with open("output.txt", "w") as f:
    f.write("Python is easy.")
```

### *Explanation*

- Opens `output.txt` in write mode.
- **If `output.txt` exists, its content is deleted and replaced with `"Python is easy."`.**
- If `output.txt` does not exist, it is created.

### *Remark*

- `file` or `f` created by `open()` is a **file object** of type `_io.TextIOWrapper`.
- Always close files after reading or writing to avoid memory leaks or locked files.
- Use `with` statement for cleaner and safer code.
- Reading and writing modes include:

| Mode | Meaning |
|---|---|
| `"r"` | Read (default). File must exist. |
| `"w"` | Write. Overwrites existing file or creates a new one. |
| `"a"` | Append. Adds to end of file or creates a new one. |
| `"r+"` | Read and write. File must exist. |
| `"w+"` | Write and read. Overwrites existing file or creates new one. |
| `"a+"` | Append and read. Creates file if it does not exist. |

---

10.Advanced Topics

# 10.1.Lambda Expressions

## Definition

#### Lambda expressions create small anonymous functions.

### *Feature*

- No need for `def`
- Used for short simple functions

### *Syntax*

```python
lambda arguments: expression
```

### Example

```
add = lambda x, y: x + y
print(add(3, 4)) # Output: 7
```

### Explanation

- `lambda x, y: x + y` returns x + y

### Remark

- Useful in functions like `map` and `filter`

---

## 10.2.List Comprehensions

### Definition

**List comprehension is a way to create lists quickly.**

### Feature

- Shorter than normal for loops
- Returns a new list

### Syntax

```
[expression for item in iterable]
```

### Example

```
squares = [x*x for x in range(5)]
print(squares) # Output: [0,1,4,9,16]
```

### Explanation

- Creates a list of x*x for each x in range(5)

### Remark

- Makes code cleaner and shorter

---

## 10.3.Exception Handling (try-except)

### Definition

**Exception handling manages errors in programs.**

### *Feature*

- Uses `try` to test code
- Uses `except` to handle errors

### *Syntax*

```
try:
    # code
except ErrorType:
    # handle error
```

### *Example*

```
try:
    num = int("abc")
except ValueError:
    print("Invalid number")
```

### *Explanation*

- If `int("abc")` fails, prints error message

### *Remark*

- Prevents program from crashing

---

## 10.4.Multiple Paradigms in Python

### Definition

**Python supports different programming styles.**

### *Feature*

- Procedural: simple step-by-step
- Object-Oriented: classes and objects
- Functional: functions like map, filter, lambda

### *Syntax*

```python
# Procedural
print("Hi")

# OOP
class Dog:
    def bark(self):
        print("Woof")
d = Dog()
d.bark()

# Functional
nums = [1,2,3]
squared = list(map(lambda x: x*x, nums))
```

### *Example*

```python
# Functional example
double = lambda x: x*2
print(double(5)) # Output: 10
```

### *Explanation*

- Can mix styles in one program

### *Remark*

- Flexibility is a strength of Python

---

## 10.5.Dynamic Typing

### Definition

**Dynamic typing is a feature of a programming language where the type of a variable is determined at runtime, not in advance when writing the code.**

### *Feature*

- Variables do not have fixed types.
- The type can change if assigned a different value later.
- Easier and faster to write code, but may cause type-related errors only at runtime.

### *Syntax*

```python
x = 10      # x is an integer
x = "hello" # x is now a string
```

### *Example*

```python
def print_type(value):
    print(type(value))

print_type(42)       # <class 'int'>
print_type("hello")  # <class 'str'>
```

### *Explanation*

- In Python, you can assign an integer to a variable, then later assign a string to the same variable without errors.
- The interpreter determines the type when the program is running.

### *Remark*

- Dynamic typing makes Python flexible, but it requires careful testing to avoid type errors at runtime.

---

## 10.6.Duck Typing

### Definition

**Duck typing is a concept where the type or class of an object is less important than the methods or properties it has. It is summarized as "If it looks like a duck and quacks like a duck, it is treated as a duck."**

### *Feature*

- Focuses on what an object can do, not what it is.
- Allows writing flexible and general code.
- Common in dynamically typed languages like Python.

### *Syntax*

```python
class Duck:
    def quack(self):
        print("Quack!")

class Person:
    def quack(self):
        print("I'm quacking like a duck!")
```

```python
def make_it_quack(duckish):
    duckish.quack()
```

## Example

```python
donald = Duck()
john = Person()

make_it_quack(donald)  # Output: Quack!
make_it_quack(john)    # Output: I'm quacking like a duck!
```

## Explanation

- The function `make_it_quack` does not care whether the object is of type `Duck`.
- As long as the object has a `quack` method, it will work, showing duck typing.

## Remark

- Duck typing enhances flexibility but may cause runtime errors if expected methods do not exist.