Day 6

Agenda
Functional Programming: Immutability, Closures and Decorators, generators, Coroutines, iterators, Declarative programming,

Immutability

Immutability refers to the idea that data should not be modified after it is created. Instead of changing existing data, new data is created with the necessary modifications. This concept is fundamental in functional programming for several reasons:

Predictability: Since data doesn't change, functions will always produce the same output given the same input.
Concurrency: Immutability eliminates issues related to shared mutable state in concurrent programming.
Simplified Debugging: With immutable data, the state of the program is easier to understand and trace.

Example;

```
# Immutable tuple
coordinates = (10, 20)
# Any attempt to modify will create a new tuple
new_coordinates = coordinates + (30,)

print(coordinates)       # Output: (10, 20)
print(new_coordinates)   # Output: (10, 20, 30)
```

```
(10, 20)
(10, 20, 30)
```

Closures

- In nested functions the **outer function is called the enclosing function** and the **inner function is called the closure function.**

- A Closure is a function object that remembers variables in enclosing scopes even if they are not present in memory.
- It means they can access these values even when the function is invoked outside their scope.

Example:

```
def outerFunction(text):
    text = text
    def innerFunction():
        print(text)
    return innerFunction
myFunction = outerFunction('Hey!')
myFunction()
```

```
Hey!
```

- As observed from the above code, closures help to invoke functions outside their scope.
- The function innerFunction has its scope only inside the outerFunction. But with the use of closures, we can easily extend its scope to invoke a function outside its scope.

When and why to use Closures

- As **closures** are used as **call back functions**(A callback function is a function that is passed as an argument to another function and is intended to be called at a later time or in response to some event),
- Being call back functions, they provide some sort of **data hiding**. This helps us to reduce the use of global variables.
- When we have few functions in our code, closures prove to be an efficient way. But if we need to have many functions, then go for class (OOP).
- Python **Decorators make extensive use of closures** as well.

Characteristics of Closures

- Three characteristics of a Python closure are:
    a. it is a nested function

b. it has access to a free variable in outer scope
c. it is returned from the enclosing function

- A free variable is a variable that is not bound in the local scope
- Python closures help avoid the usage of global values and provide some form of data hiding.

Example

```
def outerfunc(x):
    def innerfunc():
        print(x)
    return innerfunc  #Return the object(name) instead of calling the function
myfunc=outerfunc(7)
myfunc()
```

```
7
```

- The point to note here is that instead of calling innerfunc() here, we returned it (the object).
- Once we've defined outerfunc(), we call it with the argument 7 and store it in variable myfunc().
- When we call myfunc next, how does it remember that 'x' is 7?
- A Python closure is when some data gets attached to the code.
- So, this value is remembered even when the variable goes out of scope, or the function is removed from the namespace.

One more example:

```python
def make_counter():
    count = 0  # This is the enclosed variable

    def counter():
        nonlocal count  # This allows the nested function to modify the enclosing variable
        count += 1
        return count

    return counter

# Create two independent counters
counter1 = make_counter()
counter2 = make_counter()

# Using the first counter
print(counter1())  # Output: 1
print(counter1())  # Output: 2
print(counter1())  # Output: 3

# Using the second counter
print(counter2())  # Output: 1
print(counter2())  # Output: 2

# The first counter is independent of the second counter
print(counter1())  # Output: 4
```

```
1
2
3
1
2
4
```

Decorators

- A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure.
- The **outer function is called the decorator**, which takes the original function as an argument and returns a modified version of it.
- Decorators are usually called before the definition of a function.

Note: Decorator function takes a function as an argument, therefore, define a function and pass

it to decorator.

Example

```
def decor(func):
    def inner():
        print("------------------")
        func()
        print("------------------")
    return inner

def msg():
    print("Python Programming")

msg = decor(msg)
msg()
```

```
------------------
Python Programming
------------------
```

@ symbol with decorator

- The above program can be written in a more elegant way by using @ symbol.
- Instead of assigning the function call to a variable, Python provides a much more elegant way to achieve this functionality using the @ symbol.

```
def decor(func):
    def inner():
        print("------------------")
        func()
        print("------------------")
    return inner
@decor
def msg():
    print("Python Programming")

msg()
```

```
------------------
Python Programming
------------------
```

Example

The output of the following program will depend on current time. If the current time is day time then it will give the output Playing music.

```
from datetime import datetime

def not_duering_night(func):
    def inner():
        if 7 <= datetime.now().hour < 22:
            func()
        else:
            print("Sorry! Unable to play music in night")
    return inner

@not_duering_night
def music():
    print("Playing music")

music()
```

```
Sorry! Unable to play music in night
```

Decorating Functions with Parameters

- Outer function will take function name as an arguments and Inner function will take parameters.
- If we use ***args** and ****kwargs** in the inner wrapper function. Then it will accept an arbitrary number of positional and keyword arguments.

```
def do_twice(func):
    def wrapper_do_twice(*args, **kargs):
        func(*args, **kargs)
        func(*args, **kargs)
    return wrapper_do_twice

@do_twice
def message(name):
    print(f"Hello {name}")


message("Mohit")
```

```
Hello Mohit
Hello Mohit
```

Returning value from Decorated Function

- If you want to return the value from the decorated function, then you need to **make sure the wrapper function returns the return value** of the decorated function.

Example

```
def do_twice(func):
    def wrapper_do_twice(*args, **kargs):
        func(*args, **kargs)
        return func(*args, **kargs)
    return wrapper_do_twice

@do_twice
def message(name):
    return f"Hello {name}"

text = message("Mohit")
print(text)
```

```
Hello Mohit
```

Apply Multiple Decorators to a Function

Example

- For num() function we are applying 2 decorator functions. Firstly the inner decorator will work and then the outer decorator.

```
def decor1(func):
    def inner():
        x = func()
        return x * x
    return inner

def decor(func):
    def inner():
        x = func()
        return 2 * x
    return inner


@decor1
@decor
def num():
    return 10

print(num())     #first decor and then decor1
```

```
400
```

Generators

- A Python **generator** is a kind of an **iterable**, like a Python **list or tuple**.
- It generates a sequence of values that we can iterate on.
- You can use it to iterate on a for-loop in python, but you **can't index it**.
- Simply, a generator is a function that returns an object (iterator) which we can iterate over.

Create Generator Function

- To create a python generator, we use the **yield** statement, inside a function, instead of the return statement.
- If a function contains **at least one yield statement**, it becomes a generator function.

- The difference between yield and return statement is that a return statement terminates a function entirely, yield statement pauses the function saving all its states and later continues from there on successive calls.

Example Print numbers from 1 to 10 using Generator.

```
def generate_numbers():
    for num in range(1, 11):
        yield num

# Create the generator
numbers_generator = generate_numbers()
print(type(numbers_generator))

# Print numbers from the generator
for number in numbers_generator:
    print(number)
```

```
<class 'generator'>
1
2
3
4
5
6
7
8
9
10
```

Example: Print all prime numbers from 5 to 50 using a generator

```
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            return False
    return True

def generate_primes(start, end):
    for num in range(start, end + 1):
        if is_prime(num):
            yield num

# Create the generator
primes_generator = generate_primes(5, 50)

# Print prime numbers from the generator
for prime in primes_generator:
    print(prime, end = ', ')
```

```
5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
```

Example: print all the characters of the passed string in reverse order

```
#print all the characters of the passed string in reverse order
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1, -1, -1):
        yield my_str[i]

for char in rev_str("hello"):
    print(char)
```

```
o
l
l
e
h
```

More than one yield in Generator

Generators may contain more than one Python yield statement. This is comparable to how a Python generator function may contain more than one return statement.

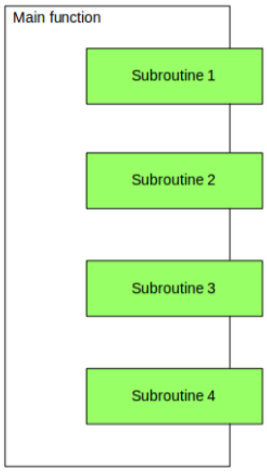
```
def my_gen(x):
    while(x > 0):
        if x%2==0:
            yield 'Even'
        else:
            yield 'Odd'
        x -= 1

for i in my_gen(7):
    print(i)
```

```
Odd
Even
Odd
Even
Odd
Even
Odd
```

Coroutine

We all are familiar with function which is also known as a subroutine, procedure, sub-process, etc.

Functions in Python are called by the main function which is responsible for coordinating the use of these subroutines. Subroutines have a single entry point.



- Coroutines are generalizations of functions. They are used for cooperative multitasking where a process voluntarily yield (give away) control periodically or when idle in order to enable multiple applications to be run simultaneously. The difference between coroutine and function is :
- coroutines are similar to generators but with few extra methods and slight changes in how we use yield statements.
- **Value to yield comes from send() function.**
- Generators produce data for iteration while coroutines can also consume data.
- yield can also be used as an expression.
- When we call coroutine nothing happens, it runs only in response to the **next()** and **sends ()** method.

Syntax:

x= (yield)

Example:

```python
def print_name(prefix):
    print("Searching prefix:{}".format(prefix))
    while True:
        name = (yield)
        if prefix in name:
            print(name)

# calling coroutine, nothing will happen
corou = print_name("Dear")

# This will start execution of coroutine and
# Prints first line "Searching prefix..."
# and advance execution to the first yield expression
corou.__next__()

# sending inputs
corou.send("Atul")
corou.send("Dear Atul")
```

```
Searching prefix:Dear
Dear Atul
```

Closing a Coroutine

- Coroutine might run indefinitely, to close coroutine **close()** method is used.

Example

```
def print_name(prefix):
    print("Searching prefix:{}".format(prefix))
    try :
        while True:
                name = (yield)
                if prefix in name:
                    print(name)
    except GeneratorExit:
            print("Closing coroutine!!")

corou = print_name("Dear")
corou.__next__()
corou.send("Atul")
corou.send("Dear Atul")
corou.close()
```

```
Searching prefix:Dear
Dear Atul
Closing coroutine!!
```

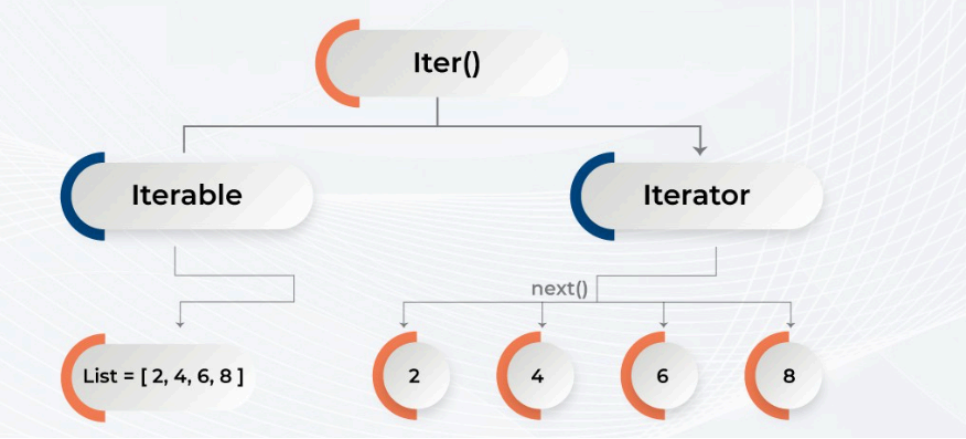
Iterator

- An iterator is an **object that contains a countable number of values**.
- Iterator in python is an object that is used to iterate over **iterable objects like lists, tuples, dicts, and sets.**
- The iterator object is initialized using the iter() method. It uses the next() method for iteration.
- Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods __iter__() and __next__().

Iterable vs Iterator

- Lists, tuples, dictionaries, and sets all are iterable objects.
- All iterable objects have an __iter__() method which is used to get an iterator.
- Iterator along with __iter__() also contains __next__().

- Not all iterables are iterators, but all iterators are iterables.



Example: Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.).

```
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self
    def __next__(self):
        x = self.a
        self.a += 1
        return x

ob = MyNumbers()
myiter = iter(ob)
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

```
1
2
3
4
5
```

Declarative programming

- Declarative programming is a programming paradigm that expresses the logic of computation without describing its control flow.
- In contrast to imperative programming, which requires the developer to write explicit instructions in code, declarative programming focuses on what needs to be done.
- Declarative programming is often associated with languages and tools that allow for concise and readable code.

Key Characteristics of Declarative Programming:

- **Focus on What, Not How:** Specify what the program should accomplish rather than detailing the steps to achieve it.
- **Higher-Level Abstractions:** Use higher-level constructs and abstractions that hide implementation details.
- **Immutability and Side-Effect-Free Functions:** Prefer immutability and functions without side effects.

Difference between Iterative vs Declarative programming

Iterative programming	Declarative programming

```python
# Imperative style
numbers = [1, 2, 3, 4, 5]
squares = []
for n in numbers:
    squares.append(n ** 2)
print(squares)
# Output: [1, 4, 9, 16, 25]
```

```
[1, 4, 9, 16, 25]
```

```python
# Declarative using list comprehensions
numbers = [1, 2, 3, 4, 5]
squares = [n ** 2 for n in numbers]
print(squares)
# Output: [1, 4, 9, 16, 25]
```

```
[1, 4, 9, 16, 25]
```

Declarative programming uses List comprehension, map, filter, reduce etc..

Example: Declarative style using filter

```python
# Declarative style using filter
numbers = [1, 2, 3, 4, 5]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)
# Output: [2, 4]
```

```
[2, 4]
```

Example: Declarative style using reduce to find sum of list element.

```
from functools import reduce

# Declarative style using reduce
numbers = [1, 2, 3, 4, 5]
total = reduce(lambda x, y: x + y, numbers)
print(total)
# Output: 15
```

```
15
```