Programming in Python (CSM210-2C)

Unit 3

Advanced Concept

Iterator, generator, decorators, closure, property

Iterator

Iterators are objects that can be iterated upon. They serve as a common feature of the Python programming language, neatly tucked away for looping and list comprehensions.

Any object that can derive an iterator is known as an iterable

There is a lot of work that goes into constructing an iterator.

For instance, the implementation of each iterator object must consist of **an __iter__() and __next__()** method.

In addition to the prerequisite above, the implementation must also have a way to track the object's internal state and raise a StopIteration exception once no more values can be returned. **These rules are known as the iterator protocol**.

Implementing your own iterator is a drawn-out process, and it is only sometimes necessary.

A simpler alternative is to use a generator object. Generators are a special type of function that use the **yield keyword** to return an iterator that may be iterated over, one value at a time.

The ability to discern the appropriate scenarios to implement an iterator or use a generator will improve your skills as a Python programmer.

Glossary

Term	Definition
Iterable	A Python object which can be looped over or iterated over in a loop. Examples of iterables include lists, sets, tuples, dictionaries, strings, etc.
Iterator	An iterator is an object that can be iterated upon. Thus, iterators contain a countable number of values.
Generator	A special type of function which does not return a single value: it returns an iterator object with a sequence of values.
Lazy Evaluation	An evaluation strategy whereby certain objects are only produced when required. Consequently, certain developer circles also refer to lazy evaluation as “call-by-need.”
Iterator Protocol	A set of rules that must be followed to define an iterator in Python.
next()	A built-in function used to return the next item in an iterator.
iter()	A built-in function used to convert an iterable to an iterator.
yield()	A python keyword similar to the return keyword, except yield returns a generator object instead of a value.

Python Iterators & Iterables

- Iterables are objects capable of returning their members one at a time – they can be iterated over.
- Popular built-in Python data structures such as lists, tuples, and sets qualify as iterables.
- Other data structures like strings and dictionaries are also considered iterables: a string can produce iteration of its characters, and the keys of a dictionary can be iterated upon.
- As a rule of thumb, consider any object that can be iterated over in a for-loop as an iterable.

Exploring Python iterables with examples

Given the definitions, we may conclude that all iterators are also iterable. However, every iterable is not necessarily an iterator. An iterable produces an iterator only once it is iterated on.

To demonstrate this functionality, we will instantiate a list, which is an iterable, and produce an iterator by calling **the iter()** built-in function on the list.

```
list_instance = [1, 2, 3, 4]

print(iter(list_instance))
```

OUTPUT:

```
"""

<list_iterator object at 0x7fd946309e90>

"""
```

- This code creates a list called list_instance with four elements.
- Then, the iter() function is called on list_instance, which returns an iterator object that can be used to iterate over the elements of the list.

- Finally, the print() function is used to display the iterator object, which is represented as <list_iterator object at 0x7fd946309e90> in the output.
- This output shows that the iterator object has been successfully created and is ready to be used to iterate over the elements of the list.

Although the list by itself is not an iterator, calling the iter() function converts it to an iterator and returns the iterator object.

To demonstrate that not all iterables are iterators, we will instantiate the same list object and attempt to call the **next() function**, which is used to return the next item in an iterator.

```
list_instance = [1, 2, 3, 4]

print(next(list_instance))

"""

TypeError                    Traceback (most recent call last)

<ipython-input-2-0cb076ed2d65> in <module>()

    3 print(iter(list_instance))

    4

----> 5 print(next(list_instance))

TypeError: 'list' object is not an iterator

"""
```

- This code creates a list called list_instance with four elements.
- Then, it tries to use the next() function on the list, which raises a TypeError because lists are not iterators.

- To iterate over a list, you need to first convert it to an iterator using the iter() function.

Exploring Python iterators with examples

Thus, if the goal is to iterate on a list, then an iterator object must first be produced. Only then can we manage the iteration through the values of the list.

```
# instantiate a list object

    list_instance = [1, 2, 3, 4]

# convert the list to an iterator

    iterator = iter(list_instance)

# return items one at a time

    print(next(iterator))
    print(next(iterator))
    print(next(iterator))
    print(next(iterator))
```

OUTPUT

```
1
2
3
4
```

This code creates a list object called list_instance with four elements. Then, it converts the list to an iterator using the **iter()** function and assigns it to the variable iterator.

The **next()** function is used to retrieve the next item in the iterator.

The first next() call returns the first item in the iterator, which is the first element of the original list (1).

The second next() call returns the second item in the iterator, which is the second element of the original list (2).

This process continues until all items in the iterator have been returned.

Finally, the print() function is used to display each item returned by the next() function.

The output shows each element of the original list printed on a separate line.

Python automatically produces an iterator object whenever you attempt to loop through an iterable object.

```
list_instance = [1, 2, 3, 4]

for iterator in list_instance:

  print(iterator)

"""

1

2

3

4

"""
```

When the StopIteration exception is caught, then the loop ends.

The values obtained from an iterator can only be retrieved from left to right. Python does not have a previous() function to enable developers to move backward through an iterator.

The lazy nature of iterators

It is possible to define multiple iterators based on the same iterable object. Each iterator will maintain its own state of progress. Thus, by defining multiple iterator instances of an iterable object, it is possible to iterate to the end of one instance while the other instance remains at the beginning.

```
list_instance = [1, 2, 3, 4]

iterator_a = iter(list_instance)

iterator_b = iter(list_instance)

print(f"A: {next(iterator_a)}")

print(f"A: {next(iterator_a)}")

print(f"A: {next(iterator_a)}")

print(f"A: {next(iterator_a)}")

print(f"B: {next(iterator_b)}")

"""

A: 1

A: 2

A: 3

A: 4

B: 1

"""
```

Notice iterator_b prints the first element of the series.

Thus, we **can say iterators have a lazy nature**: when an iterator is created, the elements are not yielded until they are requested. In other words, the elements

of our list instance would only be returned once we explicitly ask them to be **with next(iter(list_instance)).**

However, all of the values from an iterator may be extracted at once by calling a built-in iterable data structure container (i.e., list(), set(), tuple()) on the iterator object to force the iterator to generate all its elements at once.

```
# instantiate iterable
    list_instance = [1, 2, 3, 4]
# produce an iterator from an iterable
    iterator = iter(list_instance)
    print(list(iterator))
"""
[1, 2, 3, 4]
"""
```

This code creates a list called list_instance with the values 1, 2, 3, and 4. Then, it creates an iterator object called iterator using the iter() function and passing in the list_instance object.

Finally, it prints the result of calling the list() function on the iterator object, which converts the iterator back into a list and prints it. The output is the same as the original list_instance object.

In Python, an iterable is an object that can return an iterator, which is an object that can be iterated (looped) upon. The iter() function returns an iterator object from an iterable. The list() function can be used to convert an iterator object back into a list.

It’s not recommended to perform this action, especially when the elements the iterator returns are large since this will take a long time to process.

Whenever a large data file swamps your machine's memory, or you have a function that requires its internal state to be maintained upon each call but creating an iterator does not make sense given the circumstances, a better alternative is to use a generator object.

MCQ on Iterators

1. Which method must be implemented by a class to make it an iterator?

a. __iter__()

b. __next__()

c. Both __iter__() and __next__()

d. __getitem__()

2. What is the purpose of the __iter__() method in an iterator?

a. To return the next value in the sequence

b. To return the iterator object itself

c. To raise a StopIteration exception

d. To initialize the iterator

3. What does the __next__() method do in an iterator?

a. Initialize the iterator

b. Return the iterator object itself

c. Raise a StopIteration exception

d. Return the next value in the sequence

4. Which exception signals the end of an iteration in a custom iterator?

a. ValueError b. IndexError c. StopIteration d. KeyError

5. What is the iterable protocol in Python?

a. Implementing __iter__() and __next__() methods

b. Implementing __init__() and __getitem__() methods

c. Implementing __len__() and __getitem__() methods

d. Implementing __call__() and __An__iter() methods

6. Which built-in function returns an iterator from an iterable?

a. next() b. iter() c. list() d. str()

7. What is the output of the following code?

my_list = [1, 2, 3, 4]

my_iter = iter(my_list)

print(next(my_iter))

print(next(my_iter))

a. 1 2 b. 2 1 c. 1 3 d. 3 4

8. What happens when next() **is called on an iterator that has reached the end of its sequence?**

a. It raises a ValueError b. It raises an IndexError

c. It returns None d. It raises a StopIteration exception

1. **Answer:** c. Both __iter__() and __next__()
2. **Answer:** b. To return the iterator object itself
3. **Answer:** d. Return the next value in the sequence
4. **Answer:** c. StopIteration
5. **Answer:** a. Implementing __iter__() and __next__() methods
6. **Answer:** b. iter()
7. **Answer:** a. 1 2
8. **Answer:** d. It raises a StopIteration exception

What is Lazy Evaluation/ call-by-need?

Lazy Evaluation is a programming concept often used in functional languages, where the evaluation of expressions is delayed until their results are actually needed. This approach helps optimize the execution of programs by reducing the computational load, enabling efficient processing and allocation of resources. In the context of data processing and analytics, Lazy Evaluation techniques can be applied to improve performance and scalability of data pipeline operations.

How it works

- Lazy evaluation avoids unnecessary work by only evaluating expressions when their results are needed.
- It can also save memory and enable infinite data structures.
- The result of the expression is cached for future use.

Benefits

- Lazy evaluation can improve the performance and scalability of data pipelines.
- It can also help define control flow structures as abstractions.
- It can help define partly-defined data structures.

Examples

- In Python, lazy evaluation can be used with functions that produce collections of objects.

- For example, you can filter names first and then convert them to uppercase.

Programming languages that support lazy evaluation Haskell, Scala, Clojure, and Python.

- **Strict evaluation/ Eager evaluation**

The opposite of lazy evaluation, where every expression is evaluated, regardless of whether it's needed

The opposite of lazy evaluation, where every expression is evaluated, regardless of whether it's needed

Generator

- The most expedient alternative to implementing an iterator is to use a generator. Although generators may look like ordinary Python functions, they are different.
- For starters, a generator object does not return items. Instead, it uses **the yield keyword** to generate items on the fly. Thus, we can say a generator is a special kind of function that leverages lazy evaluation.
- **Generators do not store their contents in memory** as you would expect a typical iterable to do. For example, if the goal were to find all of the factors for a positive integer, we would typically implement a traditional function as follows:

```
def factors(n):
  factor_list = []
  for val in range(1, n+1):
      if n % val == 0:
          factor_list.append(val)
  return factor_list
print(factors(20))
"""
[1, 2, 4, 5, 10, 20]
"""
```

The code above returns the entire list of factors.

However, notice the difference when a generator is used instead of a traditional Python function:

```
def factors(n):
  for val in range(1, n+1):
    if n % val == 0:
      yield val
print(factors(20))
```

""" <generator object factors at 0x7fd938271350> """

The print statement then calls the factors function with an argument of 20 and prints the resulting generator object.

The output of the print statement is the memory location of the generator object, indicated by the hexadecimal number.

Since we **used the yield keyword instead of return**, the function is not exited after the run.

In essence, we told Python to create a generator object instead of a traditional function, which enables the state of the generator object to be tracked.

Consequently, it is possible to **call the next() function** on the lazy iterator to show the elements of the series one at a time.

```
print(next(factors_of_20))
"""          1          """
```

Another way to create a generator is with a **generator comprehension.**

Generator expressions adopt a similar syntax to that of a list comprehension, except it uses **rounded brackets** instead of squared.

```
print((val for val in range(1, 20+1) if n % val == 0))
"""
<generator object <genexpr> at 0x7fd940c31e50>
"""
```

Python's yield Keyword

- The **yield** keyword controls the flow of a generator function. Instead of exiting the function as seen when **return** is used, the **yield** keyword returns the function but remembers the state of its local variables.
- The generator returned from the **yield** call can be assigned to a variable and iterated upon with the next() keyword – this will execute the function up to the first **yield** keyword it encounters. Once the **yield** keyword is hit, the execution of the function is

suspended. When this occurs, the function's state is saved. Thus, it is possible for us to resume the function execution at our own will.

The function will continue from the call to **yield**. For example:

```
def yield_multiple_statments():
  yield "This is the first statment"
  yield "This is the second statement"
  yield "This is the third statement"
  yield "This is the last statement. Don't call next again!"

example = yield_multiple_statments()
print(next(example))
print(next(example))
print(next(example))
print(next(example))
print(next(example))
"""
This is the first statment
This is the second statement
This is the third statement
This is the last statement. Don't call next again or else!
---------------------------------------------------------------
StopIteration                Traceback (most recent call last)
<ipython-input-25-4aaf9c871f91> in <module>()
    11 print(next(example))
```

```
    12 print(next(example))
---> 13 print(next(example))
StopIteration:
"""
```

Summary:

To recap, iterators are objects that can be iterated on, and generators are special functions that leverage lazy evaluation.

Implementing your own iterator means you must create an __iter__() and __next__() method, whereas a generator can be implemented using the yield keyword in a Python function or comprehension.

MCQs on Generators in Python

1. Which keyword is used to create a generator in Python?

a. yield b. return c. generate d. produce

2. What is the primary difference between a generator and a regular function?

a. Generators use the return keyword, while regular functions use the yield keyword.

b. Generators return a single value, while regular functions can return multiple values.

c. Generators can only be called once, while regular functions can be called multiple times.

d. Generators use the yield keyword, while regular functions use the return keyword.

3. What will be the output of the following code?

python

```
def simple_generator():
    yield 1
    yield 2
    yield 3
gen = simple_generator()
print(next(gen))
print(next(gen))
```

a. 1 2 b. 1 3 c. 2 3 d. 1 1

4. How can you convert a generator into a list in Python? a. Using the tolist() method

b. Using the list() function

c. Using the array() function

d. Using the toarray() method

5. Which of the following statements is true about generators?

a. Generators are memory-efficient because they produce items one at a time.

b. Generators can store all the generated values in memory at once.

c. Generators can only be used for integer values.

d. Generators cannot be used in a for loop.

6. What will happen if next() **is called on a generator that has no more items to yield?**

a. It will return None.

b. It will raise a ValueError.

c. It will raise a StopIteration exception.

d. It will go back to the first yield statement.

Question Answers on Generators in Python

Q1: What is a generator in Python, and how does it differ from a regular function?

Answer: A generator in Python is a special type of iterator that is defined using the yield keyword. Unlike regular functions that use the return keyword to return a value and terminate, generators use yield to produce a value and pause execution, which can be resumed later. This allows generators to produce a sequence of values over time, making them memory-efficient, as they don't need to store the entire sequence in memory.

Q2: Explain the purpose of the yield **keyword in Python.**

Answer: The yield keyword is used to create a generator in Python. When a generator function calls yield, it produces a value and suspends the function's execution. The function retains its state, so when it is resumed, it continues execution from the point where it was suspended. This allows the generator to produce a sequence of values one at a time, making it suitable for iterating over large datasets without consuming a lot of memory.

Q3: How do you create a generator expression, and how does it differ from a list comprehension?

Answer: A generator expression is similar to a list comprehension but uses parentheses instead of square brackets. It creates a generator

object that produces values lazily, one at a time, without storing them all in memory at once. Here is an example:

python

```
# List comprehension
squares_list = [x * x for x in range(1, 6)]

# Generator expression
squares_gen = (x * x for x in range(1, 6))
```

While a list comprehension creates a list containing all the generated values, a generator expression creates a generator that produces each value on demand, making it more memory-efficient.

Q4: What is the StopIteration **exception, and when is it raised in the context of generators?**

Answer: The StopIteration exception is raised when a generator has no more items to yield. This exception signals the end of iteration, and it is raised automatically by the generator when it reaches the end of its sequence. When using a generator in a for loop or with the next() function, the StopIteration exception is caught internally to terminate the loop or iteration gracefully.

Q5: Provide an example of a generator function that yields the Fibonacci sequence up to n **numbers.**

Answer: Here is an example of a generator function that yields the Fibonacci sequence up to n numbers:

python

```
def fibonacci_generator(n):
    a, b = 0, 1
    count = 0
    while count < n:
        yield a
        a, b = b, a + b
        count += 1

# Using the Fibonacci generator
fib_gen = fibonacci_generator(10)

for value in fib_gen:
    print(value)
```

This generator function initializes the first two numbers of the Fibonacci sequence (a and b) and uses a while loop to yield each number in the sequence up to n numbers.

Answer: a. yield

Answer: d. Generators use the yield keyword, while regular functions use the return keyword.

Answer: a. 1 2

Answer: b. Using the list() function

Answer: a. Generators are memory-efficient because they produce items one at a time.

Answer: c. It will raise a StopIteration exception.