**Decorators** it used to modify the behaviour of a function or class. Decorators allow us to wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it

A decorator is a function that takes function as a argument an extend its functionality and return a modified function with extended functionality

**First Class Objects**

In python functions are First Class Objects, meaning that functions in python can be used or passed as arguments.

**Properties of first-class functions:**

* A function is an instance of the Object type.
* You can store the function in a variable.
* You can pass the function as a parameter to another function.
* You can return the function from a function.
* You can store them in data structures such as hash tables, lists, …

**Examples:**

**Treating the functions as objects.**

# Python program to illustrate functions

# can be treated as objects

def shout(text):

return text.upper()

print(shout('Hello'))

yell = shout

print(yell('Hello'))

**Passing the function as an argument**

# Python program to illustrate functions

# can be passed as arguments to other functions

def shout(text):

return text.upper()

def whisper(text):

return text.lower()

def greet(func):

# storing the function in a variable

greeting = func("""Hi, I am created by a function passed as an argument.""")

print (greeting)

greet(shout)

greet(whisper)

**Returning functions from another function**

# Python program to illustrate functions

# Functions can return another function

def create\_adder(x):

def adder(y):

return x+y

return adder

add\_15 = create\_adder(15)

print(add\_15(10))

**Decorators:**

It is used to modify the behaviour of the function or class.

In decorators, functions are taken as arguments into the another function and then called inside the wrapper function.

A screenshot of a computer

Description automatically generated

Example:

# defining a decorator

def hello\_decorator(func):

# inner1 is a Wrapper function in

# which the argument is called

# inner function can access the outer local

# functions like in this case "func"

def inner1():

print("Hello, this is before function execution")

# calling the actual function now

# inside the wrapper function.

func()

print("This is after function execution")

return inner1

# defining a function, to be called inside wrapper

def function\_to\_be\_used():

print("This is inside the function !!")

# passing 'function\_to\_be\_used' inside the

# decorator to control its behaviour

function\_to\_be\_used = hello\_decorator(function\_to\_be\_used)

# calling the function

function\_to\_be\_used()

The decorator function or Outer function must take a argument which is the function it decorates or modify

If a Decorator needs to take an argument, then the decorator function needs another level of nested function

Example:

A screen shot of a computer code

Description automatically generated

The inner decorator function (*wrapper*) should take the argument that are passed to the decorated function (*hello*).

In these cases you don't have any Also note that the *wrapper* function doesn't really need to return *func*. wrapper is essentially a replacement of func. That's what decorators do. They replace the function you decorate with the inner function of the decorator (by calling the outer function with the decorated function as an argument).

Example, if the function required arguments or it has the return values.

A screenshot of a computer program

Description automatically generated

**High order function:**

The function that takes one or more function as arguments and return a function as a result or do both.

Difference between High order function and first order function:

The high order function describes the mathematical order of the function that operates on the other function. While first order function is a computer science term for programming language entities that have no restriction on their use

Higher-order functions are fundamentally tied to the concept of first-class functions. Without Python's support for treating functions as first-class citizens, higher-order functions wouldn't be possible.

We can say - Without first-class functions, we cannot have higher-order functions, as they inherently rely on the ability to treat functions as first-class entities, operating on them by taking them as arguments or returning them as results.

**Lambda function:**

Also known as anonymous functions

Syntax: lambda arguments: expression

This functions can also passed or used in the High Order functions.

Examples:

A screenshot of a computer code

Description automatically generated

A screenshot of a computer program

Description automatically generated

**Closures:**

A closure is a feature in many programming languages, including Python, that allows a function to remember and access variables from an enclosing scope even after the outer function has finished executing.

In simpler terms, a closure is an inner function that has access to variables from its containing (or outer) function, even after that outer function has completed its execution.

Example:

A screenshot of a computer code

Description automatically generated

Explanation:

Now here, the outer\_scope function takes name and city as parameters. Inside outer\_scope, the inner\_scope function is defined to print a greeting message using name and city. Instead of calling inner\_scope, outer\_scope returns the inner\_scope function itself.

When outer\_scope is called with specific arguments, it creates and returns a closure that captures these arguments. For instance, greet\_priyanshu is a closure that remembers Dr Priyanshu and Jaipur, while greet\_sam remembers Sam and New York. When these closures are called, they produce the respective greeting messages.

Even though outer\_scope has finished executing in both cases, the inner\_scope functions (now greet\_priyanshu and greet\_sam) retain access to their respective name and city variables from their enclosing scopes, demonstrating closure behavior.

**Types of Decorators:**

**Function Decorators:**

The most common decorator which takes a function and return a new function.  
 A screen shot of a computer program

Description automatically generated

**Method Decorators:**

Used to decorate methods within a class. They often handle special cases, such as the *self* argument for instance methods.

A screen shot of a computer program

Description automatically generated

* **wrapper(self, \*args, \*\*kwargs):** The wrapper must accept self because it is a method of an instance. self is the instance of the class and \*args and \*\*kwargs allow for other arguments to be passed if needed.

**Class Decorators:**

[Class](https://www.geeksforgeeks.org/python-classes-and-objects/)decorators are used to modify or enhance the behavior of a

class. Like function decorators, class decorators are applied to the class definition. They work by taking the class as an argument and returning a modified version of the class.

A screen shot of a computer code

Description automatically generated

**Explanation:**

* **add\_class\_name(cls):**This decorator adds a new attribute, class\_name, to the class cls. The value of class\_name is set to the name of the class (cls.\_\_name\_\_).
* **@add\_class\_name:** This applies the add\_class\_name decorator to the Person class.
* **Result:**When the Person class is defined, the decorator automatically adds the class\_name attribute to it.
* **print(Person.class\_name):**Accessing the class\_name attribute that was added by the decorator prints the name of the class, Person.

**Common Built-in Decorators:**

Python provides several [built-in decorators](https://www.geeksforgeeks.org/top-python-built-in-decorators-that-optimize-python-code-significantly/) that are commonly used in class definitions. These decorators modify the behavior of methods and attributes in a class, making it easier to manage and use them effectively. The most frequently used built-in decorators are @staticmethod,

@classmethod, and  @property.

**@staticmethod:**

The [@staticmethod](https://www.geeksforgeeks.org/python-staticmethod/) decorator is used to define a method that doesn’t operate on an instance of the class (i.e., it doesn’t use self). Static methods are called on the class itself, not on an instance of the class.

Example:

A screenshot of a computer program

Description automatically generated

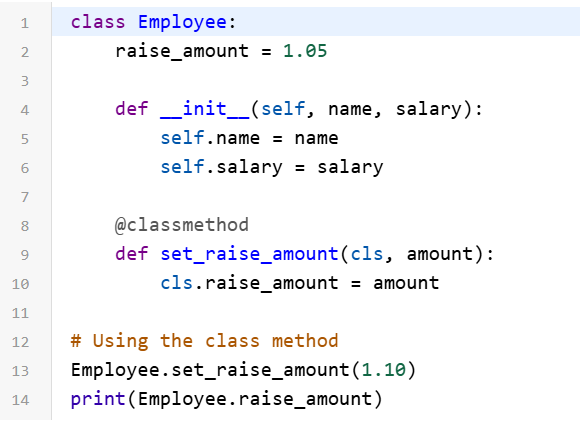
**Explanation:**

* add is a static method defined with the @staticmethod decorator.
* It can be called directly on the class MathOperations without creating an instance.

**@classmethod:**

The [@classmethod](https://www.geeksforgeeks.org/classmethod-in-python/) decorator is used to define a method that operates on the class itself (i.e., it uses cls). Class methods can access and modify class state that applies across all instances of the class.

Example:



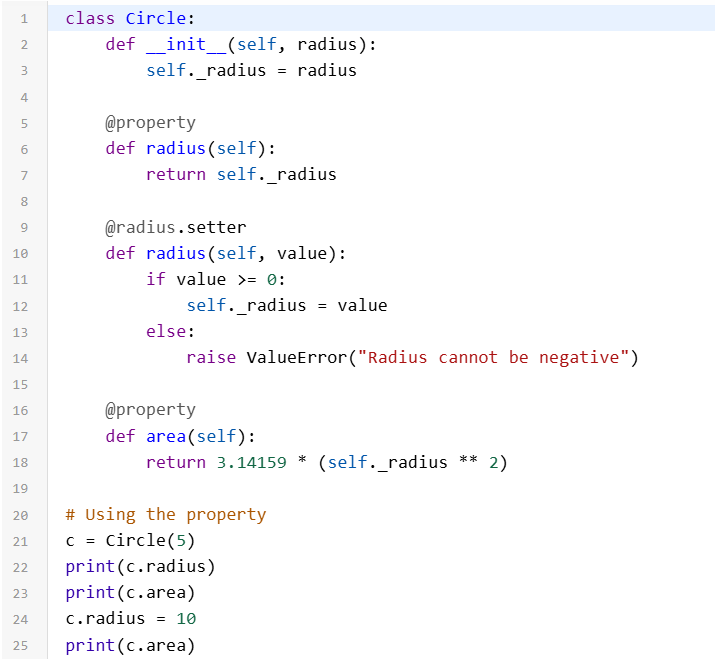
Explanation:

* set\_raise\_amount is a class method defined with the @classmethod decorator.
* It can modify the class variable raise\_amount for the class Employee and all its instances.

**@property:**

The [@property decorator](https://www.geeksforgeeks.org/python-property-decorator-property/) is used to define a method as a property, which allows you to access it like an attribute. This is useful for encapsulating the implementation of a method while still providing a simple interface.

Example:



**Explanation:**

* radius and area are properties defined with the @property decorator.
* The radius property also has a setter method to allow modification with validation.
* These properties provide a way to access and modify private attributes while maintaining encapsulation.

**Chaining Decorators:**

In simpler terms [chaining decorators](https://www.geeksforgeeks.org/chain-multiple-decorators-in-python/) means decorating a function with multiple decorators.

Syntax:

A white background with black text

Description automatically generated

Example:

A screenshot of a computer code

Description automatically generated

In above example the decorator are like **decor1(decor(num()))** Where *decor*  is first wrapped and then the whole wrapped function is wrapped with *decor1*