Python Built-in Data Types:

Python provides several built-in data types, which can be classified into different categories. Some of the most commonly used ones include:

1. Number Data Type

Python provides several built-in numeric data types that are used to represent different types of numbers. The three main numeric data types in Python are:

a) Integer (int)

b) Floating-point (float)

c) Complex (complex)

Each of these types serves a different purpose, and Python provides various operations and functions to work with them.

```
# Integer (int)

x = 10

print(type(x))  # Output: <class 'int'>


# Float (float)

y = 10.5

print(type(y))  # Output: <class 'float'>


# Complex (complex)

z = 3 + 4j

print(type(z))  # Output: <class 'complex'>
```

a) Integer (int)

Definition:

An integer is a whole number that can be positive, negative, or zero. It does not have a fractional (decimal) part.

Example:

```
x = 10      # Positive integer
y = -5      # Negative integer
z = 0       # Zero

print(type(x))  # Output: <class 'int'>
print(type(y))  # Output: <class 'int'>
print(type(z))  # Output: <class 'int'>
```

Characteristics:

Can be of arbitrary length (limited by system memory).

Supports basic arithmetic operations: +, -, *, /, //, %, **.

Can be converted to other types (float, complex, str, etc.).

Operations with Integers:

```
a = 10
b = 3

print(a + b)   # Addition: 13
print(a - b)   # Subtraction: 7
print(a * b)   # Multiplication: 30
print(a / b)   # Division (float result): 3.3333
print(a // b)  # Floor Division (integer result): 3

print(a % b)   # Modulus (remainder): 1

print(a ** b)  # Exponentiation (power): 1000
```

b) Floating-Point (float)

Definition:

A floating-point number (or float) is a number that has a decimal or fractional part. It is used to represent real numbers.

Example:

```
x = 10.5   # Positive float

y = -3.14  # Negative float

z = 0.0    # Zero as a float


print(type(x))  # Output: <class 'float'>

print(type(y))  # Output: <class 'float'>

print(type(z))  # Output: <class 'float'>
```

Characteristics:

1. Supports decimal numbers.
2. Uses double-precision floating point representation (IEEE 754 standard).
3. Can be represented in scientific notation (e or E).
4. Operations are similar to integers.

Scientific Notation Example:

```
a = 3.5e3  # 3.5 * 10^3 = 3500.0

b = 1.2E-2 # 1.2 * 10^-2 = 0.012
```

```
print(a)  # Output: 3500.0
print(b)  # Output: 0.012
```

Operations with floats:

```
a = 5.5
b = 2.0

print(a + b)   # 7.5
print(a - b)   # 3.5
print(a * b)   # 11.0
print(a / b)   # 2.75
print(a // b)  # 2.0 (Floor Division)
print(a % b)   # 1.5 (Modulus)
print(a ** b)  # 30.25 (Exponentiation)
```

Rounding and Precision Handling:

```
x = 3.14159
print(round(x,2))  # Output: 3.14
```

c) Complex Numbers (complex)

Definition:

A complex number consists of a real part and an imaginary part (denoted by j in Python). It is written as a + bj, where:

a is the real part (float or int).

b is the imaginary part (float or int).

j is the imaginary unit (√-1).

Example:

```
c1 = 2 + 3j

c2 = -1.5 + 4.5j


print(type(c1))  # Output: <class 'complex'>
```

Characteristics:

1. Used in scientific computing and engineering.
2. Python provides built-in functions for handling complex numbers.

Operations with Complex Numbers:

```
c1 = 2 + 3j

c2 = 1 - 4j


print(c1 + c2)  # (3 - 1j)

print(c1 - c2)  # (1 + 7j)

print(c1 * c2)  # (14 - 5j)

print(c1 / c2)  # (-0.6470588235294118 + 0.7058823529411765j)
```

Accessing Real and Imaginary Parts:

```
c = 4 + 5j

print(c.real)  # Output: 4.0

print(c.imag)  # Output: 5.0
```

Converting Complex to Other Types:

```
c = 3 + 4j

print(abs(c))  # Output: 5.0 (Magnitude of the complex number)
```

Type Conversions Between Numeric Types :

Python allows conversion between numeric types using:

int()

float()

complex()

Examples:

```
# Convert float to int
x = int(10.99)
print(x)  # Output: 10

# Convert int to float
y = float(7)
print(y)  # Output: 7.0

# Convert int to complex
z = complex(5)
print(z)  # Output: (5+0j)

# Convert float to complex
w = complex(3.5)
print(w)  # Output: (3.5+0j)
```

Mathematical Functions in Python

Python provides a math module for advanced mathematical operations.

```
import math

print(math.sqrt(16))   # Square root: 4.0
print(math.pow(2, 3))  # Power: 8.0
print(math.log(10))    # Natural log: 2.3025
print(math.sin(math.pi/2))  # Sine: 1.0
print(math.factorial(5))  # Factorial: 120
```

Boolean

In Python, bool is a subclass of int, where:

True is equivalent to 1

False is equivalent to 0

Examples:

```
print(True + 10)  # Output: 11
print(False + 10)  # Output: 10
print(isinstance(True, int))  # Output: True
```

2. Sequence Data Types

Python provides several built-in sequence data types that store collections of items. A sequence is an ordered collection of elements, where each element can be accessed using an index.

Types of Sequence Data Types:

List (list) - Mutable sequence of elements.

Tuple (tuple) - Immutable sequence of elements.

Range (range) - Immutable sequence of numbers.

String (str) - Immutable sequence of characters.

Let’s explore each of these in detail with examples.

1. List (list)

A list is a mutable (modifiable) ordered collection of elements. It can contain elements of different data types (integers, floats, strings, other lists, etc.).

Characteristics of Lists:

Ordered (elements maintain the order of insertion).

Mutable (can be changed after creation).

Supports duplicate elements.

Allows different data types in the same list.

Creating a List:

```
# Creating a list

my_list = [10, 20, 30, "Python", 3.14]

print(my_list)  # Output: [10, 20, 30, 'Python', 3.14]

print(type(my_list))  # Output: <class 'list'>
```

Accessing Elements in a List:

```
print(my_list[0])  # Output: 10 (First element)

print(my_list[-1])  # Output: 3.14 (Last element)
```

Modifying a List:

```
my_list[1] = 99  # Changing an element
print(my_list)  # Output: [10, 99, 30, 'Python', 3.14]
```

List Operations:

```
# Adding elements
#my_list.append(50)  # Adds to the end
#my_list.insert(0,1000)  # Adds to the end
print(my_list)  # Output: [10, 99, 30, 'Python', 3.14, 50]

# Removing elements
#my_list.remove(50)  # Removes 30 from list
print(my_list)  # Output: [10, 99, 'Python', 3.14, 50]

#indexing
print(my_list[1]) # Output: 99



# Slicing
print(my_list[1:4])  # Output: [99, 'Python', 3.14]

# Repetition
print(my_list * 2)  # Output: [10, 99, 'Python', 3.14, 50, 10, 99, 'Python', 3.14, 50]
```

# Membership :Membership tests whether an element exists in a sequence using the in or not in operators.

```
print(20 in my_list)   # Output: True (20 is in the list)
print(10 not in my_list)  # Output: True (40 is not in the list)
```

# del : The del statement is used to delete items from a sequence (like lists).

# It can be used to remove an element at a specific index or to remove a slice of the sequence.

```
del my_list[1]   # Deletes element at index 1 (20)
print(my_list)    # Output: [10, 30, 40]

del my_list[1:3]  # Deletes elements from index 1 to 2 (30, 40)
print(my_list)    # Output: [10]

# Checking length
print(len(my_list))  # Output: 5

#concate
list2=[59,67,"Hi"]
print(my_list + list2)
```

2. Tuple (tuple)

A tuple is an immutable (unchangeable) ordered collection of elements. Like lists, tuples can contain elements of different data types.

Characteristics of Tuples:

Ordered (elements maintain insertion order).

Immutable (cannot be changed after creation).

Supports duplicate elements.

More memory-efficient than lists.

Creating a Tuple:

```
# Creating a tuple
my_tuple = (10, "Python", 3.14)
print(my_tuple)  # Output: (10, 'Python', 3.14)
print(type(my_tuple))  # Output: <class 'tuple'>
```

Accessing Elements in a Tuple:

```
print(my_tuple[0])  # Output: 10

print(my_tuple[-1])  # Output: 3.14
```

Tuple Immutability:

```
my_tuple[1] = 99  # This will raise an error: TypeError: 'tuple' object does not support item assignment
```

Tuple Operations:

```
# Tuple concatenation

new_tuple = my_tuple + (100, 200)

print(new_tuple)  # Output: (10, 'Python', 3.14, 100, 200)


#indexing

print(new_tuple[1]) # Output: Python



# Slicing

print(my_tuple[1:4])  # Output: (20, 30, 'Python')


# Repetition

print(my_tuple * 3)  # Output: (10, 'Python', 3.14, 10, 'Python', 3.14, 10, 'Python', 3.14)


# Membership

print(10 in my_tuple)  # Output: True (2 is in the tuple)

print(5 not in my_tuple)  # Output: True (5 is not in the tuple)


# Length of tuple

print(len(my_tuple))  # Output: 3


# del

#del my_tuple[1] #error

del my_tuple   #Deletes the entire tuple

print(len(my_tuple))
```

Tuple with a Single Element:

```
single_element_tuple = (10,)  # Note the comma! Otherwise, it is treated as an integer.
print(type(single_element_tuple))  # Output: <class 'tuple'>
```

3. Range (range)

The range type represents an immutable sequence of numbers, commonly used for loops.

Characteristics of Ranges:

Immutable (cannot be changed after creation).

Efficient memory usage as numbers are generated on demand.

Supports start, stop, and step.

Creating a Range:

```
# Range from 0 to 4
r = range(5)
print(list(r))  # Output: [0, 1, 2, 3, 4]

# Range with a start and stop
r2 = range(2, 10)
print(list(r2))  # Output: [2, 3, 4, 5, 6, 7, 8, 9]

# Range with step
r3 = range(1, 10, 2)  # Step of 2
print(list(r3))  # Output: [1, 3, 5, 7, 9]
```

4. String (str)

A string is an immutable sequence of characters. Strings are enclosed in single ('), double (") or triple (''' or """) quotes.

Characteristics of Strings:

Immutable (cannot be changed after creation).

Supports indexing and slicing.

Supports string operations like concatenation, repetition, and formatting.

Creating a String:

```
my_string = "Hello, Python!"
print(my_string)  # Output: Hello, Python!
print(type(my_string))  # Output: <class 'str'>
```

Accessing Characters in a String:

```
print(my_string[0])  # Output: 'H'
print(my_string[-1])  # Output: '!'
```

String Operations:

```
# Concatenation
str1 = "Hello"
str2 = "World"
print(str1 + " " + str2)  # Output: 'Hello World'

# Repetition
print(str1 * 3)  # Output: 'HelloHelloHello'

# Length of string
print(len(str1))  # Output: 14

#String Slicing:
print(my_string[0:5])  # Output: 'Hello'
print(my_string[:5])   # Output: 'Hello'
print(my_string[7:])   # Output: 'Python!'

#Membership
print('e' in str1)   # Output: True ('e' is in the string)
print('o' not in str1)  # Output: True ('z' is not in the string)


# String methods
print(str1.lower())  # Output: 'hello'
print(str1.upper())  # Output: 'HELLO'
str3=str1 + " " + str2
print(str3)
print(str3.replace("World","Python" ))  # Output: 'Hello, World!'
```

Comparison of Sequence Data Types:

Data Type	Mutability	Ordered	Allows Duplicate Elements	Example
List (list)	Mutable	Yes	Yes	[10, 20, "Python"]
Tuple (tuple)	Immutable	Yes	Yes	(10, 20, "Python")
Range (range)	Immutable	Yes	No	range(5)
String (str)	Immutable	Yes	Yes	"Hello, Python!"

3. Python Built-in Mapping Data Type:

Dictionary (dict)

A dictionary (dict) is a built-in mapping data type in Python that stores key-value pairs. It is one of the most powerful and commonly used data structures in Python. Unlike sequences (lists, tuples, strings), which store values indexed by position, dictionaries store values indexed by keys.

Characteristics of a Dictionary:

Unordered (before Python 3.7) → Since Python 3.7, dictionaries maintain insertion order.

Mutable → Can be modified (add, remove, update key-value pairs).

Keys must be unique → Duplicate keys are not allowed.

Keys must be immutable → Can be int, float, str, tuple (but not list or another dict).

Values can be of any data type → Lists, Tuples, Sets, or even another Dictionary.

Creating a Dictionary:

Dictionaries can be created using:

i) Curly braces {}

ii) The dict() constructor

Example 1: Creating a Dictionary

```
# Using curly braces
my_dict = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}
print(my_dict)
# Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}

# Using dict() constructor
my_dict2 = dict(name="Bob", age=30, city="London")
print(my_dict2)
# Output: {'name': 'Bob', 'age': 30, 'city': 'London'}
```

Example 2: Dictionary with Different Data Types

```
person = {
    "name": "John",
    "age": 28,
    "hobbies": ["reading", "gaming", "coding"],  # List inside dictionary
    "address": {"city": "Los Angeles", "zip": 90001},  # Nested dictionary
    100: "Numeric Key",
    (1, 2): "Tuple as Key"  # Tuples are immutable, so they can be keys
}
print(person)
```

3. Accessing Dictionary Elements

You can access values using keys (not indices like lists).

Using Square Brackets []

```
print(person["name"])  # Output: John
print(person["hobbies"])  # Output: ['reading', 'gaming', 'coding']
print(person.get("salary")) Output: KeyError
```

Using get() Method (Avoids KeyError)

```
print(person.get("age"))  # Output: 28
print(person.get("salary"))  # Output: None (instead of error)
print(person.get("salary", "Not Found"))  # Output: Not Found (default value)
```

Modifying a Dictionary:

Dictionaries are mutable, so you can update, add, or remove elements.

Updating an Existing Key

```
person["age"] = 29
print(person["age"])  # Output: 29
```

Adding a New Key-Value Pair:

```
person["gender"] = "Male"
```

```
print(person)

# Output: {'name': 'John', 'age': 29, 'hobbies': [...], 'address': {...}, 'gender': 'Male'}
```

Removing Elements:

```
del person["age"]  # Removes the key-value pair

print(person)


removed_value = person.pop("gender")  # Removes and returns the value

print(removed_value)  # Output: Male


person.clear()  # Removes all items from the dictionary

print(person)  # Output: {}
```

Dictionary Operations :

Checking if a Key Exists:

```
print("name" in person)  # Output: True

print("salary" in person)  # Output: False
```

Dictionary Length (len())

```
print(len(my_dict))  # Output: 3 (number of key-value pairs)
```

Iterating Through a Dictionary:

Using a for loop

```
for key in my_dict:
    print(key, ":", my_dict[key])
```

Using items() Method

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

Using keys() and values() Methods

```
print(my_dict.keys())  # Output: dict_keys(['name', 'age', 'city'])

print(my_dict.values())  # Output: dict_values(['Alice', 25, 'New York'])
```

Dictionary Methods :

get(key, default)	Returns the value for a key; returns default if key is missing	d.get("age", 30)
keys()	Returns all dictionary keys	d.keys()
values()	Returns all dictionary values	d.values()
items()	Returns all key-value pairs as tuples	d.items()
update(dict2)	Merges dict2 into dict	d.update({"city": "Paris"})
pop(key)	Removes and returns value of the given key	d.pop("age")
del d[key]	Deletes key-value pair	del d["name"]
clear()	Removes all elements	d.clear()

Nested Dictionary:

A dictionary can contain another dictionary as a value.

```
student = {
  "name": "Emily",
  "grades": {"math": 90, "science": 85, "history": 88},
  "address": {"city": "Chicago", "zip": "60601"}
}
print(student["grades"]["math"])  # Output: 90
print(student["address"]["city"])  # Output: Chicago
```

Dictionary Comprehension :

Just like list comprehensions, Python supports dictionary comprehensions.

```
squares = {x: x**2 for x in range(1, 6)}
print(squares)
```

When to Use a Dictionary?

 Use a list when you need an ordered collection and access by index.

 Use a dictionary when you need key-value mapping and fast lookups.

 Use a nested dictionary when dealing with structured data.

4. Python Built-in Set Data Type (set)

A set in Python is an unordered collection of unique elements. It is similar to mathematical sets and provides efficient

operations for set-related tasks like union, intersection, and difference.

Characteristics of Sets:

Unordered → The elements do not follow a specific order.

Unique elements → Duplicates are automatically removed.

Mutable → You can add or remove elements.

Supports mathematical operations like union, intersection, difference.

Elements must be immutable → Can include int, float, str, tuple, but not list or dict.

Creating a Set:

Sets can be created using:

i) Curly braces {}

ii) The set() constructor

```
# Using curly braces
my_set = {10, 20, 30, 40}
print(my_set)
# Output: {40, 10, 20, 30}  (Order may vary)

# Using set() function
my_set2 = set([1, 2, 3, 4, 5])
print(my_set2)
# Output: {1, 2, 3, 4, 5}
```

Removing Duplicates Automatically:

```
duplicate_set = {10, 20, 20, 30, 30, 40}
print(duplicate_set)
# Output: {40, 10, 20, 30}  (Duplicates removed)
```

Creating an Empty Set:

```
empty_set = set()  # □ Correct way
print(type(empty_set))  # Output: <class 'set'>

empty_dict = {}  # □ This creates an empty dictionary, not a set
print(type(empty_dict))  # Output: <class 'dict'>
```

Accessing Set Elements:

Sets do not support indexing or slicing because they are unordered.

```
my_set = {10, 20, 30}
print(my_set[0]) # □ This will raise a TypeError
```

Instead, you can iterate through a set using a for loop:

```
for num in my_set:
    print(num)
```

Modifying a Set:

Adding Elements (add())

```
my_set.add(50)
print(my_set)
# Output: {50, 10, 20, 30} (Order may vary)
```

Adding Multiple Elements (update())

```
my_set.update([60, 70, 80])
print(my_set)
```

Removing Elements:

```
my_set.remove(30)  # Removes 30; raises an error if the element is not found
print(my_set)

my_set.discard(100)  # Does nothing if 100 is not found (no error)
print(my_set)

popped_value = my_set.pop()  # Removes a random element
print(popped_value)

my_set.clear()  # Removes all elements
print(my_set)  # Output: set()
```

Set Operations:

Python sets support powerful mathematical operations.

Union (| or union())

Returns all unique elements from both sets.

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}

print(A | B)  # Output: {1, 2, 3, 4, 5, 6}
```

print(A.union(B)) # Output: {1, 2, 3, 4, 5, 6}

Intersection (& or intersection()):

Returns common elements.

print(A & B) # Output: {3, 4}

print(A.intersection(B)) # Output: {3, 4}

Difference (- or difference()):

Returns elements in A that are not in B.

print(A - B) # Output: {1, 2}

print(A.difference(B)) # Output: {1, 2}

Symmetric Difference (^ or symmetric_difference()):

Returns elements that are in either set but not both.

print(A ^ B) # Output: {1, 2, 5, 6}

print(A.symmetric_difference(B)) # Output: {1, 2, 5, 6}

Checking Subsets and Supersets:

X = {1, 2, 3}

Y = {1, 2, 3, 4, 5}

print(X.issubset(Y)) # Output: True (X is inside Y)

print(Y.issuperset(X)) # Output: True (Y contains all elements of X)

Checking Membership:

my_set = {10, 20, 30}

print(10 in my_set) # Output: True

print(40 in my_set) # Output: False

Frozen Set (frozenset):

A frozenset is an immutable version of a set.

fs = frozenset([1, 2, 3, 4])

```
print(fs)

# fs.add(5)  ▯ This will raise an AttributeError
```

Mutable and Immutable Data Types:

Mutable: Can be changed after creation.

Immutable: Cannot be changed after creation.

Examples:

```
# Immutable (String)
s = "hello"
s[0] = "H"  # This will cause an error!



# Mutable (List)
lst = [1, 2, 3]
lst[0] = 100
print(lst)  # Output: [100, 2, 3]
```

Data Type	Mutable / Immutable
int, float, complex	Immutable
str	Immutable
tuple	Immutable

list	Mutable
set	Mutable
dict	Mutable

Python type() function :

The type() function in Python is used to determine the type of an object. It can also be used to create new classes dynamically.

Syntax:

```
type(object)
```

or

```
type(name, bases, dict)
```

Checking the Type of an Object:

When used with a single argument, type() returns the type of the given object.

Example:

```
print(type(10))       # Output: <class 'int'>
print(type(10.5))     # Output: <class 'float'>
print(type("Hello"))  # Output: <class 'str'>
print(type([1, 2, 3])) # Output: <class 'list'>
print(type((1, 2, 3))) # Output: <class 'tuple'>
```

```python
print(type({1, 2, 3})) # Output: <class 'set'>
print(type({'a': 1}))  # Output: <class 'dict'>
```

Creating Classes Dynamically:

When used with three arguments, type(name, bases, dict), type() creates a new class.

```python
type(class_name, base_classes, attributes)
```

i) class_name: Name of the class (string)

ii) base_classes: Tuple containing base classes (for inheritance)

iii) attributes: Dictionary containing attributes and methods

Example:

```python
# Creating a class dynamically
MyClass = type("MyClass", (object,), {"x": 10, "show": lambda self: print(self.x)})

# Creating an instance of the dynamically created class
obj = MyClass()
obj.show()  # Output: 10

print(type(MyClass))  # Output: <class 'type'>
print(type(obj))      # Output: <class '__main__.MyClass'>
```

Python Type Casting (Type Conversion):

Type casting (or type conversion) in Python refers to converting one type of data into another. Python provides two types of type conversion:

i) Implicit Type Conversion – Done automatically by Python.

ii) Explicit Type Conversion – Done manually by the programmer using built-in functions.

1. Implicit Type Conversion (Automatic)

Python automatically converts one data type into another when it makes sense.

Example:

```
num_int = 10       # Integer
num_float = 2.5    # Float

result = num_int + num_float  # Python converts 'num_int' to float

print(result)     # Output: 12.5
print(type(result))  # Output: <class 'float'>
```

Explicit Type Conversion (Manual):

The programmer manually converts data from one type to another using built-in functions.

Common Type Casting Functions:

Function	Converts to
int(x)	Integer
float(x)	Floating point
str(x)	String
list(x)	List
tuple(x)	Tuple
set(x)	Set

dict(x)   Dictionary

Examples:

a) Converting String to Integer:

```
num_str = "100"
num_int = int(num_str)  # Converts string to integer

print(num_int)      # Output: 100
print(type(num_int))  # Output: <class 'int'>
```

b) Converting Float to Integer

```
num_float = 9.99
num_int = int(num_float)  # Converts float to integer (removes decimal part)

print(num_int)  # Output: 9
```

c) Converting Integer to String

```
num = 50
num_str = str(num)  # Converts integer to string

print(num_str)       # Output: "50"
print(type(num_str))  # Output: <class 'str'>
```

d) Converting List to Tuple

```
my_list = [1, 2, 3]
my_tuple = tuple(my_list)  # Converts list to tuple

print(my_tuple)  # Output: (1, 2, 3)
```

e) Converting List to Set

```
my_list = [1, 2, 2, 3, 4]
my_set = set(my_list)  # Removes duplicates

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

f) Converting List of Tuples to Dictionary

```
my_list = [("name", "Alice"), ("age", 25)]
my_dict = dict(my_list)

print(my_dict)  # Output: {'name': 'Alice', 'age': 25}
```

Python String Literals – Single and Triple Quoted Strings

In Python, string literals are sequences of characters enclosed in quotes. Python provides several ways to define strings using single quotes ('), double quotes ("), and triple quotes (''' or """).

1. Single-Quoted Strings ('...')

Strings can be enclosed in single quotes (').

Useful for short, simple strings.

If the string contains a single quote (') inside, use an escape character (\) or switch to double quotes.

Example:

```
string1 = 'Hello, Python!'
print(string1)  # Output: Hello, Python!
```

Handling Apostrophes (') in Single-Quoted Strings:

```
string2 = 'It\'s a beautiful day!'  # Using backslash (\) to escape '
print(string2)  # Output: It's a beautiful day!
```

or

```
string3 = "It's a beautiful day!"  # Using double quotes to avoid escaping
print(string3)  # Output: It's a beautiful day!
```

2. Double-Quoted Strings ("...")

Work the same way as single-quoted strings.

Allow easier handling of single quotes (') inside the string.

Example:

```
string4 = "Python is fun!"
print(string4)  # Output: Python is fun!
```

Handling Quotes Inside:

```
string5 = "She said, \"Python is awesome!\""
print(string5)  # Output: She said, "Python is awesome!"
```

Or

```
string6 = 'She said, "Python is awesome!"'  # Using single quotes outside
print(string6)  # Output: She said, "Python is awesome!"
```

Triple-Quoted Strings ('''...''' or """...""")

Used for multiline strings.

Can contain both single and double quotes without escaping.

Often used for docstrings (multi-line comments in Python functions).

Example (Multiline String):

```
multi_string = '''This is a
multiline string
in Python.'''
print(multi_string)
```

Using Triple Quotes to Avoid Escape Characters:

```
quote_string = '''She said, "It's a wonderful day!"'''
print(quote_string)
```

Key Differences

Type	Example	Supports Multi-line?	Requires Escaping?

Single-quoted ('...')	'Hello'	No	Yes, for ' inside
Double-quoted ("...")	"Hello"	No	Yes, for " inside
Triple-quoted (''' ... ''' or """...""")	'''Hello'''	Yes	No

Escape Sequences:

Eescape sequences are special character combinations that represent certain special

characters or actions within strings. They begin with a backslash (\) followed by a character

that indicates a specific action.

Here are some common escape sequences in Python:

\n: Newline

This escape sequence represents a newline character. When used in a string, it moves the text following it to the next line.

```
print("Hello\nWorld")  # Output:  Hello
                       # World
```

\t: Tab

This adds a tab space in the string, which is often used for indentation.

```
print("Hello\tWorld")  # Output: Hello   World
```

\\: Backslash

Since the backslash \ is used as an escape character, you need to use a double backslash to

represent an actual backslash in a string.

```
print("This is a backslash: \\")
```

```
# Output:
# This is a backslash: \
```

\': Single Quote

This allows you to include a single quote inside a string that is enclosed in single quotes.

```
print('It\'s a beautiful day.')
# Output:
# It's a beautiful day.
```

\": Double Quote

This allows you to include a double quote inside a string that is enclosed in double quotes.

```
print("He said, \"Hello!\"")
# Output:
# He said, "Hello!"
```

\r: Carriage Return

This moves the cursor to the beginning of the line. It doesn't move to the next line; it

simply overwrites the current line.

```
print("Hello\rWorld")
# Output:
# World (since "World" overwrites "Hello")
```

\b: Backspace

This deletes the previous character in the string.

```
print("Hello\bWorld")
# Output:
```

# HellWorld (The "o" is deleted by the backspace)

\f: Form Feed

This is used to move the cursor to the next page in some old printers. In most cases, it's

not commonly used in modern systems.

```
print("Hello\fWorld")
# Output: May print Hello and World with a form feed in between.
```

\v: Vertical Tab

Similar to the horizontal tab \t, but it moves the cursor vertically.

```
print("Hello\vWorld")
# Output: Moves "World" vertically relative to "Hello", though the effect might not be visible in all environments.
```

\ooo: Octal value

A sequence of three digits (ooo) represents an octal value.

```
print("\101")  # Octal value for 'A'
# Output: A
```

\xhh: Hexadecimal value

A two-digit hexadecimal value (hh) is used to represent a character.

```
print("\x48\x65\x6c\x6c\x6f")  # Hexadecimal for "Hello"
# Output: Hello
```

\uXXXX: Unicode character (4-digit)

This is used to represent a Unicode character using its 4-digit hexadecimal code.

```
print("\u0048\u0065\u006c\u006c\u006f")  # Unicode for "Hello"
# Output: Hello
```

\UXXXXXXXX: Unicode character (8-digit)

A similar to \uXXXX, but used for characters that require 8 digits.

```
print("\U0001F600")  # Unicode for    (Grinning Face emoji)
# Output:
```

These escape sequences are useful when working with strings that contain special characters or when formatting text for output.

f-Strings :

In Python, f-strings (formatted string literals) provide a concise and efficient way to embed

expressions inside string literals. They were introduced in Python 3.6 and are denoted by

placing an f or F before the opening quote of the string, followed by curly braces {}

containing expressions.

Basic Syntax:

```
name = "John"
age = 30
greeting = f"Hello, my name is {name} and I am {age} years old."
print(greeting)
# Output: Hello, my name is John and I am 30 years old.
```

In this example, the name and age variables are inserted into the string using {} within the f"" string literal.

Key Features of f-strings:

Expression Evaluation:

You can include any valid Python expression inside the curly braces, and it will be evaluated

at runtime.

```
x = 5
y = 10
result = f"The sum of x and y is {x + y}."
print(result)
# Output: The sum of x and y is 15.
```

Variable Substitution:

f-strings automatically replace the variable names with their values.

```
username = "alice"
score = 95
message = f"{username} scored {score} in the exam."
print(message)
# Output: alice scored 95 in the exam.
```

Calling Functions inside f-strings:

You can call functions inside the curly braces.

```
def double(n):
   return n * 2
```

```
number = 7
result = f"The double of {number} is {double(number)}."
print(result)
# Output: The double of 7 is 14.
```

Formatting Numbers:

f-strings allow you to format numbers easily, such as controlling decimal places, padding, and more.

```
pi = 3.141592653589793
formatted = f"Value of pi up to 2 decimal places: {pi:.2f}"
print(formatted)
# Output: Value of pi up to 2 decimal places: 3.14
```

Using !r for Repr:

You can use the !r format specifier to get the repr() of an object, which shows the string

representation of the object (useful for debugging).In this example, :.2f formats the

floating-point number to two decimal places.

```
my_list = [1, 2, 3]
print(f"The list is: {my_list!r}")
# Output: The list is: [1, 2, 3]
```

Using !s for Str:

You can also explicitly convert a value to a string using the !s specifier.

```
value = 42
print(f"The value as string: {value!s}")
# Output: The value as string: 42
```

f-string Advantages:

Concise and Readable: f-strings make string interpolation more readable and less error-prone compared to other methods like % formatting or str.format().

Performance: f-strings are faster than the str.format() method because they evaluate expressions directly at runtime, rather than parsing and handling additional method calls.

More Flexible: f-strings allow you to embed complex expressions and function calls directly inside the string.

r-string :

In Python, an r-string (or raw string) is a string prefixed with an r or R (e.g., r"string" or R"string"). The primary feature of raw strings is that they treat backslashes (\) as literal characters, meaning that they do not escape any characters following the backslash.

This is useful when working with strings that contain backslashes, such as regular expressions, file paths, or Windows paths, where you don't want to manually escape each backslash.

Basic Syntax of r-string:

```
raw_string = r"This is a raw string: C:\Users\Name"

print(raw_string)

# Output: This is a raw string: C:\Users\Name
```

In this example, the raw string treats the backslashes \ as literal characters, so you don't

need to escape them as you would in a normal string.

Key Features of r-strings:

No Escape Sequences: In a raw string, escape sequences like \n, \t, \r, \\, etc., are not processed. They are treated as regular characters in the string.

```
normal_string = "This is a newline\nSecond line"
raw_string = r"This is a raw string\nSecond line"

print(normal_string)
# Output:
# This is a newline
# Second line

print(raw_string)
# Output: This is a raw string\nSecond line
```

In the normal_string, the \n is treated as a newline character. But in the raw_string, \n is

treated as two separate characters (\ and n), not a special escape sequence.

Useful for Regular Expressions: Regular expressions (regex) in Python often contain

backslashes, so raw strings are especially useful in this case to avoid escaping backslashes.

```
import re

pattern = r"\d+"  # Matches one or more digits
text = "There are 123 apples."
match = re.search(pattern, text)

if match:
   print(match.group())
# Output: 123
```

Without using a raw string, you would have to double the backslashes, like "\\d+", which is less readable.

File Paths (Windows): In Windows file paths, backslashes are used to separate directories, so raw strings prevent the need to escape each backslash.

```
file_path = r"C:\Users\Name\Documents\file.txt"
```

```
print(file_path)
# Output: C:\Users\Name\Documents\file.txt
```

If this were a regular string, you would need to write the path as

"C:\\Users\\Name\\Documents\\file.txt".

Ending with a Single Backslash: One important thing to note is that raw strings cannot end

with a single backslash (\). If you try to do that, Python will raise a SyntaxError because

the backslash is considered an escape character for the closing quote. For example:

```
# This will cause a SyntaxError:
invalid_raw_string = r"Some path here\"
```

To avoid this, you would need to escape the backslash like this:

```
valid_raw_string = r"Some path here\\"
```

Example Usage:

1. Using r-strings for Regular Expressions:

```
import re
pattern = r"\b\w+\b"  # Word boundary and words
text = "Hello, world!"
words = re.findall(pattern, text)
print(words)
# Output: ['Hello', 'world']
```

2. Using r-strings for File Paths:

```
file_path = r"C:\Program Files\MyApp\file.txt"
print(file_path)
# Output: C:\Program Files\MyApp\file.txt
```

r-strings are particularly useful when you need to work with strings that contain backslashes and you don't want Python to treat them as escape sequences. They're commonly used in regular expressions, file paths, and when working with raw string data that includes backslashes.