In the name of Allah

Hello Everybody

Let’s get started!

List Comprehensions

List comprehensions provide a concise way to create lists based on existing sequences. They can replace for loops for simple operations.

```
squares = [x ** 2 for x in range(10)]
```

List Comprehensions with if statement

List comprehensions can include an if statement as a filter to conditionally include elements in the new list. This allows you to create a new list based on an existing list while applying a condition to filter the elements you want.

```
new_list = [expression for element in
old_list if condition]

# example
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
even_numbers = [x for x in numbers if x
% 2 == 0]
print(even_numbers)
# Output: [2, 4, 6, 8]
```

Dictionary

In Python, a dictionary is a collection of key-value pairs. It's an unordered and mutable data structure that allows you to store and retrieve values based on their associated keys. Dictionaries are often used to represent real-world entities and their properties.

Basic representation

In this example, "name", "age", and "city" are keys, and "Alice", 30, and "Wonderland" are their corresponding values. Keys must be unique within a dictionary, and they are typically strings or numbers.

```
person = {
    "name": "Alice",
    "age": 30,
    "city": "Wonderland"
}
```

Access values

You can access values in a dictionary using their keys

```
print(person["name"])   # Output: Alice
print(person["age"])    # Output: 30
```

Add, update, delete

You can also add, update, or delete key-value pairs in a dictionary

```python
# Adding a new key-value pair
person["job"] = "Engineer"

# Updating an existing value
person["age"] = 31

# Deleting a key-value pair
del person["city"]
```

Methods

Dictionaries support various methods for performing operations like getting keys, values, items, checking for key existence, and more

```
keys = person.keys()
# Returns a list of keys
values = person.values()
# Returns a list of values
items = person.items()
# Returns a list of key-value pairs

print("name" in person)
# Output: True (checks for key
existence)
```

Dictionary Comprehensions

Similar to list comprehensions, dictionary comprehensions let you create dictionaries in a compact manner.

```
squares_dict = {x: x ** 2 for x in
range(10)}
```

Iterating over Dict

To iterate over a dictionary in a for loop, you can use the dictionary's keys, values, or items (key-value pairs) as the iteration source. Here are three different ways to write a for loop to iterate over a dictionary

1. Iterating Over Keys: You can use the dictionary's .keys() method to get a list-like view of its keys and then iterate over them.

```
person = {
    "name": "Alice",
    "age": 30,
    "occupation": "Engineer"
}

for key in person.keys():
    print(key)
```

Iterating over Dict

To iterate over a dictionary in a for loop, you can use the dictionary's keys, values, or items (key-value pairs) as the iteration source. Here are three different ways to write a for loop to iterate over a dictionary

2. Iterating Over Values: You can use the dictionary's .values() method to get a list-like view of its values and then iterate over them.

```
person = {
    "name": "Alice",
    "age": 30,
    "occupation": "Engineer"
}

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

Iterating over Dict

To iterate over a dictionary in a for loop, you can use the dictionary's keys, values, or items (key-value pairs) as the iteration source. Here are three different ways to write a for loop to iterate over a dictionary

3. Iterating Over Items (Key-Value Pairs): You can use the dictionary's .items() method to get a list-like view of its key-value pairs and then iterate over them. Each iteration will provide both the key and the value.

```
person = {
    "name": "Alice",
    "age": 30,
    "occupation": "Engineer"
}

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

Unpacking

You can unpack sequences like lists and tuples into individual variables.

```
numbers = [1, 2, 3]
a, b, c = numbers
```

Function unpacking

You can also use input unpacking with functions that return multiple values, which are often packed into tuples or lists

```python
def get_name_and_age():
    return "Alice", 30

name, age = get_name_and_age()
```

split Method

The split method splits a string into a list of substrings based on a specified delimiter. By default, it splits the string using whitespace characters as the delimiter.

```
sentence = "Hello, how are you today?"
words = sentence.split()
# Split using default delimiter
(whitespace)
print(words)
# Output: ['Hello,', 'how', 'are', 'you',
'today?']

csv_data = "Alice,30,Engineer"
fields = csv_data.split(',')
# Split using comma as delimiter
print(fields)
# Output: ['Alice', '30', 'Engineer']
```

split Method in input process

You can use the split method in Python to process user input and split it into separate values. Here's an example of how you might use the split method to receive multiple input values from a user

```
# Get input from the user
input_string = input("Enter three numbers
separated by spaces: ")

# Split the input string into a list of
substrings
numbers_as_strings = input_string.split()

# Convert the list of strings to a list
of integers
numbers = [int(number) for number in
numbers_as_strings]

# Print the list of numbers
print("You entered:", numbers)
```

join Method

The join method is used to concatenate a list of strings into a single string using a specified separator.

```python
words = ['Hello', 'how', 'are', 'you',
'today?']
sentence = ' '.join(words)
# Join with a space separator
print(sentence)
# Output: 'Hello how are you today?'

fields = ['Alice', '30', 'Engineer']
csv_data = ','.join(fields)
# Join with a comma separator
print(csv_data)
# Output: 'Alice,30,Engineer'
```

Enumerate

The enumerate function in Python is used to iterate over a sequence (such as a list, tuple, or string) while keeping track of both the index and the value of each element. It returns pairs of index-value pairs, which are often useful when you want to know the position of each item while iterating.

```
for index, value in enumerate(sequence):
    # Code to be executed for each
index-value pair

# Example
fruits = ["apple", "banana", "orange"]

for index, fruit in enumerate(fruits):
    print("Index:", index, "Fruit:", fruit)
```

Enumerate

The enumerate function in Python is used to iterate over a sequence (such as a list, tuple, or string) while keeping track of both the index and the value of each element. It returns pairs of index-value pairs, which are often useful when you want to know the position of each item while iterating.

```
for index, value in enumerate(sequence):
    # Code to be executed for each
index-value pair

# Example
fruits = ["apple", "banana", "orange"]

for index, fruit in enumerate(fruits):
    print("Index:", index, "Fruit:", fruit)
```

Zip

The zip function in Python is used to combine multiple iterables (such as lists, tuples, or strings) element-wise. It pairs up elements from each iterable and returns an iterator that produces tuples containing these paired elements.

```
for item1, item2, ... in zip(iterable1, iterable2, ...):
    # Code to be executed for each element combination

# Example
names = ["Alice", "Bob", "Charlie"]
scores = [85, 92, 78]

for name, score in zip(names, scores):
    print("Name:", name, "Score:", score)
```

Python built-in function

Python has a rich set of built-in functions like len(), max(), min(), sum(), etc. Using them can simplify your code and make it more readable.

len Function

The len function returns the number of items in an iterable (like a list, tuple, or string).

```
numbers = [1, 2, 3, 4, 5]
length = len(numbers)
print(length)  # Output: 5
```

max and min Functions

The max function returns the maximum value from an iterable, and the min function returns the minimum value.

```
numbers = [12, 34, 5, 67, 8]
max_value = max(numbers)
min_value = min(numbers)
print("Max:", max_value)
# Output: Max: 67
print("Min:", min_value)
# Output: Min: 5
```

sum Function

The sum function calculates the sum of all items in an iterable (like numbers in a list).

```
numbers = [1, 2, 3, 4, 5]
total = sum(numbers)
print(total)  # Output: 15
```

sorted Function

The sorted function returns a new sorted list from the items in an iterable. You can also use the reverse parameter of sorted to sort in descending order.

```
numbers = [5, 2, 8, 1, 7]
sorted_numbers = sorted(numbers)
print(sorted_numbers)
# Output: [1, 2, 5, 7, 8]
```

abs Function

The abs function returns the absolute value of a number.

```
num = -10
abs_num = abs(num)
print(abs_num)  # Output: 10
```

round Function

The round function rounds a floating-point number to the nearest integer.

```
num = 3.7
rounded_num = round(num)
print(rounded_num)  # Output: 4
```

That’s all for today!