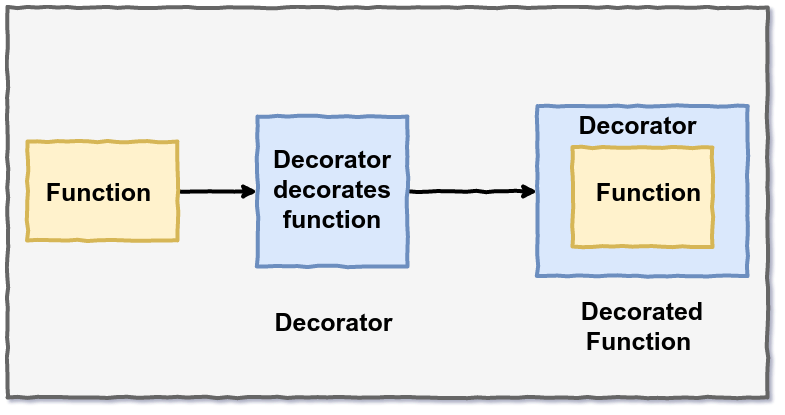
**DECORATORS**

Decorators are a very **powerful** and **useful** tool in Python since it allows programmers to **modify the behaviour of a function**. Decorators allow us to **wrap another function** in order to **extend the behaviour of the wrapped function, without permanently modifying** it.

As stated above the decorators are used to **modify the behaviour of function** or **class**. In Decorators, **functions are taken as the argument into another function** and then called inside the wrapper function.

A decorator **takes in a function**, **adds some functionality** and **returns** it.



**PREREQUISITE FOR DECORATORS :**

**Properties of Function:**

**Why Function is called as First class object(Interview Q.)**

 This means that a **function has all the rights as any other variable** in the language.

**Everything in Python are**[**objects**](https://www.programiz.com/python-programming/class). Names that we define are simply identifiers bound to these objects. [**Functions**](https://www.programiz.com/python-programming/function)**are no exceptions**, they are objects too (with attributes)

* 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, …

**Assigning a function to a variable**

def sandip():

print("I am sandip")

var = sandip #

sandip()

var()

both **sandip and var points to the same function object**

That's why we got the same output from both the function call.

**Functions are objects**

Python functions are **first class objects**. In the example below, we are assigning function to a variable. This assignment doesn’t call the function. It takes the function object referenced by F1 and creates a second name pointing to it, F2.

python functions are full objects. They **can have attributes and methods** like objects. The functions can have data variables and even functions written inside of them.

e.g.

**def** first(msg):

print(msg)

first("Hello")

second = first

second("Hello")

**names first and second refer to the same function object.**

e.g.

|  |
| --- |
| **def** F1(text):  **return** text.upper()    **print** (F1('Hello'))    F2 **=** F1    print (F2('Hello')) |

**Functions can be passed as arguments to another function.**

If you have used functions like **map, filter and reduce**

Such functions that take other functions as arguments are also called **higher order functions**

**e.g.**

def second(func):

print("Good Morning")

func()

def main():

print("Hello!")

second(main)

**Output:**

Good Morning

Hello

**e.g.**

def **isEven**(x):

if x%2==0:

return True

else:

return False

l=[0,5,10,15,20,25,30]

l1=list(filter(**isEven**,l))

print(l1) #[0,10,20,30]

**e.g.**

def inc(x):

return x + 1

def dec(x):

return x - 1

def operate(func, x):

result = func(x)

print(result)

operate(inc,3)

operate(dec,5)

**You can return the function from a function.**

**e.g.**

def **outer**():

def **inner**():

print("Hello")

return inner

new = outer() #calls outer storing in variable new

# Outputs "Hello"

new() #means we are calling inner

**e.g.**

def outer(msg):

def inner():

print(f"Hello {msg}")

return inner

x=outer("sandip")

x()

output:

Hello sandip

**e.g.**

def outer():

def inner():

print("Hello")

return inner() **#directly calls**

**outer()**

**e.g.**

def B():

print("method B.")

# first method that return second method

def A():

print("method A.")

**return B**

s=A()

s()

**e.g.**

def B(str2):

print("Good " ,str2, ".")

def A(str1, str2):

print(str1 , " and ", end = "")

return B(str2)

A("Hello", "Morning")

**Inner Functions:**

We can define a function inside other functions. Such functions are called inner functions or nested functions. Decorators in Python also use inner functions.

def parent():

print("I am the parent function")

def first\_child():

print("I am the first child function")

def second\_child():

print("I am the second child function")

first\_child()

second\_child()

parent()

output :

**Back to Decorators :**

Basically, a decorator **takes in a function**, **adds some functionality and returns it**.

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

Inside **the decorator function**, we are **defining another inner function called wrapper.** This is the **actual function that does the modification** by wrapping the passed function func. (**Logic**)

e.g.

def decorator(func):

def wrapper():

print("This is printed before the function is called")

func()

print("This is printed after the function is called")

return wrapper

def say\_hello():

print("Hello! The function is executing")

say\_hello = decorator(say\_hello)

say\_hello()

The **most important line** in the code is this:

say\_hello = decorator(say\_hello)

We passed the **say\_hello function to the decorator function**. In effect, **the say\_hello now points to the wrapper function** returned by the decorator.  
However, the **wrapper function has a reference** to the **original say\_hello()** as func, and calls that function between the two calls to print().

Therefore,

**A decorator function modifies a function by wrapping it in a wrapper function.**

**Syntactic Decorator**

The above decorator pattern got popular in the Python community but it was a little inelegant. We have to write the function name thrice and the decoration gets a bit hidden below the function definition.

Therefore, **Python introduced a new way to use decorators** by providing **syntactic sugar with the @ symbol.**

e.g.

def decor(func):

def inner():

print("This is awsome function..")

func()

return inner

def decor1(func):

def inner():

print("This is bad function..")

func()

return inner

#This is awsome function

def fun1():

print("This is function 1")

@decor1

def fun2():

print("This is function 2")

var=decor(fun1)

var()

fun2()

e.g.

**def** decor(func):  
 **def** inner(name):  
 **if** name==**"Sunny"**:  
 print(**"Sunny bro bad morning.."**)  
  
 **else**:  
 func(name)  
 **return** inner  
  
@decor  
**def** wish(name):  
 print(**"hello"**,name,**"Good Morning"**)  
  
wish(**"Sandip"**)  
wish(**"Sunny"**)

**Output:**

hello Sandip Good Morning

Sunny bro bad morning..

e.g.

**def** outer(func):  
 **def** inner(name):  
 **if** name == **'Priyanka' or** name == **'Sandip' or** name == **'Madhuri' or** name == **'Rohit' or** name == **'Seema' or** name == **'Nikhil' or** name == **'Shubham'**:  
 print(**"Hello"**, name, **"Welcome to group"**)  
 **else**:  
 func(name)  
  
 **return** inner  
  
@outer  
**def** group(name):  
 print(**"Hello"**, name, **"Not from group"**)  
  
  
group(**"Shubham"**)  
group(**"Nishigandha"**)

output:

Hello Shubham Welcome to group

Hello Nishigandha Not from group

**e.g.**

def decor(func):

def inner(a,b):

if a<b:

a,b=b,a

func(a,b)

return inner

def div(a,b):

print(a/b)

div=decor(div)

div(2,4)

**output** :

2.0

**We can give variable arguments also.**

e.g.

def outer(func):

def inner(**\*s**):

print("Good")

return func(\*s)

return inner

@outer

def F1(a):

print(f"Variable is {a}")

F1(4)

We got an error because the wrapper function we defined inside the decorator does not accept any argument.

So, **a better solution is to accept a variable number of arguments in the wrapper function** and then pass those arguments to the original function func.

\*args**and**\*\*kwargs**allow us to pass multiple arguments or keyword arguments to a function.**

def outer(func):

def inner(\*args,\*\*kwargs):

print("Good")

return func(\*args,\*\*kwargs)

return inner

@outer

def F1(a):

print(f"Variable is {a}")

F1(4)

**Reusing Decorator**

A decorator is just a regular Python function. Hence, we can **reuse** it to decorate multiple functions.

**import** functools  
  
  
**def** do\_twice(func):  
 @functools.wraps(func)  
 **def** wrapper():  
 func()  
 func()  
  
 **return** wrapper

**from** deco14 **import** do\_twice  
  
@do\_twice  
**def** f1():  
 print(**"Hello"**)  
f1()

**Decorator Chaining**

Chaining the decorators means that we can apply **multiple decorators to a single function.** These are also called nesting decorators.

*#Decorator chaining***def** deco1(func):  
 **def** inner(name):  
 **if** name==**'Preity'**:  
 print(**"Hello"**,name)  
 **else**:  
 **return** func(name)  
 **return** inner  
  
**def** deco2(func):  
 **def** inner(name):  
 **if** name==**'Akshay'**:  
 print(**"Hello"**,name)  
 **else**:  
 **return** func(name)  
 **return** inner  
  
  
@deco1   
@deco2   
**def** F1(name):  
 print(**"Hello"**, name)  
  
F1(**'sandip'**)  
  
F1(**'Preity'**)  
F1(**'Akshay'**)

## Python Iterators

Iterator in python is an **object** that is used to **iterate over iterable objects like lists, tuples, dicts, and sets**.

An iterator is an object that can be iterated upon, meaning that you **can traverse through all the values.**

The **iterator** is **initialized** using the **iter()**method. It uses the **next()** method for iteration.

1. \_\_**iter**(iterable)\_\_ method that is called for the **initialization of an iterator.** This **returns an iterator object**
2. \_\_**next**\_\_  The next **method returns the next value for the iterable.**

e.g.

mytuple = ("apple", "banana", "cherry")  
myit = **iter**(mytuple)  
  
print(next(myit))  
print(next(myit))  
print(next(myit))

Even **strings** are iterable objects, and can return an iterator:

mystr = "banana"  
myit = iter(mystr)  
  
print(next(myit))  
print(next(myit))  
print(next(myit))  
print(next(myit))  
print(next(myit))  
print(next(myit))

OR

print(myit.\_\_next\_\_())

When we use a **for loop to** traverse any iterable object, **internally** it uses the **iter() method to get an iterator object** which further uses **next()** method to iterate over.

In the following iterations, the for loop is internally(we can’t see it) **using iterator object** to **traverse over the iterables .**

# Sample **built-in iterators**

# Iterating over a list

print("List Iteration")

l = ['Today','is','Tuesday']

**for** i in l:

print(i) #For loop First convert l to **iterator** using **iter** –next –Stopiteration

#We **can not use next directly on l**

# Iterating over a tuple (immutable)

print("\nTuple Iteration")

t = ("This", "is", "tuple")

for i in t:

print(i)

# Iterating over a String

print("\nString Iteration")

s = "Brainworks"

for i in s :

print(i)

# Iterating over dictionary

print("\nDictionary Iteration")

d={1:'Hi',2:'Hello',3:'Bye'}

for x,y in d.items():

print(x,y)

**CREATING OWN ITERATOR :**

*#Creating own iterator***class** Iter1:  
 **def** \_\_init\_\_(self):  
 self.num=0  
  
 **def** \_\_iter\_\_(self):  
 **return** self  
  
 **def** \_\_next\_\_(self):  
 self.num+=1  
 **return** self.num  
  
   
  
obj=Iter1()  
print(obj.\_\_next\_\_())  
print(obj.\_\_next\_\_())

e.g.

**class** Iter1:  
 **def** \_\_init\_\_(self):  
 self.num = 0  
  
 **def** \_\_iter\_\_(self):  
 **return** self  
  
 **def** \_\_next\_\_(self):  
 self.num += 1  
 **if** self.num<=10:  
 **return** self.num  
 **else**:  
 **raise** StopIteration  
  
  
obj = Iter1()  
**for** i **in** obj:  
 print(i)

**GENERATOR**

If the **body of a def** contains **yield**, the function **automatically becomes a generator** function.

A generator-function is defined like **a normal function**, but **whenever** it **needs to generat**e a **value**, it does so with the [**yield keyword**](https://www.geeksforgeeks.org/use-yield-keyword-instead-return-keyword-python/)**rather than return**.

Python Generators are the functions that **return the traversal object** and **used to create iterators**. It **traverses the entire items at once.**

There is a lot of **complexity** in **creating iteration** in Python; we need to implement **\_\_iter\_\_() and \_\_next\_\_()** method to keep track of internal states.

It is a **lengthy** process to create iterators. That's why the **generator plays an essential role** in **simplifying** this process. If there is no value found in iteration, it raises StopIteration exception.

**def** simpleGeneratorFun():

**yield** 1

**yield** 2

**yield** 3

# Driver code to check above generator function

**for** value **in** simpleGeneratorFun():

**print**(value)

Generator **functions** return a **generator object.** Generator objects are **used** either by calling the **next** method on the generator object **or** using the generator object in a “**for** in” loop .

So a generator function **returns an generator object that is iterable,** i.e., **can be used as an**[**Iterators**](https://www.geeksforgeeks.org/iterators-in-python/)**(** An iterator is an object that can be iterated upon, meaning that you **can traverse through all the values.)**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| # A generator function  **def** simpleGeneratorFun():  **yield** 1  **yield** 2  **yield** 3    # x is a generator object  x **=** simpleGeneratorFun()    # Iterating over the generator object using next  print(x.next())  **print**(x.next())  print(x.next())  e.g.  **def** F1(n):  **for** i **in** range(n):  **yield** i  obj=F1(10) print(next(obj)) print(next(obj)) print(next(obj)) #one record/data at a time **Memory efficient ,speed**  e.g.  *#To iteratae over top to values* **def** gen():  num=1  **while**(num<=10):  **yield** num  num+=1  obj1=gen() print(obj1.\_\_next\_\_) print(next(obj1)) print(next(obj1)) **for** i **in** obj1:  print(i)  e.g. Square  **def** it():   **for** i **in** range(1,11):  **yield** i\*i  obj=it()  print(next(obj)) print(next(obj))  e.g.    **Summary**:  Generator functions allow you to **declare a function that behaves like an iterator.**  They allow programmers to make **an iterator in a fast, easy**, and clean way.   |  |  | | --- | --- | | **Iterators** | **Generators** | | Iterators are the objects that use the **next()** method to get the next value of the sequence. | A generator is a function that produces or yields a sequence of values using a **yield** statement. | | **Classes** are used to Implement the iterators. | **Functions** are used to implement the generator. | | **Every iterator is not a generator.** | **Every generator is an iterator.** | | **Complex** implementation of iterator protocols .i.e., iter() and next(). | Generators in Python are **simpler** to code than do the custom iterator using the yield statement. | | Iterators in python are **less memory efficient.** | Generators in Python are more memory efficient. |   **THANK YOU** |