Python

1) Iterators:

In Python, an iterator is an object that allows you to go through a collection of items one by one without needing to use loops or indexes directly.

Methods for Accessing elements in List:

Using print statements:

```
List=[1,2,3,4,5]
print(List)
```

```
[1, 2, 3, 4, 5]
```

Using indexing:

```
List=[1,2,3,4,5]
print(List[0])
print(List[1])
print(List[2])
print(List[3])
print(List[4])
```

```
1
2
3
4
5
```

Using For Loop

```
List=[1,2,3,4,5]
for i in List:
    print(i)
```

```
1
2
3
4
5
```

Using While Loop:

```
List=[1,2,3,4,5]
i=0
while i<len(List):
    print(List[i])
    i=i+1
```

```
1
2
3
4
5
```

Using while Loop for set:

We

```
set={1,2,3,4,5}
i=0
while i<len(set):
    print(set[i])
    i=i+1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[7], line 4
      2 i=0
      3 while i<len(set):
----> 4     print(set[i])
      5     i=i+1

TypeError: 'set' object is not subscriptable
```

We can only use for loop to iterate through other datatypes like list, tuple, set etc. However while loops cannot access elements in sets. To address this limitation we can utilize ***iterators.***

Why use iterator?

Works with any collection: Iterators work with lists, sets, tuples and more.

Save memory: Iterators only process one item at a time, which is more efficient.

Creating and using an iterator:

Step 1: Create an Iterator

Use the 'iter()' function which converts iterable objects into iterators, allowing us to access their elements one at a time

```
Lis=[1,2,3,4]
ite=iter(List)
print(ite)

#creates list
#creates iterator object ite using the iter function and prints
#ouput shows the iterator object with its memory location
```

```
<list_iterator object at 0x2c8bb88>
```

Step 2: Get items with 'next()'

Using the 'next()' function to get items one at a time

```
List=[1,2,3,4,"Python"]
ite=iter(List)
print(next(ite)) #Retrieve the next element 1
print(next(ite)) #2
print(next(ite)) #3
print(next(ite)) #4
print(next(ite)) #"Python"
```

```
1
2
3
4
Python
```

If you try to get more items than the list, you will get an error

```
print(next(ite)) # This will cause a StopIteration error because there are no elements to retrieve
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[31], line 1
----> 1 print(next(ite))

StopIteration:
```

Using iterator in For Loop:

You can use iterator in for loop

```
List=[1,2,3,4,"Python"]
ite=iter(List)
for i in List:
    print(i)
```

```
1
2
3
4
Python
```

Using iterator in while loop:

```
List=[1,2,3,4]
ite=iter(List)
while True:
    try:
        n=next(ite)
        print(n)
    except StopIteration:
        break
```

```
1
2
3
4
```

Defining custom iterators:

Now let's see how we can define our own custom iterators in Python. To create a custom iterator, we need to define a class that implements the iterator protocol:

1. ' iter()' method returns the next element in the iteration object itself and is called when the iterator is initialized.
2. 'next()' method returns the next element in the iteration sequence and is called each tie we request the next element

Create a custom iterator class named ‘Ten’ that generates number from 1 to 10

```python
#Define a custom iterator class named Ten
class Ten:
    def __init__(self):
        self.num=1
# Define the __iter__() method to return the iterator object itself
    def __iter__(self):
        return self
# Define the __next__() method to generate the next value
    def __next__(self):
        if self.num <= 10:
            value =self.num
            self.num +=1
            return value
        else:
            raise StopIteration
#Create an instance of the Ten Class
obj=Ten()

#Iterate over the instance of the ten class and print each value
for i in obj:
    print(i)
```

```
1
2
3
4
5
6
7
8
9
10
```

Key points:

- Iterator: Object providing access to items one at a time
- iter(): Converts a collection into an iterator
- next():Retrieves the next item from the iterator
- StopIteration: Exception raised when no more items available
- Customized iterator: Allows us to iterate over data in a customized manner tailored to our needs

2) Generators:

In Python, generators is a kind of function that returns an object called generator object which can return a series of values rather than a single value

Normal Function:

```
def get_numbers():
    return[1,2,3,4,5]
numbers=get_numbers()
for n in numbers:
    print(n)
```

```
1
2
3
4
5
```

Explanation:

- The function 'get_numbers' returns a list containing all numbers at once
- All numbers are stored in memory at the same time

Generator function:

Creating a generator function is similar to creating a function. We use he keyword def and a yield statement to create a generator

```
def get_numbers():
    yield 1
    yield 2
    yield 3
    yield 4
    yield 5
numbers=get_numbers()
for n in numbers:
    print(n)
```

```
1
2
3
4
5
```

Explanation:

- The function 'get_numbers' yields each number one at a time.
- Only one number is in memory at a time, making it more memory efficient

Example of generator function that yields square numbers:

```
def square_generator(n):
    for i in range(n):
        yield i**2
sq= square_generator(10)
for i in sq:
    print(i)
```

```
0
1
4
9
16
25
36
49
64
81
```

In this example, 'square_generator' is a generator function that yields the square of each number from 0 to n-1. When we iterate over the generator object 'sq', it generates and yield one square number at a time.

Generator Expressions:

Generator expressions are a compact way of creating iterators. They are similar to list comprehension, but instead of creating a list they generate items one at a time.

Simple Generator Expression:

A simple generator expression goes through each item in an iterable and applies an expression to it

Example

We have a list of numbers ,and we want to generate a sequence of their squares:

```
numbers=[1,2,3,4,5]

#Generator expression to generates squares
sq=(num ** 2 for num in numbers)

#Printing result
for square in sq:
    print(square)
```

```
1
4
9
16
25
```

Conditional Generator Expression:

A conditional generator expression adds a condition to filter items before applying the expression. It generates based on whether they meet the condition

Example:

Generate a sequence of squares for only the even numbers in the list:

```
numbers=[1,2,3,4,5]

#Generator expression to generates squares of even numbers
evensq=(num**2 for num in numbers if num % 2 == 0)

#Printing Reult
for square in evensq:
    print(square)
```

```
4
16
```

3) Closure:

To understand the concept of closures, we need to understand some basic concepts

Nested Functions:

A nested function is simply a function defined inside another function

```
def outer_fun():#outer function scope
    def inner_fun():# inner function scope
        print("This is inner function")

    print("This is outer function")
    inner_fun()
# call the outer functon
outer_fun()
```

```
This is outer function
This is inner function
```

Explanation:

- Inner_fun is defined inside the outer_fun
- When outer_fun is called,it prints *"This is outer function"* and then calls inner_fun.
- Inner_fun prints *"This is inner function"* when called

Local Variables:

A variable defined inside a function and only accessible within that function

```
def my_fun():
    local_var="I am local"
    print(local_var)
my_fun()
```

```
I am local
```

Non-Local Variables:

A variable that is defined outside of a function, but it is used inside that function

```
def outer_fun():
    non_local_var="I am non-local"
    def inner_fun():
        print(non_local_var)#Accessing the non_local variable
    inner_fun()
outer_fun()
```

```
I am non-local
```

Closure:

A closure is a function that remembers the values from its enclosing lexical scope even when the scope has finished executing

```
def outer_fun(x):
    def inner_fun(y):
        return x+y
    return inner_fun
addfive=outer_fun(5)
print(addfive(3))
```

```
8
```

Explanation:

- In the example, inner_fun is defined inside outer_fun.
- Whenever outer_fun(5) is called, it returns inner_fun with x set to 5, creating a closure.
- add_five now holds the closure, allowing it to remember the value o x(which is 5) even after outer_fun has finished executing.
- When add_five(3) is called, it adds 5(captured from the closure) and 3, resulting in 8.

Conditions for Closures in Python:

1. **Nested Functions**:
   Closure require nested functions, meaning a function defined inside another function
2. **Access to Outer variable**:
   The nested function refer to variables defined in its outer scope, such as variables from the enclosing function.
3. **Return of Nested function**:
   The outer function must return the nested function for the closure to be created and maintained.

4) Decorators:

Definition: A Python decorator is a function that accepts another function as an argument and returns a new function.

Purpose: Decorators allow you to enhance or modify the behaviour of the original function without altering its actual code.

Functionality: By "wrapping" the original function, decorators enable you to run additional code before or after the original function is executed.

Non-Permanent Changes: They adjust the function's behaviour temporarily, without making permanent changes to the function itself.

Usage: Decorators are applied using the @decorator_name syntax placed above the function definition

Why Do We Need Python Decorators?

We use decorators when we want to add the same behaviour to multiple functions or change the inputs of existing functions. They help us apply these changes in one place, making the code easier to manage and read.

To create Python decorators, follow these steps:

1. Define an outer function that accepts another function as its argument.
2. Inside this outer function, create an inner function that includes the additional features you need.
3. Use the @ symbol followed by the name of the decorator function to apply it to the target function.

Basic Function

First, let's define a basic function:

```
def hello():
    print("hi")
print(hello())
```

```
hi
None
```

This will print "hi" and then None because the hello function itself returns None.

Adding Functionality with Decorators

Now, we'll use a decorator to add additional functionality to this function

```
def upper_decor(fun):
    # This is a higher-order function because it takes a function as an argument
    def wrapper():
        result = fun()  # Call the input function
        return result.upper()  # Modify the result
    return wrapper

def hello():
    return "hi"

# Manually decorate the function
f = upper_decor(hello)
print(f())
```

```
HI
```

Explanation:

- upper_decor(fun) defines a function wrapper that converts fun's output to uppercase and returns wrapper.
- hello() simply returns "hi".
- f = upper_decor(hello) applies upper_decor to hello, creating a new function f that will return the uppercase version of hello's output.
- print(f()) calls the decorated function f, which outputs "HI" by converting "hi" to uppercase

Create a decorator function that takes another function as an argument

```
def double_decor(fun):
     def wrapper(n):
            any_list=fun(n)
            return[item*3 for item in any_list]
     return wrapper

def reverse_list(l):
      return l[::-1]

def upper_list(l):
      return[item.upper() for item in l]

print(reverse_list([1,2,3,4]))
print(upper_list(['a', 'b', 'c']))
```

```
[4, 3, 2, 1]
['A', 'B', 'C']
```

Explanation:

- The double_decor function defines a decorator that takes a function as input.
- Inside the decorator, there's a wrapper function (wrapper) that modifies the behaviour of the original function.
- The wrapper function triples each element of the list returned by the original function.
- reverse_list: Reverses the input list.
- upper_list: Converts input strings to uppercase

5) property:

- The property() function in Python is utilized to establish class properties, primarily used within Python classes to define attributes with custom getter, setter, and deleter methods.
- It's a fundamental tool in Python's Object-oriented paradigm, accepting getter, setter, and deleter methods as arguments to define properties within classes.
- Python natively incorporates the property() function for declaring class properties.
- Alongside property(), Python offers the @property decorator, enhancing the ease of implementing getter and setter methods in Object-Oriented Programming

Python getters and setters:

In object-oriented programming, getters and setters are used to keep data safe and private within a class. Getters help to fetch the value of private attributes, while setters allow changing or updating these values. This way, we prevent other classes from accidentally changing our data.

Python property:

Functionality: In Python classes, the property() function defines properties, which are managed by the Properties class to handle class attributes.

Usage: It's a built-in function that creates and returns a property object, serving as an interface for instance attributes in Python

Syntax:

property(fget, fset, fdel, doc)

Parameters:

fget (optional): Function for retrieving an attribute value. Default is None.

fset (optional): Function for setting an attribute value. Default is None.

fdel (optional): Function for deleting an attribute value. Default is None.

doc (optional): String containing documentation (docstring) for the attribute. Default is None.

Return Value:

Returns property attributes based on the specified getter, setter, and deleter functions.

If no arguments are provided, property() returns a base property attribute with no getter, setter, or deleter.

If doc is not provided, property() uses the docstring from the getter function.

How python property works?

Consider a basic syntax example where we create a new class attribute called 'attr' and define its three parameters as properties:

attr = property(getter, setter, deleter)

Now, if 'obj' represents an object instance of a class, accessing obj.attr in Python triggers the getter function.

When assigning a value to obj.attr like obj.attr = value, Python executes the setter function and passes the value as an argument.

Similarly, when del obj.attr is used, Python invokes the deleter method.

The argument provided as doc is utilized by Python as the attribute's docstring.

Example:

```
class Student:
    def __init__(self):
        self._grade = None

    def get_grade(self):
        return self._grade

    def set_grade(self, value):
        if value < 0 or value > 100:
            raise ValueError("Grade must be between 0 and 100")
        self._grade = value

    def del_grade(self):
        del self._grade

    grade = property(get_grade, set_grade, del_grade, "Grade of the student")


s = Student()
print("Getting grade...")
print(s.grade)  # Calls get_grade()
print("Setting grade to: 85")
s.grade = 85     # Calls set_grade(85)
print("Deleting grade...")
del s.grade     # Deletes the grade attribute
```

```
Getting grade...
None
Setting grade to: 85
Deleting grade...
```

In this example:

- The Student class has a single property attribute called grade.
- It defines three methods: get_grade, set_grade, and del_grade.
- get_grade retrieves the grade of the student, set_grade sets the grade within the valid range of 0 to 100, and del_grade deletes the grade attribute.
- The grade property is created using the property() function.
- When accessing s.grade, it internally calls the getter method get_grade().

- Assigning a new value to s.grade invokes the setter method set_grade() to ensure the grade is within the valid range.
- Attempting to delete s.grade invokes the deleter method del_grade().

Regular Expressions:

Why regular expressions are used?

consider another scenario where regular expressions can be useful:

Suppose you have a large text document containing various email addresses, and you need to extract all the email addresses from it

text = """

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Email addresses: john.doe@example.com, jane.doe@example.org,

johndoe1234@email.co.uk, test.email@email.domain

"""

Regular Expressions can be used in this case to recognize the patterns and extract the required information easily.

What are regular expressions?

Regular Expressions are powerful tools used to identify specific patterns within text strings. They enable tasks such as validating data, searching for patterns, and performing operations like find-and-replace and formatting within text.

Basic regular expressions:

Finding a word in string

```
import re

# Sample text containing the word "sword"
text = "The knight wielded a mighty sword in battle."

# Regular expression pattern to match the word "sword"
pattern = r'\bsword\b'

# Find all occurrences of the word "sword" using the regular expression pattern
matches = re.findall(pattern, text)

# Print the matches found
print("Occurrences of 'sword':", matches)
```

```
Occurrences of 'sword': ['sword']
```

Replacing a string:

```
import re

Fruits = "apple orange grapes"

regex = re.compile(r"\borange\b")

Fruits = regex.sub("watermelon", Fruits)

print(Fruits)
```

```
apple watermelon grapes
```

consider an example where we have a text document containing various phone numbers in different formats, and we want to extract all the phone numbers from it using regular expressions.

```python
import re

# Sample text containing phone numbers in different formats
text = """
Phone numbers:
- (123) 456-7890
- 987-654-3210
- +1 (555) 123-4567
- 555.678.9012
- 1234567890
"""

# Regular expression pattern to match phone numbers
phone_pattern = r'\b(?:\+\d{1,2}\s*)?(?:\(\d{3}\)|\d{3})[-. ]?\d{3}[-. ]?\d{4}\b'

# Find all phone numbers using the regular expression pattern
phone_numbers = re.findall(phone_pattern, text)

# Print the extracted phone numbers
print("Extracted phone numbers:")
for phone in phone_numbers:
    print(phone)
```

```
Extracted phone numbers:
987-654-3210
555.678.9012
1234567890
```

Explanation:

- We have a sample text document containing phone numbers in different formats.
- We define a regular expression pattern phone_pattern to match phone numbers. This pattern accounts for various formats including those with area codes, country codes, and different separators like hyphens, dots, and spaces.
- We use the re.findall() function to find all occurrences of phone numbers in the text that match the specified pattern.
- Finally, we print out all the extracted phone numbers.

Python Date and Time:

Python datetime Module

This module provides classes for manipulating dates and times in both simple and complex ways. It allows you to work with dates, times, and combined date-time representations. You can create datetime objects to represent specific dates and times, perform arithmetic operations on them, and format them as strings for display or storage.

Formatting a datetime object as a string:

```
import datetime
current_datetime = datetime.datetime.now()
formatted_date = current_datetime.strftime("%Y-%m-%d")
print(formatted_date)
```

```
2024-05-26
```

Formatting a datetime object with time:

```
import datetime
current_datetime = datetime.datetime.now()
formatted_datetime = current_datetime.strftime("%Y-%m-%d %H:%M:%S")
print(formatted_datetime)
```

```
2024-05-26 01:05:24
```

Python datetime.strftime()

The strftime() method is used to convert a datetime object into a string representation based on a specified format. You can specify the format using various directives, which are placeholders for different components of the datetime object, such as year, month, day, hour, minute, and second.

Formatting a datetime object as a string:

```
import datetime
current_datetime = datetime.datetime.now()
formatted_date = current_datetime.strftime("%Y-%m-%d")
print(formatted_date)
```

```
2024-05-26
```

Formatting a datetime object with time:

```
import datetime
current_datetime = datetime.datetime.now()
formatted_datetime = current_datetime.strftime("%Y-%m-%d %H:%M:%S")
print(formatted_datetime)
```

```
2024-05-26 01:10:02
```

Python datetime.strptime()

The strptime() method is the inverse of strftime(). It converts a string representation of a date and time into a datetime object. You need to specify the format of the input string using directives that match the components of the string representation.

Converting a string to a datetime object:

```
import datetime
date_string = "2024-05-26"
converted_date = datetime.datetime.strptime(date_string, "%Y-%m-%d")
print(converted_date)
```

```
2024-05-26 00:00:00
```

Converting a string with time to a datetime object:

```
import datetime
datetime_string = "2024-05-26 12:30:00"
converted_datetime = datetime.datetime.strptime(datetime_string, "%Y-%m-%d %H:%M:%S")
print(converted_datetime)
```

```
2024-05-26 12:30:00
```

Current date & time

This refers to obtaining the current system date and time. You can use the datetime.now() function to get the current date and time as a datetime object. Alternatively, you can use date.today() to get just the current date without the time component.

Getting the current date and time:

```
import datetime
current_datetime = datetime.datetime.now()
print(current_datetime)
```

```
2024-05-26 01:16:57.305000
```

Getting the current date only:

```
import datetime
current_date = datetime.date.today()
print(current_date)
```

```
2024-05-26
```

Get current time

This involves extracting the current time from the system clock. You can use datetime.now().time() to get the current time as a time object, which represents only the time portion without the date. Alternatively, you can access specific components like the hour using datetime.now().hour.

Getting the current time:

```
import datetime
current_time = datetime.datetime.now().time()
print(current_time)
```

```
01:19:32.209000
```

Getting the current hour:

```
import datetime
current_hour = datetime.datetime.now().hour
print(current_hour)
```

```
1
```

Timestamp to datetime

Timestamps are often used to represent a point in time as the number of seconds since a particular reference point (epoch). The fromtimestamp() method converts a Unix timestamp into a datetime object. You can also use utcfromtimestamp() to obtain a UTC datetime object from the timestamp.

Converting timestamp to datetime:

```
import datetime
timestamp = 1622014845
datetime_object = datetime.datetime.fromtimestamp(timestamp)
print(datetime_object)
```

```
2021-05-26 13:10:45
```

Converting timestamp to UTC datetime:

```
import datetime
timestamp = 1622014845
utc_datetime = datetime.datetime.utcfromtimestamp(timestamp)
print(utc_datetime)
```

```
2021-05-26 07:40:45
```

Python time Module

The time module provides various functions for working with time-related operations. It includes functions to obtain the current time, measure time intervals, and format time strings. You can use time.time() to get the current time in seconds since the epoch.

Getting the time elapsed using time module:

```python
import time
start_time = time.time()
# Some code or operation
end_time = time.time()
elapsed_time = end_time - start_time
print(f"Elapsed time: {elapsed_time} seconds")
```

```
Elapsed time: 0.0 seconds
```

Formatting time using time module:

```python
import time
current_time = time.strftime("%H:%M:%S")
print(current_time)
```

```
01:25:34
```

Python time sleep()

The sleep() function suspends the execution of the current thread for a specified number of seconds. It's useful for delaying the execution of code or creating time intervals between operations. You specify the duration of the pause in seconds as an argument to the function.

Pausing execution for 5 seconds:

```python
import time
print("Start")
time.sleep(5)
print("End")
```

```
Start
End
```

Pausing execution for 1 minute:

```python
import time
print("Start")
time.sleep(60)
print("End")
```

```
Start
End
```