STRING FORMATTING IN PYTHON – NOTES

String formatting is a way to **insert variables or values** into strings.

1. Using format() method (Python 3 and above)

```
name = "Alice"
age = 25
print("My name is {} and I am {} years old.".format(name, age))
```

With index numbers:

```
print("My name is {0} and I am {1} years old. {0} is learning
Python.".format(name, age))
```

With named placeholders:

```
print("My name is {n} and I am {a} years old.".format(n="Alice", a=25))
```

2. Using f-strings (Python 3.6+)

```
name = "Bob"
age = 30
print(f"My name is {name} and I am {age} years old.")
```

You can use expressions inside f-strings:

```
print(f"Next year I will be {age + 1} years old.")
```

3. Using % operator (Old method, still supported)

```
name = "Charlie"
age = 28
print("My name is %s and I am %d years old." % (name, age))
```

WHAT IS A SET IN PYTHON?

A **set** is an unordered, unindexed collection of **unique elements**.
It is **mutable** (can be changed) but cannot contain **mutable items** (like lists or dictionaries).
Defined using curly braces `{}` or the `set()` function.

```
my_set = {1, 2, 3, 4}
another_set = set([3, 4, 5])
```

KEY FEATURES OF SETS:

Feature	Description
Unordered	No indexing, elements have no order
No Duplicates	All elements are unique
Mutable	You can add or remove items

CREATING A SET:

```
# Empty set (use set(), not {})
empty_set = set()
# Set with values
```

```
fruits = {'apple', 'banana', 'mango'}
```

COMMONLY USED SET METHODS:

Method	Description	Example
`add()`	Adds an element to the set	`my_set.add(10)`
`remove()`	Removes a specific element (error if not found)	`my_set.remove(2)`
`discard()`	Removes element (no error if not found)	`my_set.discard(5)`
`pop()`	Removes and returns a random item	`item = my_set.pop()`
`clear()`	Empties the entire set	`my_set.clear()`
`copy()`	Returns a shallow copy	`new_set = my_set.copy()`
`update()`	Adds multiple elements from another set/list	`my_set.update([7, 8])`

SET OPERATIONS:

Operation	Symbol / Method	Description
Union	`	`or`set1.union(set2)`
Intersection	`& or set1.intersection(set2)`	Common elements in both sets
Difference	`- or set1.difference(set2)`	Elements in `set1` but not in `set2`
Symmetric Difference	`^ or set1.symmetric_difference(set2)`	Elements in either set, but not both

EXAMPLES:

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

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

SET MEMBERSHIP

```
if 2 in my_set:
    print("2 is present")

if 9 not in my_set:
    print("9 is not present")
```

WHAT IS A DICTIONARY IN PYTHON?

A **dictionary** is an **unordered collection** of **key-value pairs**.
Each key is **unique** and maps to a value.
Defined using curly braces `{}` with keys and values separated by a colon `:`

```
student = {'name': 'John', 'age': 21, 'course': 'Python'}
```

KEY FEATURES OF DICTIONARIES:

Feature	Description
Key-Value	Each element is a key-value pair
Unordered	Order is preserved from Python 3.7+
Mutable	You can change, add, or remove items
No Duplicates	Keys must be unique

CREATING A DICTIONARY:

```
# Basic dictionary
person = {'name': 'Alice', 'age': 25}

# Empty dictionary
empty_dict = {}

# Using dict() function
data = dict(name='Bob', age=30)
```

ACCESSING AND MODIFYING ITEMS:

```
# Access value by key
print(person['name'])          # Output: Alice

# Add or update a value
person['age'] = 26             # Updates age
person['city'] = 'London'      # Adds new key-value pair

# Using get() (returns None if key doesn't exist)
print(person.get('gender'))    # Output: None
```

REMOVING ITEMS:

Method	Description	Example
`pop(key)`	Removes item by key	`person.pop('age')`
`popitem()`	Removes the last inserted item	`person.popitem()`
`del`	Deletes key or entire dictionary	`del person['name']`
`clear()`	Removes all items	`person.clear()`

LOOPING THROUGH A DICTIONARY

```
# Loop through keys
for key in person:
    print(key)

# Loop through values
```

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

# Loop through key-value pairs
for key, value in person.items():
    print(key, value)
```

USEFUL DICTIONARY METHODS:

Method	Description
`keys()`	Returns all keys
`values()`	Returns all values
`items()`	Returns all key-value pairs as tuples
`get(key)`	Returns value for key, or None if not found
`update(dict2)`	Updates dictionary with another dictionary
`copy()`	Returns a shallow copy
`fromkeys(keys, val)`	Creates new dict with keys and a default value
`setdefault()`	Returns value of key, sets default if not found

EXAMPLE: FULL DICTIONARY WORKFLOW:

```
student = {
    'name': 'Rahul',
    'roll': 101,
    'subject': 'Math'
}

# Add new key
student['marks'] = 95

# Update existing value
student['subject'] = 'Science'

# Access value
print(student.get('name'))

# Loop through keys and values
for k, v in student.items():
    print(f"{k}: {v}")
```

HOW TO RAISE A CUSTOM ERROR

You can use Python's built-in exceptions with a custom message:

```
age = -5
if age < 0:
    raise ValueError("Age cannot be negative")
```

SHORT-HAND `IF-ELSE` IN PYTHON

Python allows a concise way to write `if-else` in one line, called a **ternary conditional operator**.

```
age = 20
status = "Adult" if age >= 18 else "Minor"
print(status)

# Output: Adult
```

`IF __NAME__ == "__MAIN__"` IN PYTHON — EXPLAINED SIMPLY

WHAT DOES IT MEAN?

```
if __name__ == "__main__":
    # run this code
```

This line checks whether the script is being **run directly** or **imported as a module**.

WHY IS IT USED?

To **prevent certain code from running when the file is imported** elsewhere.

HOW IT WORKS:

When you **run a Python file directly**, Python sets the special variable `__name__` to `"__main__"`.

When you **import the file into another Python script**, `__name__` becomes the **name of the module** (i.e., the file name, not `"__main__"`).

Example:

```
# file: myscript.py

def greet():
    print("Hello!")

if __name__ == "__main__":
    greet()  # This will only run if you run myscript.py directly
```

```
python myscript.py
```

Output: **`Hello!`**

But if you do:

```
import myscript
```

Nothing will print automatically.

WHY USE IT?

- Helps in writing **reusable** and **testable** code.

- Keeps test/demo code separate from functions/classes.

ENUMERATE FUNCTION IN PYTHON

WHAT IS **ENUMERATE()**

The `enumerate()` function adds a **counter** (index) to an **iterable** (like a list, tuple, or string) and returns it as an `enumerate` object.

SYNTAX:

```
enumerate(iterable, start=0)
```

- `iterable`: Any iterable object (like list, tuple, string, etc.)
- `start`: (Optional) The index to start counting from. Default is 0.

EXAMPLE 1: WITH LIST:

```
fruits = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(fruits):
    print(index, fruit)
```

Output:

```
0 apple
1 banana
2 cherry
```

EXAMPLE 2: STARTING FROM 1:

```
for index, fruit in enumerate(fruits, start=1):
    print(index, fruit)
```

Output:

```
1 apple
2 banana
3 cherry
```

HOW IMPORTING IN PYTHON WORKS

Importing in Python is the process of loading code from a Python module into the current script. This allows you to use the functions and variables defined in the module in your current script, as well as any additional modules that the imported module may depend on.

Once a module is imported, you can use any of the functions and variables defined in the module by using the dot notation. For example, to use the sqrt function from the math module, you would write:

```
import math
result = math.sqrt(9)
print(result)  # Output: 3.0
```

from keyword:

You can also import specific functions or variables from a module using the from keyword. For example, to import only the sqrt function from the math module, you would write:

```
from math import sqrt

result = sqrt(9)
print(result)  # Output: 3.0
```

You can also import multiple functions or variables at once by separating them with a comma:

```
from math import sqrt, pi

result = sqrt(9)
print(result)  # Output: 3.0

print(pi)  # Output: 3.141592653589793
```

importing everything:

It's also possible to import all functions and variables from a module using the * wildcard. However, this is generally not recommended as it can lead to confusion and make it harder to understand where specific functions and variables are coming from.

```
from math import *

result = sqrt(9)
print(result)  # Output: 3.0

print(pi)  # Output: 3.141592653589793
```

Python also allows you to rename imported modules using the as keyword. This can be useful if you want to use a shorter or more descriptive name for a module, or if you want to avoid naming conflicts with other modules or variables in your code.

The "as" keyword:

```
import math as m

result = m.sqrt(9)
print(result)  # Output: 3.0

print(m.pi)  # Output: 3.141592653589793
```

The dir function:

Finally, Python has a built-in function called dir that you can use to view the names of all the functions and variables defined in a module. This can be helpful for exploring and understanding the contents of a new module.

```
import math

print(dir(math))
```

This will output a list of all the names defined in the math module, including functions like sqrt and pi, as well as other variables and constants.