Iterators and Generators

Definitions, Implementations and Examples

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#python-advanced

Table of Contents

What are Iterators?

`__iter__()` and `__next__()`

What are Iterators?

- Iterator is simply an **object** that can be **iterated** upon
- An object which will return data, **one element** at a time
- Iterator object must implement two methods, `__iter__()` and `__next__()` (iterator protocol)
- An object is called iterable if we can **get an iterator** from it
- Such are: **list**, **tuple**, **string**, etc...

Example: Iterator

- The **iter()** function (which calls the **__iter__()** method) returns an iterator from an iterable

```
my_list = [4, 7, 0, 3]
# get an iterator using iter()
my_iter = iter(my_list)
print(next(my_iter))        # 4
print(next(my_iter))        # 7
print(my_iter.__next__())   # 0
print(my_iter.__next__())   # 3
next(my_iter)               # Error
```

For Loops and Iterators

- The for loop can iterate automatically through the list
- The for loop can iterate over any iterable
- A for loop is implemented as:

```
iter_obj = iter(iterable)
while True:
    try:
        element = next(iter_obj) # get the next item
        # do something with element
    except StopIteration:
        # if StopIteration is raised, break from loop
        break
```

Explanation

- The for loop creates an iterator object (`iter_obj`) by calling `iter()` on the iterable
- Inside the loop, it calls `next()` to get the next element and executes the body of the for loop with this value
- After all the items exhaust, `StopIteration` is raised, which is internally caught, and the loop ends

Problem: Custom Range

- Since iterators are implemented using classes, create a **class** called **custom_range** that receives **start** and **end** upon initialization
- Implement the **__iter__** and **__next__** methods, so the iterator returns the numbers from the start to the end (inclusive)

```
one_to_ten = custom_range(1, 10)
for num in one_to_ten:
    print(num)
```

```
1
2
…
10
```

Solution: Custom Range

```
class custom_range:
    def __init__(self, start, end):
        self.i = start
        self.n = end

    def __iter__(self):
        return self

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

Problem: Reverse Iter

- Create a class called **reverse_iter** which should receive an iterable upon initialization
- Implement the **__iter__** and **__next__** methods, so the iterator returns the items of the iterable in **reversed** order

```
reversed_list = reverse_iter([1, 2, 3, 4])
for item in reversed_list:
    print(item)
```

```
4
3
2
1
```

Solution: Reverse Iter

```
class reverse_iter:
    def __init__(self, iterable):
        self.iterable = iterable
        self.i = len(self.iterable) - 1

    def __iter__(self):
        return self

    def __next__(self):
        # TODO: Implement
```

What are Generators?

Way of Creating Iterators

What are Generators?

- Generators are a simple way of **creating iterators**
- A generator is a **function** that returns an object (iterator)
- This iterator can later be iterated over (one value at a time)

```
yield
```

Example: Generators

```
def first_n(n):
    num = 0
    while num < n:
        yield num
        num += 1

sum_first_n =
sum(first_n(5))
print(sum_first_n)
```

The `yield` Statement

- If a function contains at least one **yield** statement, it becomes a **generator** function
- Both **yield** and **return** will return a value from a function
- The difference between yield and return is that the return statement **terminates** a function entirely
- Yield, however, **pauses** the function **saving** all its states, and later **continues** from there on successive calls

Generators vs Normal Functions

- Generator function contains one or more **yield** statements
- It returns an **iterator** but does not start execution immediately
- Methods like `__iter__()` and `__next__()` are implemented automatically
- Once the function yields, the function is **paused**
- When the function terminates, `StopIteration` is raised automatically on further calls

Example: Generator Function

```
def my_gen():
    n = 1
    print('This is printed first')
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n
```

The value of "n" is remembered between calls

Generator Expression

- Generators can be easily created using **generator expressions**
- Same as lambda function creates an anonymous function, **generator expression** creates an anonymous **generator function**
- The syntax for generator expression is similar to that of a list comprehension
- The difference between them is that generator expression produces **one item at a time**

Example: Generator Expression

```
# Initialize the list
my_list = [1, 3, 6, 10]

# square each term using list comprehension
# Output: [1, 9, 36, 100]
print([x**2 for x in my_list])

# the same thing can be done using the generator expression
# Output: <generator object <genexpr> at 0x0000000002EBDAF8>
print((x**2 for x in my_list))
```

Problem: Squares

- Create a generator function called **squares** that should receive a number **n**
- It should generate the squares of all numbers from **1 to n** (inclusive)

```
print(list(squares(5)))
```

```
[1, 4, 9, 16, 25]
```

```
def squares(n):
    i = 1
    while i <= n:
        yield i * i
        i += 1
```

Problem: Generator Range

- Create a generator function called **genrange** that receives a **start** and an **end**
- It should generate all the numbers from the start to the end (inclusive)

```
print(list(genrange(1, 10)))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

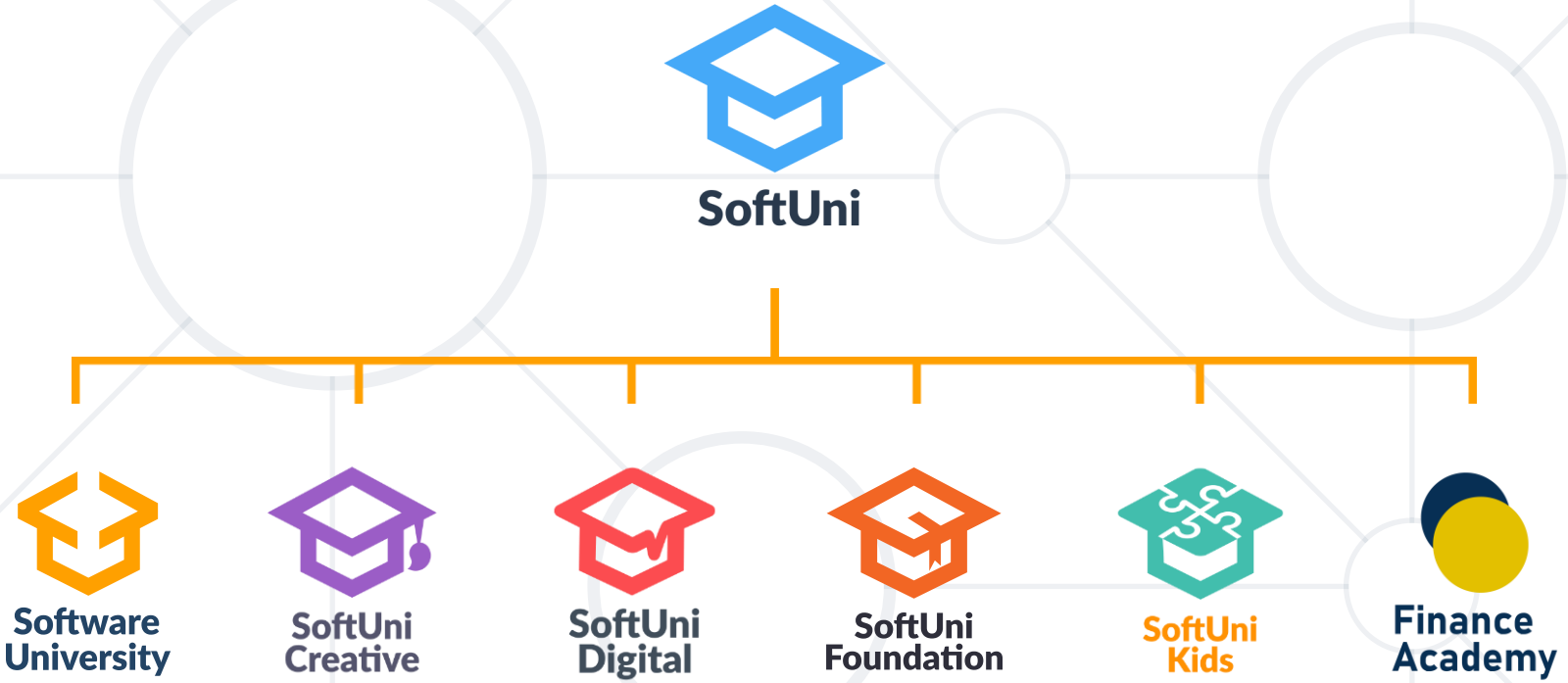
Solution: Generator Range

```
def genrange(start, end):
    i = start
    n = end
    while i <= n:
        yield i
        i += 1
```

Summary

- Iterator is an object that can be iterated upon
- Functions that return iterators are called generators
- Generators - `yield`, Functions - `return`
- Generator expression is like a list comprehension

Questions?



SoftUni Diamond Partners

Software University

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License

- This course (slides, examples, demos, exercises, homework, documents, videos, and other assets) is **copyrighted content**
- Unauthorized copy, reproduction, or use is illegal
- © SoftUni – https://about.softuni.bg
- © Software University – https://softuni.bg