PYTHON REVISION

1. **Basic Syntax and Data Types**
2. **Control Flow (if, for, while)**
3. **Functions and Scope**
4. **Lists, Tuples, Dictionaries, and Sets**
5. **File Handling**
6. **Object-Oriented Programming**
7. **Modules and Packages**
8. **Error Handling (try, except)**
9. **Comprehensions**
10. **Iterators and Generators**
11. **Decorators**
12. **Context Managers**
13. **Regular Expressions**
14. **Libraries (e.g., NumPy, Pandas, etc.)**

BASIC SYNTAX AND DATA TYPES

1. Basic Syntax

- **Print statement**:
  In Python, you can print to the console using the `print()` function:

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

- **Variables**:
  Variables in Python do not require explicit declaration of types:

```
x = 5       # Integer
y = "Hello" # String
z = 3.14    # Float
```

- **Comments**:
  Use `#` for single-line comments, and triple quotes for multi-line comments:

```
# This is a single-line comment
"""
This is a
multi-line comment
"""
```

2. Data Types

Python supports several built-in data types. Here's a summary of the most common ones:

- **Integers** (`int`): Whole numbers, e.g., `5`, `-3`.

```
a = 10
print(type(a))  # <class 'int'>
```

- **Floating Point Numbers** (`float`): Numbers with decimals, e.g., `3.14`.

```
b = 5.5
print(type(b))  # <class 'float'>
```

- **Strings** (`str`): A sequence of characters enclosed within single or double quotes.

```
s = "Hello, World!"
print(type(s))  # <class 'str'>
```

- **Booleans** (`bool`): Represent `True` or `False` values.

```
is_python_fun = True
print(type(is_python_fun))  # <class 'bool'>
```

- **NoneType**: Represents the absence of a value.

```
x = None
print(type(x))  # <class 'NoneType'>
```

3. Type Conversion

You can convert between different data types using built-in functions:

- `int()`: Converts to integer
- `float()`: Converts to float
- `str()`: Converts to string
- `bool()`: Converts to boolean

Example:

```
x = "123"
y = int(x)  # Converts string "123" to integer 123
z = float(x)  # Converts string "123" to float 123.0
```

4. Arithmetic Operators

- `+` : Addition
- `-` : Subtraction
- `*` : Multiplication
- `/` : Division
- `%` : Modulus (remainder)

- `**` : Exponentiation
- `//` : Floor division (integer result)

Example:

```
a = 10
b = 3
print(a + b)  # 13
print(a ** b) # 10^3 = 1000
```

5. Input from the User

You can use `input()` to get input from the user:

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

By default, `input()` returns a string, so you may need to convert the input:

```
age = int(input("Enter your age: "))  # Converts string input to
integer
```

CONTROL FLOW

2. Control Flow in Python

Control flow in Python allows you to control the execution of your code based on conditions and loops. Here are the primary control flow structures: `if`, `for`, and `while`.

1. `if` Statements

The `if` statement is used to test a condition. If the condition is `True`, the block of code within the `if` statement is executed. You can also include `elif` (else-if) and `else` statements to test multiple conditions.

Syntax:

```
if condition:
    # code block executed if condition is True
elif another_condition:
    # code block executed if another_condition is True
else:
    # code block executed if all conditions are False
```

Example:

```
x = 10

if x > 10:
    print("x is greater than 10")
elif x == 10:
    print("x is equal to 10")
else:
    print("x is less than 10")
```

Comparison Operators:

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

Logical Operators:

- `and`: True if both operands are True
- `or`: True if at least one operand is True
- `not`: Reverses the logical state of its operand

Example:

```
age = 25
is_student = False

if age > 18 and not is_student:
    print("Eligible for full membership")
```

2. `for` Loops

`for` loops are used for iterating over a sequence (such as a list, tuple, dictionary, set, or string). You can loop through the items in the sequence and execute code for each one.

Syntax:

```
for item in iterable:
    # code block executed for each item
```

Example:

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

for fruit in fruits:
    print(fruit)
```

You can use the `range()` function to loop through a sequence of numbers.

Example:

```
for i in range(5):
    print(i)  # Output: 0, 1, 2, 3, 4
```

- `range(start, stop, step)` allows specifying a range with a start, stop, and step size.

Example:

```
for i in range(0, 10, 2):
    print(i)  # Output: 0, 2, 4, 6, 8
```

3. `while` Loops

`while` loops repeatedly execute a block of code as long as a condition is `True`. Be cautious with `while` loops to avoid infinite loops.

Syntax:

```
while condition:
    # code block executed as long as condition is True
```

Example:

```
count = 0

while count < 5:
    print(count)
    count += 1  # Increments count to avoid infinite loop
```

4. `break` and `continue`

- `break`: Terminates the loop entirely.
- `continue`: Skips the rest of the current loop iteration and moves to the next iteration.

Example with `break`:

```
for i in range(10):
    if i == 5:
        break  # Loop exits when i equals 5
    print(i)  # Output: 0, 1, 2, 3, 4
```

Example with `continue`:

```
for i in range(5):
    if i == 3:
```

```
        continue  # Skips the iteration when i equals 3
    print(i)  # Output: 0, 1, 2, 4
```

5. `else` with Loops

Both `for` and `while` loops can have an `else` clause, which is executed when the loop completes normally (i.e., not terminated by `break`).

Example:

```
for i in range(5):
    print(i)
else:
    print("Loop completed")

# Output:
# 0
# 1
# 2
# 3
# 4
# Loop completed
```

Example with `break`:

```
for i in range(5):
    if i == 3:
        break
    print(i)
else:
    print("Loop completed")  # This won't run because the loop was
broken

# Output:
# 0
# 1
# 2
```

---

3. Functions and Scope in Python

Functions allow you to write reusable, modular code. In Python, functions are defined using the `def` keyword.

---

1. Defining and Calling Functions

Syntax:

```
def function_name(parameters):
    """Docstring explaining the function (optional)"""
    # Function body
    return value  # (optional)
```

Example:

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

print(greet("Alice"))  # Output: Hello, Alice
```

---

2. Function Parameters and Arguments

Python functions can have different types of parameters:

Positional Arguments

These are the normal arguments passed in order.

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

print(add(5, 3))  # Output: 8
```

Default Arguments

You can provide default values for parameters.

```
def greet(name="Guest"):
    return "Hello, " + name

print(greet())       # Output: Hello, Guest
print(greet("Bob"))  # Output: Hello, Bob
```

Keyword Arguments

You can specify arguments by name.

```
def person_info(name, age):
    return f"{name} is {age} years old."
```

```
print(person_info(age=22, name="Alice"))  # Output: Alice is 22 years old.
```

Arbitrary Arguments (`*args` and `**kwargs`)

- `*args` allows a function to accept any number of positional arguments.
- `**kwargs` allows a function to accept any number of keyword arguments.

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

print(add_numbers(1, 2, 3, 4))  # Output: 10

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

print_details(name="Alice", age=22, city="New York")
# Output:
# name: Alice
# age: 22
# city: New York
```

---

3. Scope of Variables

Scope determines where a variable can be accessed.

Local Scope

Variables declared inside a function are only accessible within that function.

```
def my_function():
    x = 10  # Local variable
    print(x)

my_function()
# print(x)  # This would cause an error (x is not defined outside the function)
```

Global Scope

Variables declared outside any function can be accessed anywhere in the script.

```
x = 20  # Global variable

def show():
    print(x)  # Accessing the global variable
```

```
show()  # Output: 20
```

Modifying Global Variables Inside a Function

To modify a global variable inside a function, use the `global` keyword.

```
x = 5

def change_x():
    global x
    x = 10

change_x()
print(x)  # Output: 10
```

---

4. Lambda Functions (Anonymous Functions)

Lambda functions are small, one-line anonymous functions.

Syntax:

```
lambda arguments: expression
```

Example:

```
square = lambda x: x * x
print(square(5))  # Output: 25
```

Lambda functions can have multiple arguments:

```
add = lambda a, b: a + b
print(add(3, 7))  # Output: 10
```

---

5. Higher-Order Functions

Python allows functions to take other functions as arguments.

Example with `map()`:

```
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x ** 2, numbers))
print(squared)  # Output: [1, 4, 9, 16]
```

Example with `filter()`:

```
numbers = [1, 2, 3, 4, 5, 6]
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # Output: [2, 4, 6]
```

Summary

✔ Functions make code reusable and modular.
✔ Parameters can be positional, keyword, or default.
✔ `*args` and `**kwargs` allow flexible function inputs.
✔ Scope defines where a variable can be accessed.
✔ Lambda functions provide a compact way to write simple functions.
✔ Higher-order functions like `map()` and `filter()` use functions as arguments.

List, Triple, Dictnary and Sets

---

1. Lists

A list is an **ordered, mutable** (changeable) collection of items. Defined with square brackets `[]`.

Creating a List

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

Accessing Elements

```
print(fruits[0])     # Output: apple
print(fruits[-1])    # Output: cherry (last item)
```

Modifying a List

```
fruits[1] = "orange"
print(fruits)  # ['apple', 'orange', 'cherry']
```

Common List Methods

```
fruits.append("grape")      # Add item to end
fruits.insert(1, "kiwi")    # Insert at index 1
fruits.remove("apple")      # Remove by value
fruits.pop()                # Remove last item
fruits.sort()               # Sort the list
fruits.reverse()            # Reverse the list
```

List Slicing

```
nums = [0, 1, 2, 3, 4, 5]
print(nums[1:4])  # [1, 2, 3]
print(nums[:3])   # [0, 1, 2]
print(nums[::2])  # [0, 2, 4]
```

---

2. Tuples

Tuples are **ordered and immutable**. Defined with parentheses `()`.

Creating and Accessing Tuples

```
point = (10, 20)
print(point[0])   # Output: 10
```

Tuples are useful when you want **data that shouldn't change**, like coordinates.

Single-Item Tuple

```
t = (5,)  # Must include comma
```

---

3. Dictionaries

Dictionaries are **unordered (in Python <3.7), mutable**, and store **key-value pairs**.

Creating a Dictionary

```
person = {
    "name": "Alice",
    "age": 22,
    "city": "Delhi"
}
```

Accessing/Modifying Values

```
print(person["name"])      # Output: Alice
person["age"] = 23
```

Common Dictionary Methods

```
person["email"] = "alice@example.com"  # Add
person.pop("city")                     # Remove by key
print(person.keys())                   # Get all keys
print(person.values())                 # Get all values
print(person.items())                  # Get all key-value pairs
```

Looping Through a Dictionary

```
for key, value in person.items():
    print(key, "->", value)
```

---

4. Sets

Sets are **unordered, mutable**, and contain **only unique items**. Defined with curly braces `{}`.

Creating a Set

```
s = {1, 2, 3, 4, 4, 2}
print(s)  # Output: {1, 2, 3, 4}
```

Set Methods

```
s.add(5)           # Add item
s.remove(2)        # Remove item (error if not found)
s.discard(10)      # Safe remove (no error if not found)
s.clear()          # Remove all items
```

Set Operations

```
a = {1, 2, 3}
b = {3, 4, 5}

print(a | b)   # Union: {1, 2, 3, 4, 5}
print(a & b)   # Intersection: {3}
print(a - b)   # Difference: {1, 2}
print(a ^ b)   # Symmetric Difference: {1, 2, 4, 5}
```

Summary Table

Type	Ordered	Mutable	Allows Duplicates	Syntax
List	✅	✅	✅	`[]`
Tuple	✅	❌	✅	`()`
Dict	✅*	✅	❌ (keys)	`{key: val}`
Set	❌	✅	❌	`{}` or `set()`

*Dicts maintain insertion order from Python 3.7+

FILE HANDLING

---

1. Opening a File

Use the built-in `open()` function:

```
file = open("filename.txt", "mode")
```

Modes:

Mode	Description
`'r'`	Read (default)
`'w'`	Write (overwrites file)
`'a'`	Append
`'x'`	Create (error if exists)
`'b'`	Binary mode
`'t'`	Text mode (default)

2. Reading from a File

Basic Read

```
f = open("sample.txt", "r")
content = f.read()
print(content)
f.close()
```

Read Line by Line

```
f = open("sample.txt", "r")
for line in f:
    print(line.strip())  # .strip() removes newline characters
f.close()
```

Other read methods

```
f.read()      # Reads the entire file
f.readline()  # Reads one line
f.readlines() # Returns a list of all lines
```

---

3. Writing to a File

Using 'w' Mode (overwrites)

```
f = open("output.txt", "w")
f.write("Hello, file!\n")
```

```
f.write("Second line.")
f.close()
```

Using 'a' Mode (append)

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

---

4. Using `with` Statement (Best Practice)

It automatically closes the file.

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

with open("output.txt", "w") as f:
    f.write("Written with 'with'!")
```

---

5. File Check & Deletion

Check if file exists

```
import os

if os.path.exists("sample.txt"):
    print("File found!")
else:
    print("File not found!")
```

Delete a file

```
os.remove("sample.txt")
```

---

6. Read/Write Binary Files

```
# Write binary
with open("data.bin", "wb") as f:
    f.write(b"Binary data")

# Read binary
with open("data.bin", "rb") as f:
```

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

Summary

Action	Method
Open a file	`open("file", "r")`
Read contents	`read()`, `readline()`, `readlines()`
Write/Append	`write()`
Close file	`close()`
Auto-close	`with open(...) as`
Delete file	`os.remove()`