PYTHON

LIST

- A list is versatile and fundamental data structure that allows to store a collection of items.
- List are mutable
- They can hold elements of different data types and the elements can be accessed using indices.
- They are denoted by []

```
my_list_int = [1, 2, 3, 4, 5] # Creating a list

my_list_str = ["apple", "banana", "cherry"] # Creating a list of strings

my_list_mixed = [10, "hello", 3.14, True] # Creating a list with mixed data types

empty_list = [] # Creating an empty list

squares = [x**2 for x in range(1, 6)] # Creating a list using list comprehension
```

#List indexing

```
#create a list
My_list = [10, 20, 30, 40, 50]

# Accessing elements using positive indexing
print(my_list[0]) # Output: 10
print(my_list[2]) # Output: 30
print(my_list[4]) # Output: 50

# Accessing elements using negative indexing
print(my_list[-1]) # Output: 50 (Last element)
print(my_list[-3]) # Output: 30 (Third element from the end)
```

•Positive indices start from 0 for the first element and increment by 1 for each subsequent element.
•Negative indices start from -1 for the last element and decrement by 1 for each element from the end.

```
#slicing lists

# Creating a list
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Slicing the list to get a sublist
sublist = my_list[2:7] # Elements from index 2 (inclusive) to index 7 (exclusive)
print(sublist) # Output: [3, 4, 5, 6, 7]

# Slicing with step
step_slice = my_list[::2] # Every second element, starting from the beginning
print(step_slice) # Output: [1, 3, 5, 7, 9]

# Negative slicing
neg_slice = my_list[-5:-2] # Elements from the fifth-to-last to the second-to-last
print(neg_slice) # Output: [6, 7, 8]

# Slicing with negative step
neg_step_slice = my_list[::-1] # Reversing the list using a negative step
print(neg_step_slice) # Output: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

#modifying lists and list methods

# Creating a list
my_list = [1, 2, 3, 4, 5]

# Modifying list elements
my_list[2] = 10
print(my_list) # Output: [1, 2, 10, 4, 5]

# Appending an element to the end of the list
my_list.append(6)
print(my_list) # Output: [1, 2, 10, 4, 5, 6]
```

PYTHON

```
# Extending the list with another list
my_list.extend([7, 8, 9])
print(my_list) # Output: [1, 2, 10, 4, 5, 6, 7, 8, 9]

# Inserting an element at a specific index
my_list.insert(3, 11)
print(my_list) # Output: [1, 2, 10, 11, 4, 5, 6, 7, 8, 9]

# Removing an element by its value
my_list.remove(4)
print(my_list) # Output: [1, 2, 10, 11, 5, 6, 7, 8, 9]

# Removing the last element and getting its value
last_element = my_list.pop()
print(last_element) # Output: 9
print(my_list)      # Output: [1, 2, 10, 11, 5, 6, 7, 8]

# Sorting the list in ascending order
 my_list.sort()
print(my_list) # Output: [1, 2, 5, 6, 7, 8, 10, 11]

# Reversing the elements of the list
 my_list.reverse()
print(my_list) # Output: [11, 10, 8, 7, 6, 5, 2, 1]

# finding the length of the list
my_list = [1, 2, 3, 4, 5]
length = len(my_list)
print(length) # Output: 5

#checking membership in a list
The 'in' keyword is a useful way to determine whether an element exists in alist
or any other iterable object

my_list = [1, 2, 3, 4, 5]

# Checking if an element is present in the list
element_1 = 3
element_2 = 6
```

```
# Using 'in' keyword to check membership
is_element_1_present = element_1 in my_list
is_element_2_present = element_2 in my_list
# Printing the results
print(is_element_1_present) # Output: True
print(is_element_2_present) # Output: False


# list comprehensions

# Example 1: Creating a list of squares using a for loop
squares_for_loop = []
for x in range(1, 6): squares_for_loop.append(x**2)
print(squares_for_loop) # Output: [1, 4, 9, 16, 25]

# Example 2: Creating the same list using list comprehension
squares_list_comprehension = [x**2 for x in range(1, 6)]
print(squares_list_comprehension) # Output: [1, 4, 9, 16, 25]

#copying Lists

original_list = [1, 2, 3]

# Method 1: Using the copy() method to create a shallow copy
copy_1 = original_list.copy()

# Method 2: Using list slicing to create a shallow copy
copy_2 = original_list[:]

# Modifying the shallow copies (won't affect the original list)
copy_1[0] = 10 copy_2[2] = 30

# Printing the original list and the shallow copies
print("Original List:", original_list) # Output: Original List: [1, 2, 3]
print("Shallow Copy (using copy()):", copy_1) # Output: Shallow Copy (using
copy()): [10, 2, 3]
print("Shallow Copy (using list slicing):", copy_2) # Output: Shallow Copy
(using list slicing): [1, 2, 30]
```

PYTHON

TUPLE

- A Tuple is an ordered, immutable collection of elements
- This means once a tuple is created, its elements cannot be changed, added or removed
- A tuple is defined using parentheses '**()**' and can contain elements of different data types

```
# Empty tuple
empty_tuple = ()

# Tuple with elements my_tuple = (1, 2, 3, 'hello', True)

# accessing elements
my_tuple = (1, 2, 3, 'hello', True)

print(my_tuple[0]) # Output: 1
print(my_tuple[3]) # Output: 'hello'
print(my_tuple[-1]) # Output: True (Negative indexing starts from the end)

# slicing
my_tuple = (1, 2, 3, 'hello', True)

print(my_tuple[1:4]) # Output: (2, 3, 'hello')
print(my_tuple[:3]) # Output: (1, 2, 3)
print(my_tuple[2:]) # Output: (3, 'hello', True)

# tuple length
my_tuple = (1, 2, 3, 'hello', True)
print(len(my_tuple)) # Output: 5

# tuple concatenation
tuple1 = (1, 2, 3)
tuple2 = ('a', 'b')

concatenated_tuple = tuple1 + tuple2
print(concatenated_tuple) # Output: (1, 2, 3, 'a', 'b')
```

```
# tuple repetition
my_tuple = (1, 2, 3)

repeated_tuple = my_tuple * 3
print(repeated_tuple)

# Iterating over a Tuple
my_tuple = (1, 2, 3, 'hello', True)

for item in my_tuple:
      print(item)

#output
1
2
3
hello
True
```

#Tuple unpacking

- Tuple unpacking allows to assign the individual element of a tuple to separate variables in a single line
- The number of variables on the left side of the assignment must match the number of elements in the tuple

```
my_tuple = (1, 2, 3) # Unpacking the tuple into separate variables
a, b, c = my_tuple
print(a, b, c) # Output: 1 2 3

my_tuple = (1, 2, 3)
# Incorrect: Number of variables doesn't match the number of elements in the tuple
# This will raise a ValueError
 a, b = my_tuple
```

PYTHON

```
# checking for membership
my_tuple = (1, 2, 3, 'hello', True)

print(3 in my_tuple) # Output: True
print('world' in my_tuple) # Output: False

# tuple methods
Since tuples are immutable, they have only two basic methods
```

- Count(x): returns the number of occurrences of the element ‘x’ in the tuple.
- Index(x): returns the index of the first occurrence of element ‘x’ in the tuple

```
my_tuple = (1, 2, 3, 2, 4)

print(my_tuple.count(2)) # Output: 2 (number of occurrences of 2)
print(my_tuple.index(3)) # Output: 2 (index of the first occurrence of 3)
```

DICTIONARY

- A dictionary is a versatile and powerful data structure that allows to store and retrieve data in a **key-value** format
- Also known as associative array or hash map in other programming languages
- Dictionaries are implemented using a hash table, which provides efficient key-based lookups, insertions and deletions

- A dictionary is created using curly braces ‘{}’
- **Key-value** pairs are separated by colons ‘:’

```
my_dict = {'key1': 'value1', 'key2': 'value2', 'key3': 'value3’}
```

- Dictionary keys must be unique and immutable(eg: strings, numbers, tuples).
- Lists or other dictionaries cannot be used as keys because they are mutable

```
# accessing values
value = my_dict['key2’]
print(value) # Output: value2
```

```
# adding and updating items
my_dict = {'key1': 'value1', 'key2': 'value2', 'key3': 'value3’}

my_dict['new_key'] = 'new_value’
my_dict['key1'] = 'updated_value’
print(my_dict) # Output: {'key1': 'updated_value', 'key2': 'value2', 'key3':
'value3', 'new_key': 'new_value’}


# removing items
my_dict = {'key1': 'updated_value', 'key2': 'value2', 'key3': 'value3',
'new_key': 'new_value’}

# Using 'del' keyword to remove 'key3’
del my_dict['key3’]

# Using 'pop()' method to remove 'key2
my_dict.pop('key2’)

print(my_dict) # Output: {'key1': 'updated_value', 'new_key': 'new_value’}

# Trying to access 'key3' after removal (will raise KeyError)
value = my_dict['key3'] # Raises KeyError: 'key3’

# Trying to access 'key2' after removal (will raise KeyError)
value = my_dict['key2'] # Raises KeyError: 'key2’

# Dictionary methods
my_dict = {'name': 'John', 'age': 30, 'city': 'New York’}

# keys(): Returns a list of all keys
all_keys = my_dict.keys()
print(all_keys) # Output: dict_keys(['name', 'age', 'city’])

# values(): Returns a list of all values
all_values = my_dict.values()
print(all_values) # Output: dict_values(['John', 30, 'New York’])
```

PYTHON

```python
# items(): Returns a list of tuples containing key-value pairs
all_items = my_dict.items()
print(all_items)
# Output: dict_items([('name', 'John'), ('age', 30), ('city', 'New York')])

# get(key, default): Returns the value associated with the key, or a default value if the key is not found
name = my_dict.get('name', 'Unknown’)
occupation = my_dict.get('occupation', 'Unemployed’)
print(name) # Output: John
print(occupation) # Output: Unemployed

# clear(): Removes all items from the dictionary
my_dict.clear()
print(my_dict) # Output: {}

# copy(): Creates a shallow copy of the dictionary
original_dict = {'key1': 'value1', 'key2': 'value2’}
copied_dict = original_dict.copy()
print(copied_dict) # Output: {'key1': 'value1', 'key2': 'value2’}

# Dictionary comprehension
squares = {x: x**2 for x in range(1, 6)}

print(squares) # Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# checking key existence
my_dict = {'key1': 'value1', 'key2': 'value2', 'key3': 'value3’}

if 'key1' in my_dict:
     print("'key1' exists in the dictionary.")
else:
     print("'key1' does not exist in the dictionary.") # output - 'key1' exists in the dictionary.

if 'key4' in my_dict:
    print("'key4' exists in the dictionary.")
else:
    print("'key4' does not exist in the dictionary.") # output - 'key4' does not exist in the dictionary.
```

```python
# length of a dictionary

my_dict = {'key1': 'value1', 'key2': 'value2', 'key3': 'value3’}

num_items = len(my_dict)
print(num_items) # Output: 3

# iterating over a dictionary
my_dict = {'name': 'John', 'age': 30, 'city': 'New York’}

# Iterating over keys and printing keys and corresponding values
for key in my_dict:
      print(key, my_dict[key])
#output
name John
age 30
city New York

# Iterating over values and printing each value
for value in my_dict.values():
     print(value)
#output
John
30
New York

# Iterating over key-value pairs and printing each key and its corresponding value
for key, value in my_dict.items():
      print(key, value)

#output
name John
age 30
city New York
```

PYTHON

SETS

- A set is an unordered collection of unique elements
- It is defined using curly braces **{}** or the built-in **'set()'** constructor
- Sets are similar to lists and tuples, but they donot allow duplicate elements and their elements are not indexed
- Sets are particularly useful when needed to store unique elements and perform operations like union, intersection, difference etc

```
# empty set
empty_set = set()

# Set with elements
my_set = {1, 2, 3, 4, 5}

# creating set using set() constructor
my_set = set([1, 2, 3, 4, 5])

print(my_set)      #output: {1, 2, 3, 4, 5}

# adding elements
my_set = {1, 2, 3, 4, 5}
my_set.add(6)

print(my_set)       #output: {1, 2, 3, 4, 5, 6}

# removing elements
my_set = {1, 2, 3, 4, 5}
my_set.remove(3)
my_set.discard(4)

print(my_set)        #output: {1, 2, 5}

set1 = {1, 2, 3}
set2 = {3, 4, 5}
```

```
# Set operations
# Union of sets set1 and set2
union_set = set1 | set2
# Output: {1, 2, 3, 4, 5}
# The union_set contains all the elements from both set1 and set2 without any duplicates.

# Intersection of sets set1 and set2
intersection_set = set1 & set2
# Output: {3}
# The intersection_set contains only the elements that are present in both set1 and set2.

# Difference between sets set1 and set2
difference_set = set1 - set2
# Output: {1, 2}
# The difference_set contains elements that are in set1 but not in set2.

# Symmetric Difference between sets set1 and set2
symmetric_difference_set = set1 ^ set2
# Output: {1, 2, 4, 5}
# The symmetric_difference_set contains elements that are in either set1 or set2 but not in both.

# Subset relationship check
is_subset = set1.issubset(set2)
# Output: False
# set1 is not a subset of set2, as set2 has elements {4, 5} that are not present in set1.

# Superset relationship check
is_superset = set1.issuperset(set2)
# Output: False # set1 is not a superset of set2, as set1 lacks elements {4, 5} that are present in set2.
```

PYTHON

```
# Set methods

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

# Length of the set
length_of_set = len(my_set)      # Output: 5

# Check if an element is present in the set
element_exists = 3 in my_set # Output: True

# Clear all elements from the set
my_set.clear() # Output: set()

# Copy the set
new_set = my_set.copy() # Output: set() # The copy() method creates a shallow
copy of the set and assigns it to 'new_set'. # Since 'my_set' was cleared and
became an empty set, 'new_set' is also an empty set.

# Pop an element from the set
popped_element = my_set.pop() # Output: Raises KeyError if the set is empty
# The pop() method removes and returns an arbitrary element from the set.
# However, if the set is empty, it raises a KeyError. Since we cleared 'my_set',
this line will raise a KeyError.

#iterating over sets
my_set = {1, 2, 3, 4, 5}
for element in my_set:
      print(element)
#possible output
2
3
1
4
5
#The output might vary due to the unordered nature of sets. Each time the loop
may get a different order of elements.
```

```
# frozenset
A ‘frozenset’ is an immutable version of a set. Once created, cannot be modified

frozen_set = frozenset([1, 2, 3])
print(frozen_set)

#output
frozenset({1, 2, 3})
```

CONDITIONAL EXPRESSIONS

- Conditional expressions are a way to write concise and inline if-else statements
- They are also known as “ternary operators”

```
 value_if_true if condition else value_if_false

#The   ‘condition’ is evaluated and if it is ‘True’ the ‘value_if_true’ is
returned
#Otherwise, the ‘value_if_false’ is returned
#The result of the of the expression can be assigned to a variable or used
directly in an expression


# example
x = 10
y = 20

result = "x is greater" if x > y else "y is greater"
print(result)  Output: "y is greater"
```

PYTHON

RELATIONAL OPERATORS

OPERATOR	DESCRIPTION	EXAMPLE	RESULT
>	GREATER THAN	5 > 3	TRUE
<	LESS THAN	5 < 3	FALSE
>=	GREATER THAN OR EQUAL TO	5 >= 5	TRUE
<=	LESS THAN OR EQUAL TO	3 <= 5	TRUE
==	EQUAL TO	5 == 5	TRUE
!=	NOT EQUAL TO	5 != 3	TRUE

These operators are used to compare values and return Boolean results based on the comparison.

```
# Example:
a = 5
b = 7
print(a > b) # Output: False
print(a < b) # Output: True
print(a == b) # Output: False
```

LOGICAL OPERATORS

Logical operators are used to combine multiple conditions

OPERATOR	DESCRIPTION	EXAMPLE	RESULT
and	RETURNS TRUE IF BOTH CONDITIONS ARE TRUE, OTHERWISE FALSE.	5 > 3 and 10 > 5	TRUE
or	RETURNS TRUE IF AT LEAST ONE OF THE CONDITIONS IS TRUE, OTHERWISE FALSE.	5 > 3 or 10 < 5	TRUE
not	RETURNS THE OPPOSITE OF THE CONDITION'S RESULT.	not 5 > 3	FALSE

```
# example 1

x = 5
y = 10

# Using 'and' logical operator
if x > 0 and y < 15:
   print("Both conditions are True")

# Using 'or' logical operator
if x < 0 or y > 20:
   print("At least one condition is True")

# Using 'not' logical operator
if not x == y:
   print("x is not equal to y")


# example 2

x = 15

if x < 10:
    print("x is less than 10")
elif x < 20:
    print("x is between 10 and 20")
else:
    print("x is greater than or equal to 20")
```