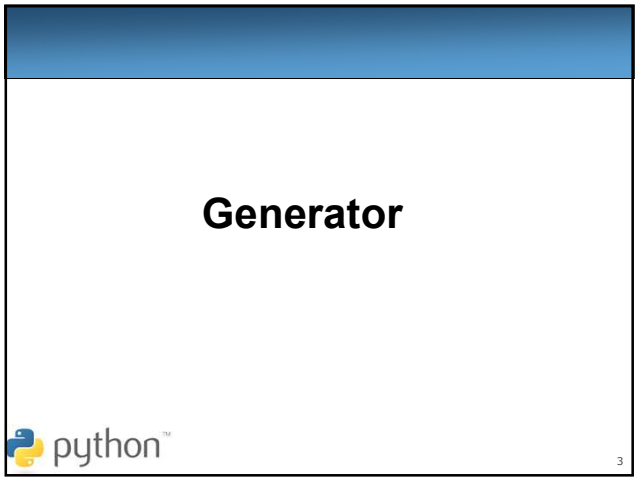

python
Programming with Python
Dr. Soharab Hossain Shaikh
BML Munjal University
https://github.com/soharabhossain/Python

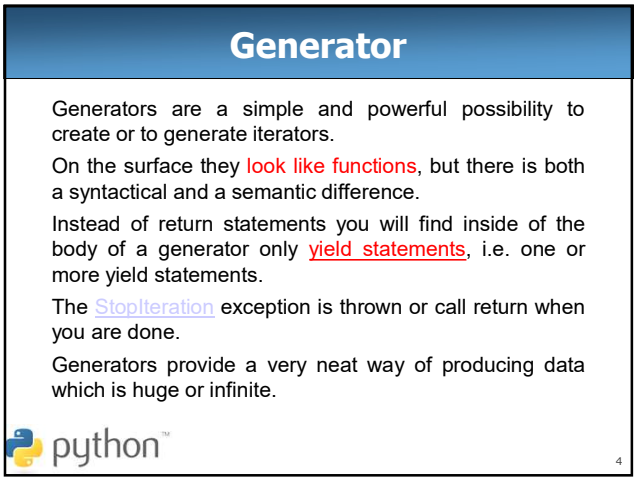
1


python
Generator & Decorator

2


Generator
python
3

3


Generator
Generators are a simple and powerful possibility to create or to generate iterators.
On the surface they look like functions, but there is both a syntactical and a semantic difference.
Instead of return statements you will find inside of the body of a generator only yield statements, i.e. one or more yield statements.
The StopIteration exception is thrown or call return when you are done.
Generators provide a very neat way of producing data which is huge or infinite.
python
4

4

How does a Generator work?

A generator is called like a function. It's return value is an iterator object. The code of the generator will not be executed in this stage.

The iterator can be used by calling the **next** method. The first time the execution starts like a function, i.e. the first line of code within the body of the iterator. The code is executed until a yield statement is reached.

yield returns the value of the expression, which is following the keyword yield. This is like a function, but Python keeps track of the position of this yield and the state of the local variables is stored for the next call.

At the next call, the execution continues with the statement following the yield statement and the variables have the same values as they had in the previous call.

The iterator is finished, if the generator body is completely worked through or if the program flow encounters a return statement without a value.

python

How to create a Generator?

Generators in Python:

- Are defined with the **def** keyword
- Use the **yield** keyword
- May use several yield keywords
- Return an **iterator**

python

Generator Example

```
def city_generator():
    yield("Gurgaon")
    yield("Delhi")
    yield("Bangalore")
    yield("Kolkata")

city = city_generator()
print(city)
print(next(city))
print(next(city))
print(next(city))
print(next(city))
#print(next(city)) # StopIteration - runtime error
```

python

Generator Output

```
>>>
========== RESTART: C:/Python/Python38-32/gen_test.py ========
<generator object city_generator at 0x032FB760>
Gurgaon
Delhi
Bangalore
Kolkata
```

python

Generator Example

```
def generate_ints(N):
    for i in range(N):
        yield i

gen = generate_ints(3)
print(gen)

print(next(gen))
print(next(gen))
print(next(gen))
#print(next(gen)) # StopIteration - runtime error
```

Generator Output

```
<generator object generate_ints at 0x0330F680>
0
1
2
>>>
```

Generator Example

```
# Generating an infite iterator
def fibonacci():
    """Fibonacci numbers generator"""
    a, b = 0, 1

    while True:
        yield a
        a, b = b, a + b

f = fibonacci()

counter = 0
for x in f:
    print (x)
    counter += 1
    if (counter > 10):
        break
```

Generator Output

```
<generator object fibonacci at 0x03CD1760>
0
1
1
2
3
5
8
13
21
34
55
>>>
```

Decorator

Python Function

In Python, the function is a first-class citizen.

It means that it can be passed as an argument to another function. It is also possible to define a function inside another function.

Such a function is called a nested function. Moreover, a function can return another function.

Decorator

A decorator is a function that receives another function as argument.

The behaviour of the argument function is extended by the decorator without actually modifying it.

Define a Decorator

```
def mydecoratorfunction(some_function): # function to be decorated passed as argument
   def wrapper_function(): # wrap the some_function and extends its behaviour
      # write code to extend the behaviour of some_function()
      some_function() # call some_function
      return wrapper_function # return wrapper function
```

Define a Decorator

In the above example, the some_function is a function whose behaviour we want to extend.

So, we will have to write a custom function like mydecoratorfunction(), which takes the some_function as an argument.

The wrapper_function() is an inner function where we can write additional code to extend the behaviour of the some_function, before or after calling it.

And finally, the wrapper_funtion() should be returned. In this way, Python includes decorator functions.

Also, we can define our own decorator function to extend the behaviour of a function without modifying it.

Example Decorator

```
# A custom function
def display(str):
    print(str)

# Decorator Function
def display_decorator(fn):
    def display_wrapper(str):
        print('Output:  ')
        fn(str)
        print('Completed executing the function....')
    return display_wrapper

# Call the function
out = display_decorator(display)
out('Hello World')
print('\n\n')
```

Output

```
Output:
Hello World
Completed executing the function....
```

Using @ to Decorate

```
# Defining Custom Decorator Function with inner function
def display_decorator(fn):
    def display_wrapper(str):
        print('Output:  ')
        fn(str)
        print('Completed executing the function....')
    return display_wrapper

@display_decorator
def display(str):
    print(str)

# Call the function
display('Hello India')
```

Output

```
Output:
Hello India
Completed executing the function....
```


21

@statismethod

The @staticmethod is a built-in decorator that defines a static method in the class in Python.

A static method doesn't receive any reference argument whether it is called by an instance of a class or by the class itself.

The following notation is used to declare a static method in a class (next slide)

22

Example

```
#----------------------------------------
# Static method
class Person:
    @staticmethod
    def greet():
        print("Hello!")

# Call with the name of the class
Person.greet()

# Call with an object
p1 = Person()
p1.greet()
```

```
Hello!
Hello!
>>>
```


23

@classmethod

```
# Class Method Decorator
# Person
class Employee:

    num_emp =0
    def __init__(self, name, age):
        self.name = name
        self.age = age
        Employee.num_emp +=1

    @classmethod
    def get_num_instance(cls):
        return Employee.num_emp

    def display(self):
        print(self.name + "'s age is: " + str(self.age))

# Create two objects
e1 = Employee('Adam', 35)
e1.display()
e2 = Employee('Alice', 27)
e2.display()

# Check the number of instances with the name of the class
print(Employee.get_num_instance())

# Also callble from an object
print(e2.get_num_instance())
```


24

Output

```
Adam's age is: 35
Alice's age is: 27
2
2
>>> |
```


25

Class methods for accessing property

```
class Person1:
    def __init__(self, name="Guest"):
        self.__name=name

    def setname(self, name):
        self.__name=name

    def getname(self):
        return self.__name

    def delname(self):
        del self.__name


p1 = Person1()

p1.setname('Turing')

name = p1.getname()
print(name)

p1.delname()
```


26

property()

The property() function is used to define properties in the Python class.

27

property()

```
class Person2:

    def __init__(self, name):
        self.__name=name

    def setname(self, name):
        print('setname() called')
        self.__name=name

    def getname(self):
        print('getname() called')
        return self.__name

    def delname(self):
        print('delname() called')
        del self.__name

    name = property(getname, setname, delname)


p1 = Person2('Elon')

p1.name = "Steve"

name = p1.name
print(name)

del p1.name
```


28

@property

@property decorator makes it easy to declare a property instead of calling the property() function.

29

@property

```
class Person3:

    def __init__(self):
        self.__name=''

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self, value):
        self.__name=value

    @name.deleter
    def name(self):
        print('Deleting..')
        del self.__name

p = Person3()

p.name = 'Bill'

name = p.name
print(name)

del p.name
```


30

Object Serialization

Object serialization is the process of converting state of an object into byte stream.

This byte stream can further be stored in any file-like object such as a disk file or memory stream.

It can also be transmitted via sockets etc. Deserialization is the process of reconstructing the object from the byte stream.

31

pickle

The dump() and load() functions of pickle module respectively perform pickling and unpickling of Python data.

dump() function writes pickled object to a file (or file like object)

load() function unpickles data from file back to Python object.

32

pickle

```
import pickle as pk

# Serialize
f = open("pic.pk","wb")
dct = {"name":"Raj", "age":23, "Gender":"Male","marks":75}
pk.dump(dct, f)
f.close()


# De-Serialize

f = open("pic.pk","rb")
d = pk.load(f)
print(d)
f.close()

"
```

pickle

```
class Person:
  def __init__(self, name, age):
     self.name=name
     self.age=age
  def show(self):
     print ("name:", self.name, "age:", self.age)

p1 = Person('Raj', 35)
print ("\n Pickling data....")
f = open("pickled.pk","wb")
pk.dump(p1,f)
f.close()
print ("\n Pickling completed....")

print ("\n Now Unpickling data....")
f = open("pickled.pk","rb")
p1 = pk.load(f)
p1.show()
```

End of Presentation