**Operators in Python**

Operators perform specific computations on values. Python offers the following categories:

* **Arithmetic Operators:**
  + +: Addition (e.g., 5 + 3 = 8)
  + -: Subtraction (e.g., 10 - 2 = 8)
  + \*: Multiplication (e.g., 4 \* 5 = 20)
  + /: Division (e.g., 12 / 3 = 4)
  + %: Modulo (remainder after division; e.g., 11 % 3 = 2)
  + //: Floor division (integer quotient; e.g., 11 // 3 = 3)
  + \*\*: Exponentiation (e.g., 2 \*\* 3 = 8)
* **Comparison Operators:**
  + ==: Equal to (e.g., x == 5 checks if x equals 5)
  + !=: Not equal to (e.g., y != 10 checks if y is not 10)
  + >, <, >=, <=: Greater than, less than, greater than or equal to, less than or equal to (e.g., for comparing values)
* **Logical Operators:**
  + and: Returns True only if both operands are True (e.g., (x > 0) and (y < 10))
  + or: Returns True if at least one operand is True (e.g., (a == 1) or (b == 2))
  + not: Inverts the truth value (e.g., not (z is None))
* **Assignment Operators:**
  + =: Simple assignment (e.g., age = 25)
  + +=, -=, \*=, /=, %=, //=, \*\*=: Combined assignment and operation (e.g., count += 1 is equivalent to count = count + 1)
* **Bitwise Operators (work on binary representations of numbers):**
  + &: Bitwise AND (e.g., 5 & 3 results in 1)
  + |: Bitwise OR (e.g., 5 | 3 results in 7)
  + ^: Bitwise XOR (e.g., 5 ^ 3 results in 6)
  + ~: Bitwise NOT (e.g., ~5 results in -6)
  + <<, >>: Left and right shift (e.g., 5 << 2 shifts bits left by 2)
* **Membership Operators:**
  + in: Checks if a value is present in a sequence (e.g., 'apple' in fruits)
  + not in: Checks if a value is not present (e.g., 5 not in numbers)

**Built-in Keywords and Methods in Python**

* **Keywords** are reserved words with specific meanings in Python. You cannot use them as variable names:
  + if, else, for, while: Control flow
  + def, class: Defining functions and classes
  + import: Importing modules
  + return: Returning values from functions
  + True, False, None: Boolean and null values
* **Methods** are functions that are attached to objects. They perform operations specific to those objects:
  + len(sequence): Returns the length of a sequence
  + type(object): Returns the type of an object
  + print(object): Prints an object to the console
  + input(): Reads user input as a string
  + range(start, stop, step): Generates a sequence of numbers
  + str(object): Converts an object to a string
  + int(object): Converts an object to an integer
  + float(object): Converts an object to a floating-point number
  + list(): Creates a list
  + dict(): Creates a dictionary
  + set(): Creates a set
  + tuple(): Creates a tuple

**Data Types in Python**

Data types specify the kind of value a variable can hold:

* **Numeric Types:**
* int **(integer):** Represents whole numbers (positive, negative, or zero).
  + Example: age = 30
  + Application: Storing quantities, counting items, performing calculations.
* float **(floating-point number):** Represents real numbers with a decimal point.
  + Example: pi = 3.14159
  + Application: Scientific calculations, measurements with decimals.
* complex**:** Represents complex numbers with a real and imaginary part (a + bi, where i is the imaginary unit).
  + Example: z = 3 + 4j
  + Application: Advanced mathematical calculations involving imaginary numbers.

**Sequence Types**

Sequences are ordered collections of items that can be accessed by index.

* list**:** A mutable (changeable) sequence enclosed in square brackets [].
  + Example: fruits = ["apple", "banana", "orange"]
  + Application: Storing collections of items that can be modified, like shopping lists or to-do lists.
* tuple**:** An immutable (unchangeable) sequence also enclosed in parentheses ().
  + Example: coordinates = (10, 20)
  + Application: Storing fixed data that doesn't need to be modified, like coordinates or configuration settings.
* range**:** Generates a sequence of numbers within a specified range. Often used in loops.
  + Example: for i in range(5): print(i) (prints 0, 1, 2, 3, 4)
  + Application: Iterating through a sequence of numbers for calculations or loops.

**Text Type**

* str **(string):** Represents textual data enclosed in single or double quotes.
  + Example: name = "Alice"
  + Application: Storing text, user input, messages, etc.

**Mapping Type**

* dict **(dictionary):** An unordered collection of key-value pairs enclosed in curly braces {}. Keys must be unique and immutable.
  + Example: person = {"name": "Bob", "age": 35}
  + Application: Storing data associated with labels or keys, like user profiles or configuration settings.

**Set Types**

Sets are unordered collections of unique items.

* set**:** A mutable set enclosed in curly braces {}.
  + Example: unique\_numbers = {1, 2, 2, 3} (results in {1, 2, 3})
  + Application: Removing duplicates from data, performing set operations like union and intersection.
* frozenset**:** An immutable version of a set.
  + Example: my\_frozenset = frozenset({4, 5, 6})
  + Application: When you need a set that cannot be modified, like representing valid options in a menu.

**Boolean Type**

* bool**:** Represents logical values: True or False.
  + Example: is\_loggedin = True
  + Application: Making decisions in conditional statements (if/else).

**Binary Types (for working with binary data)**

* bytes**:** Represents immutable sequences of 8-bit bytes.
  + Example: data = b"Hello, world!"
  + Application: Storing binary data like image files or network data.
* bytearray**:** A mutable version of bytes.
  + Example: buffer = bytearray(10)
  + Application: When you need to modify binary data.
* memoryview**:** Provides a view of a memory location.
  + Example: view = memoryview(data)
  + Application: Advanced memory manipulation for specialized tasks.

**Data Structures in Python with Code Examples**

Data structures are specialized formats for organizing, processing, retrieving, and storing data efficiently. Python offers a variety of built-in data structures to suit different needs. Here's a closer look at some common ones with code examples:

**1. Lists**

* Ordered, mutable collection of items enclosed in square brackets [].
* Elements can be of different data types.
* Use cases: Storing shopping lists, to-do lists, scores, or any collection that needs to be modified.

Python

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

# Access elements by index (starts from 0)

first\_fruit = fruits[0] # Access the first element (apple)

print(first\_fruit) # Output: apple

# Modify elements

fruits[1] = "mango" # Replace banana with mango

print(fruits) # Output: ['apple', 'mango', 'cherry']

# Add elements

fruits.append("kiwi") # Add kiwi to the end

print(fruits) # Output: ['apple', 'mango', 'cherry', 'kiwi']

# Remove elements

fruits.remove("cherry") # Remove cherry

print(fruits) # Output: ['apple', 'mango', 'kiwi']

# Iterate through elements

for fruit in fruits:

print(fruit) # Prints each fruit on a new line

# Slicing (extract a sublist)

sublist = fruits[1:3] # Extract mango and kiwi

print(sublist) # Output: ['mango', 'kiwi']

**2. Tuples**

* Ordered, immutable collection of items enclosed in parentheses ().
* Elements cannot be changed after creation.
* Use cases: Storing fixed data like coordinates, configurations, or representing data that shouldn't be modified.

Python

# Create a tuple of coordinates

coordinates = (10, 20)

print(coordinates[0]) # Access the first element (10)

# Tuples can hold mixed data types

person = ("Alice", 30, "blue") # Name, age, and favorite color

# You cannot modify elements like lists

# coordinates[0] = 15 # This will cause a TypeError

**3. Dictionaries**

* Ordered collection of key-value pairs enclosed in curly braces {}.
* Keys must be unique and immutable (like strings or numbers).
* Values can be of any data type.
* Use cases: Storing user profiles, phonebook entries, configurations, or any data where you need to access items by labels (keys).

Python

# Create a dictionary for a phonebook entry

phonebook = {"Alice": "123-456-7890", "Bob": "987-654-3210"}

# Access elements by key

alice\_number = phonebook["Alice"]

print(alice\_number) # Output: 123-456-7890

# Add new entries

phonebook["Charlie"] = "555-123-4567"

print(phonebook) # Output: {'Alice': '123-456-7890', 'Bob': '987-654-3210', 'Charlie': '555-123-4567'}

# Update existing entries

phonebook["Alice"] = "098-765-4321"

print(phonebook) # Output: {'Alice': '098-765-4321', 'Bob': '987-654-3210', 'Charlie': '555-123-4567'}

**4. Sets**

* Unordered collection of unique items enclosed in curly braces {}.
* Elements can be of any data type, but must be hashable (like strings or numbers).
* Use cases: Removing duplicates from a list, checking membership, performing set operations like union and intersection.

Python

# Create a set of numbers

numbers = {1, 2, 2, 3, 4} # Duplicates are automatically removed

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

# Check if an element is present

if 5 in numbers:

print("5 is in the set")

else:

print("5 is not in the set")

# Add elements to a set

numbers.add(5)

print(numbers)