ITERATORS AND GENERATORS

CS 3080: Python Programming

UCCS University of Colorado Colorado Springs

Iterators

- We use **for** statement for looping over a **list**.
- If we use it with a **string**, it loops over its characters.
- If we use it with a **dictionary**, it loops over its keys.
- So there are many types of objects which can be used with a for loop.
- These are called **iterable objects**.

Iteration protocol

- If an object is iterable, it can be passed to the built-in function **iter** which takes an iterable object and returns an iterator.
- In Python, iterable means an object can be used in iteration.

- **An iterator is a value producer that yields values. Built-in function next() is used to obtain the next value from an iterator.**
- **An iterator retains its internal state. It knows which values have been obtained, so when you call next(), it knows what value to return next.**

Creating an iter object

- To make a custom class be iterable, it has to implement the __**iter**__ and __**next**__ methods.
  - *The __**iter**__ method is what makes an object iterable. The return value of __**iter**__ is the class itself.*
  - *The __**next**__ method is what the class should return at each iteration. It raises **StopIteration** when there are no more elements.*

Creating an iter object

```
class MyRange:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __iter__(self):
        return self

    def __next__(self):
        if self.i < self.n:
            result = self.i
            self.i += 1
            return result
        else:
            raise StopIteration()
```

Generators

- Generators simplifies creation of iterators. A generator is a function that produces a sequence of results instead of a single value.
- When a generator function is called, it returns a generator object without executing the function. When **next() method is called for the first time**, the function starts executing until it reaches yield statement. The yielded value is returned.
- Each time the **yield** statement is executed the function generates a new value.

```
def myRange(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

Generators

- Generators simplifies creation of iterators. A generator is a function that produces a sequence of results instead of a single value.
- When a generator function is called, it returns a generator object without executing the function. When **next() method is called for the first time**, the function starts executing until it reaches yield statement. The yielded value is returned.
- Each time the **yield** statement is executed the function generates a new value.

```
def myRange(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

- **Generator functions look like regular functions, but use yield instead of return**
- **yield indicates a value is sent back, but doesn't exit. Instead, the state of the function is remembered**

Generators

- Generators simplifies creation of iterators. A generator is a function that produces a sequence of results instead of a single value.
- When a generator function is called, it returns a generator object without executing the function. When **next() method is called for the first time**, the function starts executing until it reaches yield statement. The yielded value is returned.
- Each time the **yield** statement is executed the function generates a new value.

```
def myRange(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

- **When next() is called on a generator (explicitly or implicitly in a for loop) but not first time, the previous state is resumed, i.e., function execution resumes after yield**

Generators expressions

- List comprehension is an easy way to define and create list in Python.
- Generator expressions allow the creation of a generator on-the-fly without a yield keyword.
- They look like list comprehensions, but returns a generator instead of a list.
- In terms of syntax, the only difference is that you use parenthesis instead of square brackets.

Generators expressions

- The type of data returned by list comprehensions and generator expressions differs.

```
list_comp = [x ** 2 for x in range(10) if x % 2 == 0]


gen_exp = (x ** 2 for x in range(10) if x % 2 == 0)


print(list_comp)

# [0, 4, 16, 36, 64]

print(gen_exp)

# <generator object <genexpr> at 0x7f600131c410>
```

Generators expressions

- The main advantage of generator over a list is that it take much less memory.
- The generator yields one item at a time—thus it is more memory efficient than a list.

```
from sys import getsizeof

my_comp = [x * 5 for x in range(1000)]
my_gen = (x * 5 for x in range(1000))

print(getsizeof(my_comp))
# 9024 bytes
print(getsizeof(my_gen))
# 120 bytes
```