எங்கள் வாழ்வும் எங்கள் வளமும்
மங்காத தமிழ் என்று சங்கே முழங்கு … *புரட்சிக்கவி*

NOTICE

www.DataScienceInTamil.com

Day 30 - Batch 3 - Python Language

Chapter 016 Python Generators,Iterators and Iterable

To watch the recorded Python and Data Science videos in YouTube:

Day 30- Batch 3 - Python Generators,Iterators and Iterable - Part 1

https://youtu.be/tL7ww2N0fEI

Official Website:

https://DataScienceInTamil.com/

மேலும் முக்கிய கேள்விகள் பதில்களுக்கு :

https://www.DatascienceInTamil.com/#faq

To join DataScienceInTamil Telegram group:

இந்த குழுவில் உங்கள் நண்பர்களை இணைக்க விரும்பினால் அதற்கான லிங்க்

https://t.me/joinchat/lUZEsr-zidpjZjEx

To Join the class, please fill the form :

https://forms.gle/QFpLHwAoinFaX2cE6

Join Zoom Meeting (From Sep 26 2022 to Oct 26 2022)

https://us06web.zoom.us/j/88900302653?pwd=MVBFUlhqTTE1LzFFRUVpTzZ2S1Vsdz09

Meeting ID: 889 0030 2653

Passcode: 1234

Monday through Friday 8 PM to 10 PM IST (From Sep 26 2022 to Oct 26 2022)

We support open-source products to spread Technology to the mass.

- This is completely a FREE training course to provide introduction to Python language
- All materials / contents / images/ examples and logo used in this document are owned by the respective companies / websites. We use those contents for FREE teaching purposes only.
- We take utmost care to provide credits whenever we use materials from external source/s. If we missed to acknowledge any content that we had used here, please feel free to inform us at info@DataScienceInTamil.com.

- ➢ All the programming examples in this document are for FREE teaching purposes only.

Thanks to all the open-source community and to the below websites from where we take references / content /code example, definitions, etc., please use these websites for further reading:

- Book : Python Notes For Professionals
- https://www.w3schools.com
- https://www.geeksforgeeks.org
- https://www.askpython.com
- https://docs.python.org
- https://www.programiz.com/
- https://www.openriskmanagement.com/
- https://pynative.com/python-sets/
- https://www.alphacodingskills.com/
- https://codedestine.com/
- https://appdividend.com/
- https://freecontent.manning.com/
- https://pynative.com/python-for-loop/
- https://stackoverflow.com/
- https://datagy.io/python-isdigit
- https://pynative.com/python-range-function/
- https://www.datacamp.com/community/tutorials/functions-python-tutorial
- https://data-flair.training/blogs/python-function/

- https://problemsolvingwithpython.com/07-Functions-and-Modules/07.07-Positional-and-Keyword-Arguments/
- https://www.tutorialsteacher.com/python/callable-method
- https://www.geeksforgeeks.org/python-exception-handling/
- https://www.javatpoint.com/python-exception-handling
- https://www.tutorialspoint.com/python/python_exceptions.htm
- https://www.w3schools.com/python/python_try_except.asp
- https://www.programiz.com/python-programming/exception-handling
- https://www.datacamp.com/community/tutorials/exception-handling-python
- https://www.geeksforgeeks.org/python-exception-handling/
- https://www.javatpoint.com/python-exception-handling
- https://www.tutorialspoint.com/python/python_exceptions.htm
- https://www.w3schools.com/python/python_try_except.asp
- https://www.programiz.com/python-programming/exception-handling
- https://www.datacamp.com/community/tutorials/exception-handling-python
- www.sentinelone.com/blog/python-stack-trace-understanding-using-debug/
- https://riptutorial.com/python/example/5532/re-raising-exceptions
- https://www.techbeamers.com/python-generator/
- https://nvie.com/posts/iterators-vs-generators/

What to cover

1. Python Generators, Iterators and Iterable
2. Iterables

24. Call the function with ‘return’ vs ‘yield’ - ‘return’
25. Call the function with ‘return’ vs ‘yield’ - ‘yield
26. Send 2 iterbales to generator fn
27. using * (args notation)
28. The generator fn give preference only to the generator object to be produced, other codes will NOT be executed
29. Store the value of the generator object as list
30. Resetting a generator (becose generator is not re-entrant)
31. How to reset a generator
32. Generator are not re-entrant – so store it in a list
33. Infinite sequences can be controlled by generator function
34. Sending objects to a generator /yield
35. How to work with fn, with out sending data (parameter)?
36. Send() accepts only one parameter to fn
37. under what circumstances we send object to generator
38. Nested loop and generater object
39. Yield from
40. yield from , takes any number of iterables and adds to gether
41. *args and yield from
42. Yield from and a function()
43. Iteration
44. next(iterator[, default])

Python Generators, Iterators and Iterable

Iterables

An iterable is **any Python object capable of returning its members one at a time**, permitting it to be iterated over in a for-loop. Familiar examples of iterables include lists, tuples, and strings - any such sequence can be iterated over in a for-loop

1. Definition: iterable (adjective) -- that which can be iterated over.
2. A good test of whether something is iterable is whether it can be used in a **for: statement.**

3. For example, if we can write for item in X:, then X is iterable.

Example for iterables:

●Containers -- We can iterate over **lists, tuples, dictionaries, sets**, And other containers.

● Some built-in (non-container) types -- Examples:

○A text file open in read mode is iterable: it iterates over the lines in the file see the book for example and string

Python Iterators

- An iterator is an object that contains a countable number of values.
- An iterator is an object that can be iterated upon, meaning that you can traverse through **all** the values using next(), list() or for loop
- Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods `__iter__()` and `__next__()`.

Iterator vs Iterable

Lists, tuples, dictionaries, and sets are all iterable objects (via iter() or __iter()__. They are iterable *containers* which you can get an iterator from.

All these objects have a `iter()` method which is used to get an iterator:

Example

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)
print(type(myit))

mylist = ["apple", "banana", "cherry"]
myit = iter(mylist)
```

```
print(type(myit))

myset= {"apple", "banana", "cherry"}
myit = iter(myset)
print(type(myit))

mytuple = {"apple":1, "banana":2, "cherry":3}
myit = iter(mytuple)
print(type(myit))

sub = "DataScience"
myit = iter(sub)
print(type(myit))

inFile = open(file="BANK100.TXT", mode='r')
print(type(inFile)) # by default it is an iterator object, ie, we no need to use  iter() to convert
to iterator
print(next(inFile)) # it prints the FIRST LINE only
```

output

```
<class 'tuple_iterator'>

<class 'list_iterator'>

<class 'set_iterator'>
```

```
<class 'dict_keyiterator'>
<class 'str_iterator'>
=================
```

Return an iterator from a tuple, and print each value:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit)) # NEXT(), TAKES AN ITERABLE OBJJECT TO IT
PARAMETER
print(next(myit))
print(next(myit))
```

output

```
apple
banana
cherry
```

Notes: If we print only the inerrable object myit, we will get this as an output

```
<tuple_iterator object at 0x000001DCC6BDE088>
```

Generator

a generator is a function that returns an object (generator object) which we can iterate over (one value at a time).

How to create a generator in Python?

1. An easy way to define an object that obeys the iterator protocol is to write a generator function. A generator function is a function that contains one or more yield statements.

2. If a function contains at least one yield statement, then that function when called, returns generator iterator, which is an object that obeys the iterator protocol, i.e. it'sa n iterator object.

3. Note that in recent versions of Python, yield is an expression. This enables the consum er to communicate back to the producer (the generator iterator)

4. It is fairly simple to create a generator in Python. It is as easy as defining a normal function, but with a yield statement instead of a return statement.

5. If a function contains at least one yield statement (it may contain other yield or return statements), it becomes a generator function. Both yield and return will return some value from a function.

6. The difference is that while a return statement terminates a function entirely, yield statement pauses the function **saving all its states** and later continues from , from where it stopped (there on successive calls).

Differences between Generator function and a Normal function

Here is how a generator function differs from a normal function.

- Generator function contains one or more yield statements.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like __iter__() and __next__() are implemented automatically. So we can iterate through the items using next().

- Once the function yields, the function is paused and the control is transferred back to the caller.
- **Local variables and their states are remembered between successive calls.**
- Finally, when the function terminates, StopIteration is raised automatically on further calls.

Generator functions

Generator functions are similar to regular functions, except that they have one or more yield statements in their body. Such functions cannot return any values (but returns only iterator object. To return the values, use object in the for loop) (however empty returns are allowed if you want to stop the generator early).

```
def fn1():
   yield

a = fn1()
```

```
print(a)
print(a.__next__())

print(list(a))
for item in a:
    print(a)
```

output

```
<generator object fn1 at 0x01378808>

None

[]
```

Normal fn terminates after the return statement

```
def fn1():
    for x in range(3, 10):
        return x**4
```

```
result = fn1()
print(result)
```

output

81

Generator fn returns only generator object and DOES not terminate the fn,

```
def fn1():
    for x in range(3,10):
        yield x**4

result = fn1()
print(result)
print(next(result))
print(next(result))
```

ouput

```
<generator object function at 0x000002349083C248>
-----------------
```

By default generator objects traverses only once

Note: Calling a generator function produces a generator object (not values), which can later be iterated over. Unlike other types of iterators, **generator objects may only be traversed once, unless we call the generator function again**

```
def fn1():
   yield 10

a = fn1()
print(a)
print(a.__next__())

a = fn1() # again we call the method and extract the value from the generator
print(a)
print(a.__next__())
```

output

```
<generator object fn1 at 0x01A18808>
10
<generator object fn1 at 0x01A189C8>
10
-----------------
```

Use for-loop / list() / next() to access the values from generator object

To return the values use the object inside the for loop as below

Note: Calling a generator function produces a generator object, which can later be iterated over. Unlike other types of iterators, generator objects may only be traversed once.

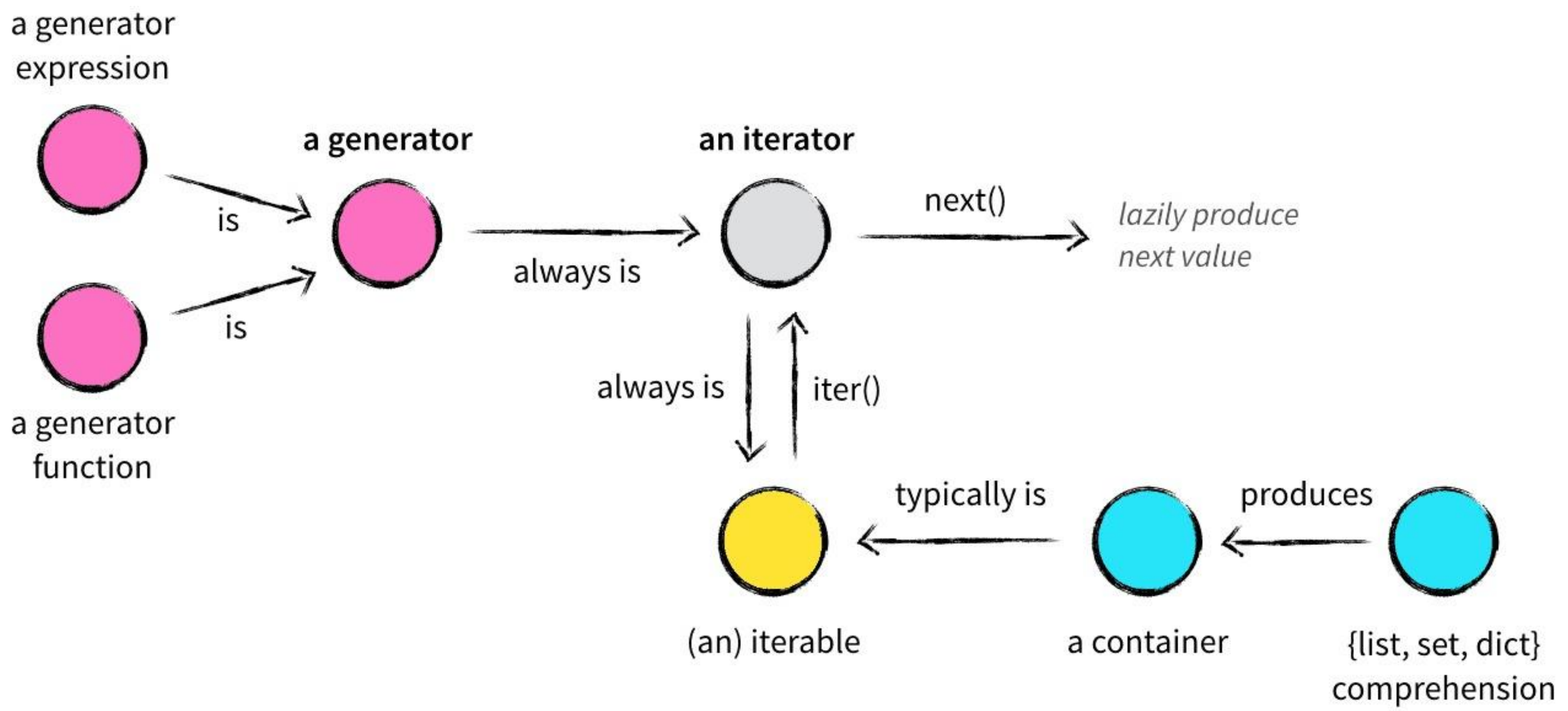
```python
def function():
    for x in range (5):
        yield x **2
```

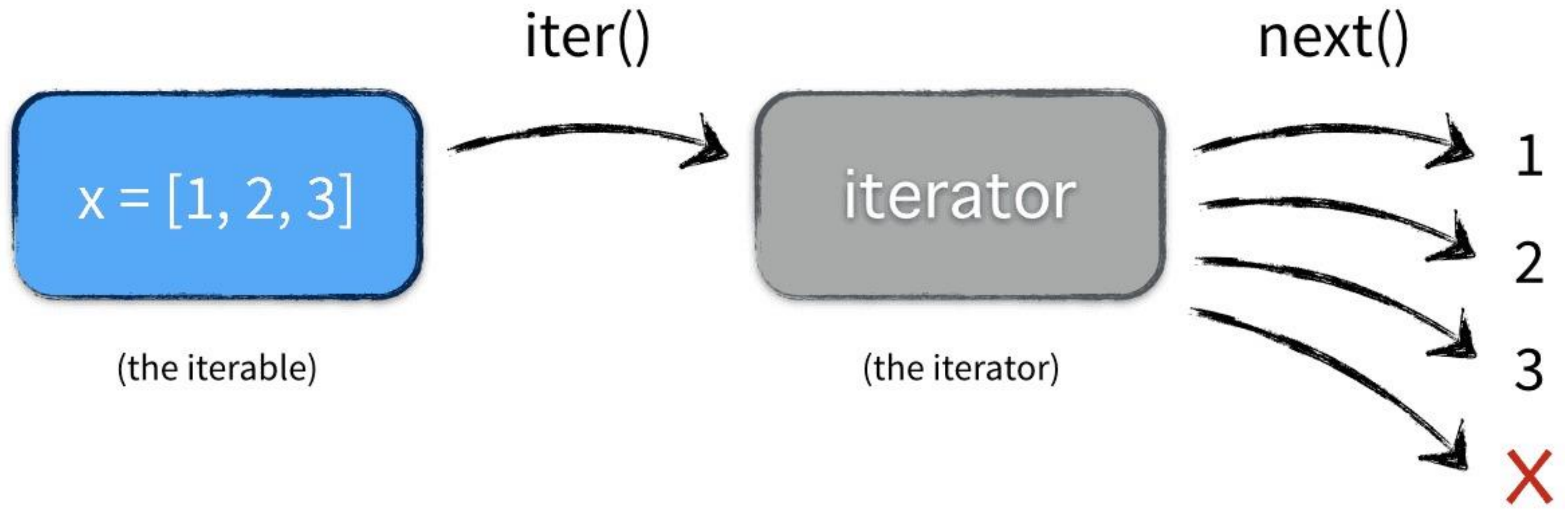
```
a= function()

print (a)
for val in a:
   print (val)
```

output

```
<generator object function at 0x0000019A88CFC248>
0
1
4
9
16
```


a generator expression
a generator function
is
is
a generator
always is
an iterator
next()
lazily produce next value
always is
iter()
(an) iterable
typically is
a container
produces
{list, set, dict} comprehension



Generators have been an important part of Python ever since they were introduced with PEP 255.

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.

What’s an iterator, you may ask?

An iterator is an object that can be iterated (looped) upon. It is used to abstract a container of data to make it behave like an iterable object. You probably already use a few iterable objects every day: strings, lists, and dictionaries to name a few.

An iterator is defined by a class that implements the Iterator Protocol. This protocol looks for two methods within the class: _iter_ and _next_.

Whoa, step back. Why would you even want to make iterators?

Saving memory space

Iterators don’t compute the value of each item when instantiated. They only compute it when you ask for it. This is known as lazy evaluation.

Lazy evaluation is useful when you have a very large data set to compute. It allows you to start using the data immediately, while the whole data set is being computed.

Use yield and return in generator function

‘return’ helps to stop the generator early

empty returns are allowed if you want to stop the generator early – see below

we break the generator function using empty return

```
def function():
   for x in range (5):
      yield x **2
```

```
    if x == 3:
        return

function()
a= function()
print (a)
for val in a:
    print (val)
```

output

```
<generator object function at 0x00000278FF3AC5C8>
0
1
4
9
```

Note:

1. empty retuns means return keyword with out any value
2. use return ‘ ’ (it gives empty return)
3. return 100 (this is not the correct empty return

Generator execution example

```
def fn1(name, age):
   print("Function executed")  # only gets executed when __next__() method called at the first time
   yield name
   yield age
   for item in range(age):
      if item == 5:
         print(item)
         return None # or return item gives the same value. return item is not same as yield item
      yield item

a = fn1("Nathan", 10)
print(a)
print(next(a))
print(next(a))
print(next(a))
```

output

```
<generator object fn1 at 0x00FE8808>
Function executed
Nathan
10
0
```

Notes:

Implement a generator function

Implement a generator function -- The generator produced should yield all values from a list/iterable that satisfy a predicate.

It should apply the transforms before return each value.
The function takes these arguments:
values -- A list of values. Actually, it could be any iterable.
predicate -- **A function that takes a single argument**, performs a test on that value, and returns True or False
transforms -- (optional) A list of functions. Apply each function in this list and returns the resulting value.

Write a simple code to the above notes - How to find the length of the iterator object

See the below pgm and it will do

- It tells you how to convert the iterable object to iterator object and then pass the iterator object as an argument to a fn parameter
- It tells you how to convert the iterator object to list and find the length from the list
- It tells you how many remaining items are available inside the iterator object

```
def fn1(lst):
   yield lst

lst1 = [1,100, 2, 200] # iterable object
iter_a = iter(lst1) # converting to iterating object using iter()...

print(fn1(iter_a))  # passing the Iterator object as an argument to fn

print(next(iter_a)) # accessing value from the iterator object using
```

next()

```
print(next(iter_a))
print(next(iter_a))
print(next(iter_a))

b =list(iter_a)
print(b)

print("Balance elements inside the iterator object ", len(b))
```

output

```
<generator object fn1 at 0x000001F64310D148>

1

100

2

200

Balance elements inside the iterator object  0/zero

===========
```

Dict iterator

```
d = {1:10,2:20,3:30}

myit = iter(d) # iter(d.keys)
print(myit)

myit = iter(d.values())
print(myit)

myit = iter(d.items())
print(myit)
```

output

```
<dict_keyiterator object at 0x01D3D438>
<dict_valueiterator object at 0x01D3D3C0>
<dict_itemiterator object at 0x01D3D438>
```

Note – look closely: key() and items() have same iterator object

Access the values from dict generagtor

```
d = {1:10,2:20,3:30}

myit = iter(d)
```

```
print(myit)
print(myit.__next__())
print(myit.__next__())
print(myit.__next__())

myit = iter(d.values())
print(myit)
print(next(myit))
print(next(myit))
print(next(myit))

myit = iter(d.items())
print(myit)
print(next(myit))
print(next(myit))
print(next(myit))
```

output

```
<dict_keyiterator object at 0x018CD438>
1
2
3
<dict_valueiterator object at 0x018CD3C0>
10
20
30
<dict_itemiterator object at 0x018CD438>
(1, 10)
(2, 20)
(3, 30)
```

What happens if we provide yield AFTER ‘return’ – see below

```
def fn1(name, age, salary):
    print("Function executed")  # only gets executed when __next__() method called at the first time
    yield name
    yield age
    yield salary

    for item in range(age):
        if item == :
            return
        yield item

    yield "Bonus is 500000" # after the return, no yield will work ie, generator is broken by 'return'

a = fn1("Usha", 10, 200000)
print(a)
print(list(a))
```

output

```
<generator object fn1 at 0x01738840>
Function executed
['Usha', 10, 200000, 0, 1, 2, 3, 4]
------------
```

What happens if we provide yield AFTER ‘break – see below

```
def fn1(name, age, salary):
    print("Function executed")  # only gets executed when __next__() method called at the first time
    yield name
    yield age
    yield salary

    for item in range(age):
        if item == 5:
            break # break comes only from  the for loop
        yield item

    yield "Bonus is 500000"

a = fn1("Usha", 10, 200000)
print(a)
print(list(a))
```

output

```
<generator object fn1 at 0x01BE7878>
Function executed
['Usha', 10, 200000, 0, 1, 2, 3, 4, 'Bonus is 500000']
```

Note: Generators return iterables (the return is always an iterator object): (reason : it has yield method, if a function has yield keyword that is called generator function

When it becomes generator function

```
def foo (): # foo isn't iterable yet...this is a normal fn
   yield 1      # now it become generator fn
   print ("DDDD")

result = foo() # ...but result already become a generator object
print (result)
'''IT  PRINTS GENERATOR OBJECT FOO, ie generator fuction
returns an ITERATOR OBEJCT'''
'''IF WE WANT RESULT FROM THE ITERATOR OBJECT USE
NEXT()'''
a= next(result)
print (a)
```

output

```
<generator object foo at 0x00000268E1B2C648>

1
```

Yield gives generator object not the value

```
def fn1(name, age):
    yield name
    yield age
    for item in range(age):
        yield item

a = fn1("Nathan", 5)
print(a)
```

output

```
<generator object fn1 at 0x01A47878>
```

Access generator values using – for loop, next(), list

Yield can be given inside the for loop (where as if we give 'return' inside the for loop, it gives only the first value)

```
def fn1( ):
    for item in range(3,10,2):
       yield item
```

```
a = fn1()
print(list(a))
```

output

```
[3, 5, 7, 9]
```

Another example

```
def fn1(name, age):
    yield name
    yield age
    for item in range(age):
        yield item

a = fn1("Mahesh", 5)
print(a)
print(list(a))
```

output

```
<generator object fn1 at 0x01F58808>

['Mahesh', 5, 0, 1, 2, 3, 4]
```

```
def fn1(name, age):
   yield name
   yield age
   for item in range(age):
      if item == 5:
         print(item)
         return None # or return item gives the same value. return item
is not same as yield item
      yield item

a = fn1("Nathan", 10)
print(a)
print(list(a))
```

output

```
<generator object fn1 at 0x00CC8808>
5
['Nathan', 10, 0, 1, 2, 3, 4]
```

----

build your own iterator using __iter__ and __next__ methods

you can build your own iterator using __iter__ and __next__ methods.

There is a lot of work in building an iterator in Python. We have to implement a class with __iter__() and __next__() method, keep track of internal states, and raise StopIteration when there are no values to be returned.

This is both lengthy and counterintuitive. Generator comes to the rescue in such situations.

Python generators are a simple way of creating iterators. All the work we mentioned above are automatically handled by generators in Python.

Generator and Iterator are n't reentrant!

```
def get(sal, bonus):
    yield 1
    yield 2
    yield  sal + bonus

iterable = get(10,20)
print(type(iterable))
print(next(iterable))
print(next(iterable))
print(next(iterable))
```

output

```
<class 'generator'>

1

2

30
```

```
# What was the first item of iterable? No way to get it now.
# Only to get a new iterator
a =get(1,2)
print(next(a))
```

output

When to use a Generator?

There are many use cases where generators can be useful. We have mentioned some of them here:

1. Generators can help to process large amounts of data. They can let us do the calculation when we want, also known as the lazy evaluation. The stream processing uses this approach.
2. We can also stack the generators one by one and use them as pipes as we do with the Unix pipes.
3. The generators can also let us establish concurrency.(ie it shows an item and also remove the SAME item the sequences / iterable
4. We can utilize Generators for reading a vast amount of large files. It will help in keeping the code cleaner and leaner by **splitting the entire process into smaller entities.**
5. Generators are super useful for **web scraping** and help increasing **crawl** efficiency. They can allow us to fetch the single page, do some operation, and

move on to the next. This approach is far more efficient and straightforward than retrieving all pages at once and then use another loop to process them.

Generators are lazy iterators created by generator functions (using yield) or generator expressions (using (an_expression for x in an_iterator)).

Data Crawling Meaning is to deal with large data-sets where you develop your crawlers (or bots) which crawl to the deepest of the web pages. Data scraping, on the other hand, refers to **retrieving information from any source** (not necessarily the web). It’s more often the case that irrespective of the approaches involved, we refer to extracting data from the web as scraping (or harvesting) and that’s a serious misconception

Data Crawling Meaning is to deal with large data-sets where you develop your crawlers (or bots) which crawl to the deepest of the web pages. Data scraping, on the other hand, refers to retrieving information from any source (not necessarily the web).

Stack works on the principle of “Last-in, first-out”. Also, the inbuilt functions in Python make the code short and simple. To add an item to the top of the list, i.e., to push an item, we use **append()** function and to pop out an element we use **pop()** function. These functions work quiet efficiently and fast in end operations.

Stacks and queues are simple data structures that allow us to store *and* retrieve data sequentially. In a *stack*, the last item we enter is the first to come out. In a *queue*, the first item we enter is the first come out. We can add items to a *stack* using the push operation *and* retrieve items using the pop operation

QUEUE: The syntax is also similar,the only difference is when removing elements you use **popleft.(from collections import deque)**

Generator expressions (List comprehension / set/ dict)

Generator expressions are similar to list, dictionary and set comprehensions, but are enclosed with parentheses. The parentheses do not have to be present when they are used as the sole argument for a function call.

```
expression = (x **2 for x in range (5))
print (expression)
print (next(expression))
print (next(expression))
print (next(expression))
print (next(expression))
print (next(expression))
print (next(expression))
```

output

```
<generator object <genexpr> at 0x000001CC7AB9C5C8>

0

1

4
```

```
9
16
print (next(expression))
StopIteration
```

This example generates the 5 first perfect squares, including 0 (in which x = 0).

Call the function with 'return' vs 'yield' - 'return'

```
def fn1(*args):
    return lst

lst = [10, 20, "AA", "BBB", True]
print("Original list    ",lst)
print("Type of the original list   ", type(lst))
print("Normal fn id, every fn has its own id", fn1)

print("Calling fn with ARGUMENT, it returns values", fn1(lst))
```

output

```
Original list    [10, 20, 'AA', 'BBB', True]
```

```
Type of the original list   <class 'list'>
Normal fn id, every fn has its own id <function fn1 at 0x01483028>
Calling fn with ARGUMENT, it returns values [10, 20, 'AA', 'BBB', True]
--------------
```

Call the function with ‘return’ vs ‘yield’ - ‘yield

We know how to call a fn with return , now look at how we call a generator fn (ie with yield)

```
def fn1(*args):
    yield args

lst = [10, 20, "AA", "BBB", True]
print("Original list    ",lst)
print("Type of the original list   ", type(lst))
print("Normal fn id, every fn has its own id", fn1)

print(fn1(lst))
# print(next(fn1(lst)))
```

output

```
Original list    [10, 20, 'AA', 'BBB', True]
Type of the original list   <class 'list'>
Normal fn id, every fn has its own id <function fn1 at 0x01B93028>
<generator object fn1 at 0x01C27A70>
-----------------
```

Another code for yield

```python
def fn1(*args):
    yield args[0]
    yield args[1]
    yield args[2]
    yield args[3]
    yield args[4]

lst = [10, 20, "AA", "BBB", True]

result = fn1(*lst) # * unpack all the list values individally
print(result)
print(next(result))
print(next(result))
```

output

```
<generator object fn1 at 0x0000018269FEE420>
```

```
10

20
```

Send 2 iterbales to generator fn

```
mylist = ["apple", "banana", "cherry"]
mytuple = ("appleT", "bananaT", "cherryT")
iterList = iter(mylist)
# print(next(iterList))

iterTuple = iter(mytuple)
# print(next(iterTuple))

# print ("List iter object ", iterList)
# print ("Tuple list object ", iterTuple)

def fn1(*args):
    yield args

a = fn1(iterTuple, iterList)
print("This is generator object ",a)
c = (next(a))
print ("This is list iterator object produced by generator object ",c)

for obj in c:
    print(obj)
    for val in obj:
        print(val)
```

output

```
This is generator object  <generator object fn1 at 0x000001F3E261D148>

This is list iterator object produced by generator object  (<tuple_iterator object at
0x000001F3E28BAE88>,
<list_iterator object at 0x000001F3E28BAE48>)
<tuple_iterator object at 0x000001F3E28BAE88>

appleT
bananaT
cherryT
<list_iterator object at 0x000001F3E28BAE48>
apple
banana
cherry
-------------------
```

using * (args notation)

The above pgm can be modified as below

Send 2 iterables as args argument using * (args notation)

```
mylist = ["apple", "banana", "cherry"]
mytuple = ("appleT", "bananaT", "cherryT")

iterList = iter(mylist)
# print(next(iterList))

iterTuple = iter(mytuple)
# print(next(iterTuple))

# print ("List iter object ", iterList)
# print ("Tuple list object ", iterTuple)

def fn1(*args):
    yield args

tupleAndList = (iterTuple, iterList)
a = fn1(*tupleAndList)                              #observe the way we use * to send multi
arguments to *args parameter
print("This is generator object ",a)
c = (next(a))
print ("This is list iterator object produced by generator object ",c)
#
for obj in c:
    print(obj)
    for val in obj:
        print(val)
```

output

```
This is generator object  <generator object fn1 at 0x01287A70>
This is list iterator object produced by generator object  (<tuple_iterator object at 0x01EF3148>, <list_iterator object at 0x0127EE80>)
<tuple_iterator object at 0x01EF3148>
appleT
bananaT
cherryT
<list_iterator object at 0x0127EE80>
apple
banana
cherry
-----------------
```

The generator function will give preference only to the generator object

to be produced, other codes will NOT be executed

```
def fn():
    print("Before yield") #this will not be printed even if we call this fn
    yield 4

a = fn()
print(a)
```

ouput

```
<generator object fn at 0x000001CB992AD148>
```

Note:
The print statement will not be printed. This generator fn returns only object
If we want to view the print statement use next() with the generaor id- see below

```python
def fn():
    print("Before yield")
    yield 4

a = fn()
print(a)
print(next(a))
```

output

```
<generator object fn at 0x0000022E9FBFD4C8>

Before yield

4
```

Another example

```python
def fn():
    print("Before yield")
    for val in range(6):
        print("To check if this can be printed without next()")
        print(val)
        yield val + 10

a = fn()
print(a)
print(next(a)) #if we don’t use this next(), any print statement will not be printed
```

output

```
<generator object fn at 0x000001F24329D148>

Before yield

To check if this can be printed without next()

0

10
```

Another example

```
def fn():
    print("Before yield")
    for val in range(6):
        print("To check if this can be printed without next()")
        print(val)
        yield val + 10 # for every time next() is called, Python will run
from 0 to 5

a = fn()
print(a)
print(next(a))
print(next(a))
```

output

```
<generator object fn at 0x01C28808>

Before yield

To check if this can be printed without next()

0

10

To check if this can be printed without next()

1

11
```

Notice that a generator's body is not immediately executed: when you call function() in the example above, it immediately returns a generator object, without executing even the first print statement. This allows generators to consume less memory than functions that return a list, and it allows creating generators that produce infinitely long sequences

For this reason, generators are often used in data science, and other contexts involving large amounts of data. Another advantage is that other code can immediately use the values yielded by a generator, without waiting for the complete sequence to be produced.

==========

Store the value of the generator object as list

However, if you need to use the values produced by a generator more than once, and if generating them costs more than storing, it may be better to store the yielded values as a list than to re-generate / reset the sequence. See 'Resetting a generator' below for more details

```
def function():
   for x in range (5):
      yield x **2

function()
a= function()
for val in a:
   lst = list (a)
   print (lst)
output
[1, 4, 9, 16]
----------------------
```

Since generator objects are iterators, one can iterate over them manually using the **next()** function. Doing so, will return the yielded values one by one on each subsequent invocation

Under the hood, each time you call **next()** on a generator, Python executes statements in the body of the generator function until it hits the next yield statement. At this point it returns the argument of the yield command, and **remembers the point where that happened (this is called maintains the**

STATE of the object). Calling **next()** once again will resume execution from that point and continue until the next yield statement.

If Python reaches the end of the generator function without encountering any more yields, a **StopIteration** exception is raised (this is normal, all iterators behave in the same way).

```
def function():
   for x in range (3):
      yield x **2

function()
a= function()
print (next(a))
print (next(a))
print (next(a))
print (next(a))
print (next(a))
```

output

```
StopIteration
```

```
0
1
4
-------------------
```

Resetting a generator

Remember that you can only iterate through the objects generated by a generator once. If you have already iterated through the objects in a script, any further attempt do so, will yield **None**.

```
def function():
   for x in range (3):
      yield x **2

function()
a= function()

for val in a: # first time iteration using for loop
   print (val)
```

```
for val in a:    # second  time iteration using for loop
    print (val)
```

outout

```
0

1

4
```

Note: here we iterate through the generator object 2 times using the for loop. But we get the result only from the first loop.

Generator can be closed in between (use close())

```
def genfun():
    while True:
        print('calling yield  1')
        yield 'a'
        print('calling yield 2')
        yield 'b'
        print('calling yield 3')
        yield 'c'
```

```
a=genfun()
print(a)
print(next(a))
print(next(a))
a.close()
print(next(a))
```

output

```
StopIteration
<generator object genfun at 0x01AC7798>
calling yield 1
a
calling yield 2
b
```

Note: If we close the gen obj using close(), we can not access further values (though we have values inside the gen object

Another way of closing the gen obj from outside

Even if we use the next(), after the for loop we get the error..see below

```
def function():
   for x in range (3):
      yield x **2
```

```
function()
a= function()
for val in a:
   print (val)

print (a)
print (next(a)
```

output

```
0
1
4
<generator object function at 0x000001AE9CBEC5C8>
  print (next(a))
StopIteration
```

Note:If we use next(a)we get error
If we use list(a), we get empty list
If we use for item in a – we get nothing

How to reset a generator / revisit the code

If you need to use the objects generated by a generator more than once, you can either **define the generator function again and use it a second time**,

or,

alternatively, you can store the output of the generator function in a list on first use. Re-defining the generator function will be a good option if you are dealing with large volumes of data, and storing a list of all data items would take up a lot of disc space. Conversely (contrarily), if it is costly to generate the items initially, **you may prefer to store the generated items in a list so that you can re-use them**

Here we declare 2 different functions to re-use the same generator function object– see the pgm here

```
def function():
    for x in range (3):
        yield x **2

def function1():
    for x in range (3):
        yield x **2

function()
a= function()
print (a)
for val in a:
    print (val)

a=function1()
print (a)
for val in a:
    print (val)
```

output

```
<generator object function at 0x0000021EE462C5C8>

0
```

```
1
4
<generator object function1 at 0x0000021EE462C248>
0
1
4
```

Generator are not rentrant – so store it list

store the output of the generator function in a **list** on first use

```
def function():
   for x in range (5):
      yield x **2

function()
a= function()
print (a)
```

```
lst = list (a)
print (lst)
```

output

```
<generator object function at 0x000002DE3C1FC248>

[0, 1, 4, 9, 16]

----------
```

Now the list item can be iterated using loop – see below

```
def function():
   for x in range (5):
      yield x **2

function()
a= function()
print (a)
lst = list (a)
print (lst)

for val in lst:
   print (val)
```

output

```
<generator object function at 0x00000240B1F3C5C8>
[0, 1, 4, 9, 16]
0
1
4
9
16
```

Note, the generator oobject can be stored in tuple and set also (apart from list)

Infinite sequences can be controlled by generator function

while True means loop forever. The **while** statement takes an expression and executes the loop body **while** the expression evaluates to (boolean) "**true**". **True** always evaluates to boolean "**true**" and thus executes the loop body indefinitely.

Generators can be used to represent infinite sequences:

```
def integers_starting_from(n):
   while True:
      yield n
      n = n+1
natural_numbers = integers_starting_from(1)
print (natural_numbers)
for val in natural_numbers:
   print((val))
```

output

```
<generator object integers_starting_from at 0x00000203E03CC248>

1

2

3

4
```

NOTE: it keeps add number infinitely

Infinite sequence of numbers as above can also be generated with the help of itertools.count. The above code could be written as below

naturals_numbers = itertools.count(1)(doubt , there is not an output)

Sending objects to a generator /yield

Note: via send(), we send the value to the yield

In addition to receiving values from a generator(As generator objects), it is possible to send an object to a generator using the send() method.

Note: The generator fn, receive the value that we send from send(), MUST store in the below format

a = **yield** (variable name = **yield)**

then the value can be manipulated and then the result is yield-ed

```
send(...) method of builtins.generator instance
    send(arg) -> send 'arg' into generator,
    return next yielded value or raise StopIteration.
```

None

```
def informToBank():
    a = yield
    yield a + 10

msg = informToBank()
next(msg)
print(msg)
print(msg.send(10))
```

output

```
<generator object informToBank at 0x0000023B64C9D4C8>

20
```

Note: instead of sending a argument to fn parameter, we send the argument/ value from send() to the generator fn

----

```
def informToBank():
    a = yield
     b = yield
     yield a + 10
    yield b + 20

msg = informToBank()
next(msg)
print(msg)
print(msg.send(10))

print(msg.send(100))
print(next(msg))
```

How to work with fn, with out sending data (parameter)?

START the generator to send parameter to fn – send()

Note: via send(), we send the value to the yield

```
def informToBank():
    print("Welcome to SBI")
    balance = yield
    print("balance :  ", balance)
    yield balance + 10

deposit = informToBank()
print(deposit) # prints the generator
print(next(deposit)) # comment execute the code to see the error"TypeError: can't send non-None value to a just-started generator" // this is where we started generator
print(deposit.send(5))
```

output

```
<generator object informToBank at 0x00988840>

Welcome to SBI

None

balance :   5

15
```

Note: doubt : how did we get None ?

Ans to remove None do the below

```
print(next(deposit) //don’t do this
next(deposit) // do this (ie just start the generator)
```

Send() accepts only one parameter to fn

Note: via send(), we send the value to the yield

We send only one value to generator using send(). So send a list via send()- see the code below

```
def informToBank():
    print("Welcome to SBI")
    balance1, balance2, balance3 = yield
    print(balance1, balance2, balance3)
    yield balance1 +10, balance2 + 10, balance3 + 10

listOfDeposits = [100, 200,300]
deposit = informToBank()
print(deposit)
print(next(deposit)) # comment execute the code to see the error"TypeError: can't send non-None value to a just-started generator"
print(deposit.send(listOfDeposits))
```

output

```
<generator object informToBank at 0x01827878>
Welcome to SBI
None
100 200 300
(110, 210, 310)
---------------
```

```
def informToBank():
    print("Welcome to SBI")
    balance1, balance2, balance3 = yield
     # print(balance1, balance2, balance3)
     yield balance1 +10, balance2 + 10, balance3 + 10

listOfDeposits = [100, 200,300]
deposit = informToBank()
# print(deposit)
print(deposit.__next__()) # comment execute the code to see the error"TypeError: can't send non-None value to a just-started generator"
b = deposit.send(listOfDeposits)
print(b[0])
print(b[1])
print(b[2])
```

output

```
Welcome to SBI
```

```
None

110

210

310
```

Use send() in class – example pgm

Note: via send(), we send the value to the yield

```
class Test():

    def informToBank(self):
        print("Welcome to SBI")
        a = yield
        for val in a:
            val = val + 100
            print(val)
        yield "Operation completed"

t = Test()
lst1 = [10, 20, 30]
msg = t.informToBank()
next(msg) # Start the generator
```

```
print(msg)
print(msg.send(lst1))
```

output

```
Welcome to SBI
<generator object Test.informToBank at 0x01497878>
110
120
130
Operation completed
---------------------
```

under what circumstances we send object to generator

(to accumulate the results ex election)

Note: via send(), we send the value to the yield

```
def election_pallet_accumulator():
    total = 0
    value =  None
    while True:
        value = yield total
        if value is None:
            break
        total = total + value
# advance until the first "yield"
generator = election_pallet_accumulator()

# from this point on, the generator aggregates values
next (generator)
print (generator.send(1))
print (generator.send(1))
print (generator.send(1))
print (generator.send(1))
#print (generator.send(None))# send None and see the results
------------------
generator = accumulator()

next (generator)
print (generator.send(1))
print (generator.send(10))
print (generator.send(12))
# Calling next(generator) is equivalent to calling generator.send(None)//gives error
next(generator)
```

output

```
1
11
23
   next(generator)
StopIteration

======
print (next(generator))
print (generator.send(1))
print (generator.send(10))
print (generator.send(12))
output
0
1
11
23
----------------------
```

What happens here is the following:

1. When you first call next(generator), the program advances to the first yield statement, and returns the value of total at that point, which is 0. The execution of the generator suspends at this point.

2. When you then call **generator.send(x),** the interpreter takes the argument x and makes it the return value of the last yield statement, which gets assigned to value. The generator then proceeds as usual, until it yields the next value.

When you finally call next(generator), the program treats this as if you're sending **None** to the generator. There is nothing special about **None**, however, this example uses **None** as a special value to ask the generator to stop.

Nested loop and generater object

```
def function():
    for x in range (3):
        yield x **2

def function1():
    for x in range (3):
        yield x **2

# function()
a= function()
# print (a)
# for val in a:
#       print (val)
#
```

```
for item1 in a:
    b = function1()
    for item2 in b:
        print(item1, item2)

# ------------------
# for item1 in range(3):
#     item1 = item1 ** 2
#     for item2 in range(3):
#         item2 = item2 ** 2
#         print(item1, item2)
```

output

```
0 0
0 1
0 4
1 0
1 1
1 4
4 0
4 1
4 4
```

Yield from

Yielding all values from another iterable

Use yield from if you want to yield all values from another iterable:

```
def foob(x):
    yield from range(x * 2) #[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    yield from range (2) #[0, 1]
    # yield from range (5) #[0, 1]

a =  (foob(5))
print(a)
print(list(a)) #it executes until the 'a' becomes None
```

Output

```
<generator object foob at 0x01047958>

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1]
```

Note:

1. if we want to use “yield from”, we have to use list to get all the values

yield from , takes any number of iterables and adds to gether

```
def foob(x,y, z):
   yield from x
   yield from  y
   yield from  z

t1 = (1,2,3,4)
t2 = [10,20,30,40]
t3 = [100, 200, 300, 400]

a = foob(t1, t2, t3)
print(list(a))
for val in a:
   print(val)
```

output

```
[1, 2, 3, 4, 10, 20, 30, 40, 100, 200, 300, 400]
```

Another code / ware house example (Try with dict + Counter)

```
def foob(Trichy, Chennai,Thanjai):
    yield from Trichy
    yield from  Chennai
    yield from  Thanjai

Trichy = (1,2,3,4)
Chennai= [10,20,30,40]
Thanjai = [100, 200, 300, 400]

a = foob(Trichy, Chennai, Thanjai)
print(list(a))
for val in a:
    print(val)
```

output

```
[1, 2, 3, 4, 10, 20, 30, 40, 100, 200, 300, 400]
```

```
def foob(Trichy, Chennai,Thanjai):
    yield from Trichy
    yield from  Chennai
    yield from  Thanjai
```

```
Trichy = ("pen", "Laptop", "Jeans")
Chennai= ["Pencil", "Mouse"]
Thanjai = ["Mouse", "Laptop", "Jeans" ]

a = foob(Trichy, Chennai, Thanjai)
b =(list(a))

from collections import Counter
print(Counter(
```

output

```
Counter({'Laptop': 2, 'Jeans': 2, 'Mouse': 2, 'pen': 1, 'Pencil': 1})
```

*args and yield from

Even we can use *args that take multiple iteratble arguments

```
def foob(*args):
   yield from args

t1 = (1,2,3,4)
t2 = [10,20,30,40]
t3 = {100, 200, 300, 400}
```

```
a = foob(t1, t2, t3)
print(list(a))
for val in a:
   print(val)
```

output

```
[(1, 2, 3, 4), [10, 20, 30, 40], {200, 100, 400, 300}]
```

Yield from and a function()

In mathematics, the Fibonacci numbers, commonly denoted F_n, form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1

```
def fibto(n):
   a, b = 1, 1
   while True:
      if a >= n:
         break
```

```
        yield a
        a, b = b, a + b
def usefib():
    yield from (fibto(10)) #[1, 1, 2, 3, 5, 8]
    yield from (fibto(20)) #[1, 1, 2, 3, 5, 8, 13]

a=list(usefib())
print (a)
```

output

```
[1, 1, 2, 3, 5, 8, 1, 1, 2, 3, 5, 8, 13]
```

Iteration

A generator object supports the iterator protocol. That is, it provides a **next() method (__next__()** in Python 3.x), which is used to step through its execution, and its __iter__ method returns itself. This means that a generator can be used in any language construct which supports generic iterable objects.

The next() function

next(iterator[, default])

Return the next item from the iterator. If default value is given and if the iterator is exhausted, it returns the value that we passed, instead of raising StopIteration

```
def nums():
   yield 1
   yield 2
   yield 5
```

```
generator =  nums()
print(next(generator, None)) #1
print(next(generator, None)) # 2
print(next(generator, None)) # 5
print(next(generator, None)) # None
print(next(generator, "Non")) #Non # user defined value is possible instead of None
```

output

```
1

2

5
None

Non
```

Note: The syntax is next(iterator[, default]). If iterator ends and a default value was passed, it is returned. **If no default was provided, StopIteration is raised.**

To check if gen is created / suspended

```
import inspect
def informToBank():
    yield 10
    yield 20
    yield 30

msg = informToBank()
print(inspect.getgeneratorstate(msg))
print(next(msg))
print(inspect.getgeneratorstate(msg))
print(next(msg))
print(inspect.getgeneratorstate(msg))
print(next(msg))
print(inspect.getgeneratorstate(msg))
```

output

```
GEN_CREATED
10
GEN_SUSPENDED
20
GEN_SUSPENDED
30
GEN_SUSPENDED
```