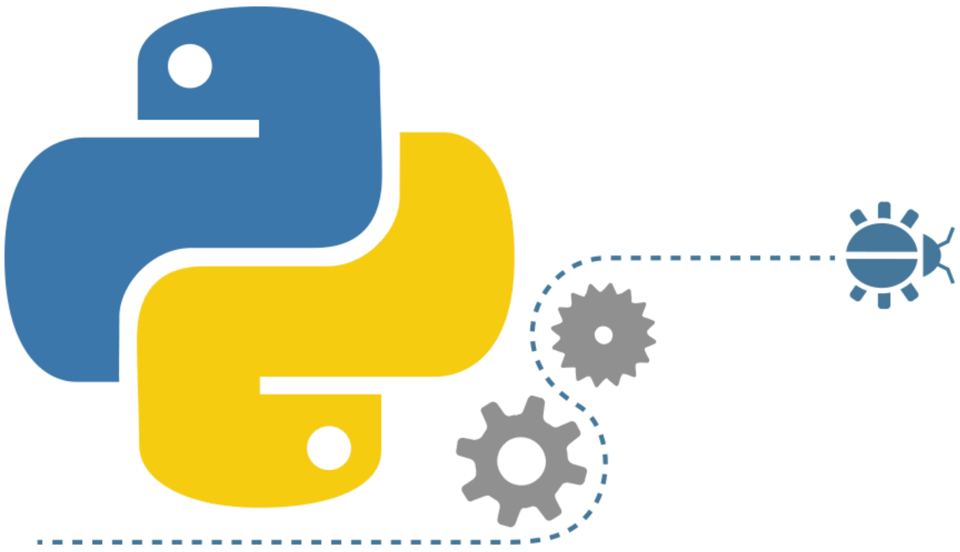
# Python Basics: Iteration, Iterables, Iterators, and Looping

Ventsislav Yordanov Follow
Nov 22, 2018 · 9 min read



Source: https://www.quora.com/Can-you-suggest-some-good-books-websites-for-learning-Python-for-a-layman

After reading this blog post, you'll know:

- How the **iteration** in Python works **under the hood**
- What are **iterables** and **iterators** and how to create them
- What is the **iterator protocol**
- What is a **lazy evaluation**

- What are the **generator functions** and **generator expressions**

## Python's for loop

Python doesn't have traditional `for` loops. Let's see a pseudocode of how a traditional for loop looks in many other programming languages.

```
for (initializer; condition; iterator)
    body
```

For loop pseudo code.txt hosted with ♡ by GitHub view raw

A Pseudocode of for loop

- The **initializer section** is executed only once, before entering the loop.
- The **condition section** must be a **boolean expression**. If this expression evaluates to True, the next loop iteration is executed.
- The **iterator section** defines what happens after each iteration.

Now, let's see how a traditional for loop can be written in JavaScript.

```javascript
let numbers = [10, 12, 15, 18, 20];
for (let i = 0; i < numbers.length; i += 1) {
    console.log(numbers[i])
}
```

Simple For Loop in JavaScript.js hosted with ♡ by GitHub view raw

Simple For Loop in JavaScript

Output:

```
10
12
15
18
20
```

Many of the other programming languages have this kind of for loop, but Python doesn't have it. However, **Python** has something called `for` **loop**, but it **works like a foreach loop**.

```python
numbers = [10, 12, 15, 18, 20]
for number in numbers:
    print(number)
```

Simple For Loop in Python.py hosted with ♡ by GitHub view raw

Simple For Loop in Python

Output:

```
10
12
15
18
20
```

From the example above, we can see that in Python's `for` loops we don't have any of the sections we've seen previously. There is no initializing, condition or iterator section.

## Iterables

An iterable is an object capable of **returning** its **members one by one**. Said in other words, an iterable is anything that you can loop over with a `for` loop in Python.

### Sequences

Sequences are a very common **type** of **iterable**. Some examples for built-in sequence types are **lists**, **strings**, and **tuples**.

```python
numbers = [10, 12, 15, 18, 20]
fruits = ("apple", "pineapple", "blueberry")
message = "I love Python ♡"
```

Lists, Tuples, and Strings in Python.py hosted with ♡ by GitHub view raw

Lists, Tuples, and Strings in Python

They support efficient **element access** using integer **indices** via the `__getitem()__` special method (indexing) and define a `__length()__` method that returns the **length** of the sequence.

```python
# Element access using integer indices
print(numbers[0])
print(fruits[2])
print(message[-2])
```

Element access using indices on sequences.py hosted with ♡ by GitHub view raw

Element access using indices.

Output:

```
10
blueberry
♥
```

Also, we can use the **slicing** technique on them. If you don't know what is slicing, you can check one of my previous articles when I explain it. The explanation for the slicing technique is in the "Subsetting Lists" section.

```python
# Slicing the sequences
print(numbers[:2])
print(fruits[1:])
print(message[2:])
```

Slicing Sequences.py hosted with ♡ by GitHub view raw

Slicing Sequences.

Output:

```
[10, 12]
('pineapple', 'blueberry')
love Python ♡
```

### Other Iterables

Many things in Python are iterables, but not all of them are sequences. **Dictionaries**, **file objects**, **sets**, and **generators** are all iterables, but none of them is a sequence.

```python
my_set = {2, 3, 5}
my_dict = {"name": "Ventsislav", "age": 24}
my_file = open("file_name.txt")
squares = (n**2 for n in my_set)
```

Sets, Dictionaries, Files, and Generators.py hosted with ♡ by GitHub view raw

Sets, Dictionaries, Files, and Generators.

## Python's for loops don't use indices

Let's think about how we can loop over an iterable without using a `for` loop in Python. Some of us may think that we can use a `while` loop and generate indices to achieve this.

```python
index = 0
numbers = [1, 2, 3, 4, 5]
while index < len(numbers):
    print(numbers[index])
    index += 1
```

Iterate over sequences without using a for loop.py hosted with ♡ by GitHub view raw

Output:

```
1
2
3
4
5
```

It seems that this approach works very well for lists and other sequence objects. What about the **non-sequence** objects? They **don't support indexing**, so this approach will not work for them.

```python
index = 0
```

```
numbers = {1, 2, 3, 4, 5}
while index < len(numbers):
    print(numbers[index])
    index += 1
```

Iterate over set without using a for loop error.py hosted with ♡ by GitHub view raw

Output:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-22-af1fab82d68f> in <module>()
      2 numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
      3 while index < len(numbers):
----> 4     print(numbers[index])
      5     index += 1

TypeError: 'set' object does not support indexing
```

Hmmm, but how the Python's for loop works on these iterables then? We can see that it works with sets.

```
numbers = {1, 2, 3, 4, 5}
for number in numbers:
    print(number)
```

Iterate over set.py hosted with ♡ by GitHub view raw

Iterate over a set.

Output:

```
1
2
3
4
5
```

## Iterators

An iterator is an object representing a **stream of data**. You can create an iterator object by applying the `iter()` built-in function to an **iterable**.

```python
numbers = [10, 12, 15, 18, 20]
fruits = ("apple", "pineapple", "blueberry")
message = "I love Python ♡"

print(iter(numbers))
print(iter(fruits))
print(iter(message))
```

Creating an iterators.py hosted with ♡ by GitHub view raw

Output:

```
<list_iterator object at 0x000001DBCEC33B70>
<tuple_iterator object at 0x000001DBCEC33B00>
<str_iterator object at 0x000001DBCEC33C18>
```

You can use an iterator to manually **loop over** the **iterable** it came from. A repeated passing of iterator to the built-in function `next()` returns **successive items** in the **stream**. Once, when you consumed an item from an iterator, it's gone. When no more data are available a `StopIteration` exception is raised.

```python
values = [10, 20, 30]
iterator = iter(values)
print(next(iterator))
print(next(iterator))
print(next(iterator))
print(next(iterator))
```

Simple iterator.py hosted with ♡ by GitHub view raw

Output:

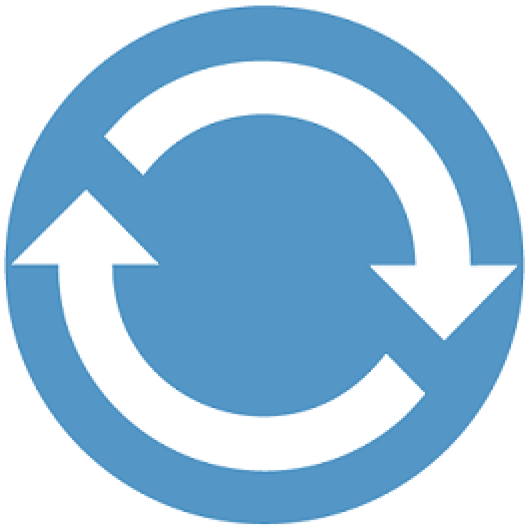
```
10
20
30
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-14-fd36f9d8809f> in <module>()
      4 print(next(iterator))
      5 print(next(iterator))
----> 6 print(next(iterator))

StopIteration:
```

Under the hood, Python’s `for` loop is using iterators.

## Understanding how Python’s for loop works



Source: https://www.incredible-web.com/blog/performance-of-for-loops-with-javascript/

Now, we know what the iterables and iterators are and how to use them. We can try to define a function that loops through an iterable without using a `for` loop.

To achieve this, we need to:

- **Create an iterator** from the given iterable
- Repedeatly **get the next item** from the iterator
- **Execute** the wanted **action**
- **Stop the looping**, if we got a **StopIteration exception** when we’re trying to get the next item

```python
def custom_for_loop(iterable, action_to_do):
    iterator = iter(iterable)
    done_looping = False
    while not done_looping:
        try:
            item = next(iterator)
        except StopIteration:
            done_looping = True
        else:
            action_to_do(item)
```

Custom For Loop in Python.py hosted with ♡ by GitHub view raw

Custom For Loop in Python. Source: https://opensource.com/article/18/3/loop-better-deeper-look-iteration-python

Let's try to use this function with a **set** of numbers and the `print` built-in function.

```python
numbers = {1, 2, 3, 4, 5}
custom_for_loop(numbers, print)
```

Custom For Loop in Python Usage.py hosted with ♡ by GitHub view raw

Custom For Loop in Python - Usage.

Output:

```
1
2
3
4
5
```

We can see that the function we've defined works very well with sets, which are not sequences. This time we can pass **any iterable** and it will work. Under the hood, all forms of **looping over iterables** in Python is working this way.

## Iterator Protocol

The iterator objects are required to support the following **two methods**, which together form the **iterator protocol**:

- iterator.__iter__()
  Return the **iterator object itself**. This is required to allow both containers (also called collections) and iterators to be used with the `for` and `in` statements.

- iterator.__next__()
  Return the **next item** from the container. If there are no more items, raise the **StopIteration** exception.

From the methods descriptions above, we see that we can loop over an iterator. So, the iterators are also iterables.

Remember that when we apply the `iter()` function to an iterable we get an iterator. If we call the `iter()` function **on an iterator** it will always **give us itself** back.

```python
numbers = [100, 200, 300]
iterator1 = iter(numbers)
iterator2 = iter(iterator1)

# Check if they are the same object
print(iterator1 is iterator2)

for number in iterator1:
    print(number)
```

Playing with iterators.py hosted with ♡ by GitHub view raw

Output:

```
True
100
200
300
```

## Additional Notes on Iterators

This may sound a little bit confusing. However, don't worry if you don't understand all things of the first time. Let's recap!

- An **iterable** is something you can **loop over**.

- An **iterator** is an object representing a **stream of data**. It does the **iterating** over an iterable.

Additionally, in Python, the **iterators are also iterables** which act as their **own iterators**.

However, the difference is that iterators don't have some of the features that some iterables have. They **don't have length** and **can't be indexed**.

Examples

```python
numbers = [100, 200, 300]
iterator = iter(numbers)
print(len(iterator))
```

Iterators no length error.py hosted with ♡ by GitHub view raw

Iterators No Length Error

Output:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-778b5f9befc3> in <module>()
      1 numbers = [100, 200, 300]
      2 iterator = iter(numbers)
----> 3 print(len(iterator))

TypeError: object of type 'list_iterator' has no len()
```

```python
numbers = [100, 200, 300]
iterator = iter(numbers)
print(iterator[0])
```

Iterators indexing error.py hosted with ♡ by GitHub view raw

Indexing Iterators Error

Output:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-64c378cb8a99> in <module>()
      1 numbers = [100, 200, 300]
      2 iterator = iter(numbers)
----> 3 print(iterator[0])

TypeError: 'list_iterator' object is not subscriptable
```

## Iterators are lazy

Source: https://giphy.com/gifs/animal-lazy-spirit-gdUxfKtxSxqtq

> *Iterators allow us to both work with and create* ***lazy iterables*** *that don't do any work until we ask them for their next item.*

Source: https://opensource.com/article/18/3/loop-better-deeper-look-iteration-python

Because of their laziness, the iterators can help us to deal with **infinitely long iterables**. In some cases, we can't even store all the information in the memory, so we can use an iterator which can give us the next item every time we ask it. Iterators can **save us** a lot of **memory** and **CPU time**.

This approach is called **lazy evaluation**.

## Iterators are everywhere

We have seen some examples with iterators. Moreover, Python has many built-in classes that are iterators. For example, an `enumerate` and `reversed` objects are iterators.

### Enumerate Example

```python
fruits = ("apple", "pineapple", "blueberry")
iterator = enumerate(fruits)
print(type(iterator))
print(next(iterator))
```

Enumerate example.py hosted with ♡ by GitHub view raw

Output:

```
<class 'enumerate'>
(0, 'apple')
```

### Reversed Example

```python
fruits = ("apple", "pineapple", "blueberry")
iterator = reversed(fruits)
print(type(iterator))
print(next(iterator))
```

Reversed example.py hosted with ♡ by GitHub view raw

Output:

```
<class 'reversed'>
blueberry
```

The Python's `zip`, `map` and `filer` objects are also iterators.

## Zip Example

```python
numbers = [1, 2, 3]
squares = [1, 4, 9]
iterator = zip(numbers, squares)
print(type(iterator))
print(next(iterator))
print(next(iterator))
```

zip example.py hosted with ♡ by GitHub view raw

Output:

```
<class 'zip'>
(1, 1)
(2, 4)
```

## Map Example

```python
numbers = [1, 2, 3, 4, 5]
squared = map(lambda x: x**2, numbers)
print(type(squared))
print(next(squared))
print(next(squared))
```

map example.py hosted with ♡ by GitHub view raw

Output:

```
<class 'map'>
1
4
```

## Filter Example

```python
numbers = [-1, -2, 3, -4, 5]
positive = filter(lambda x: x > 0, numbers)
print(type(positive))
print(next(positive))
```

filter example.py hosted with ♡ by GitHub view raw

Output:

```
<class 'filter'>
3
```

Moreover, the **file objects** in Python are also iterators.

```python
file = open("example.txt")
print(type(file))
print(next(file))
print(next(file))
print(next(file))
file.close()
```

files example.py hosted with ♡ by GitHub view raw

Output:

```
<class '_io.TextIOWrapper'>
This is the first line.

This is the second line.

This is the third line.
```

We can also iterate over **key-value pairs** of a Python **dictionary** using the `items()` method.

```python
my_dict = {"name": "Ventsislav", "age": 24}
iterator = my_dict.items()
print(type(iterator))
for key, item in iterator:
    print(key, item)
```

Iterating over dictionary.py hosted with ♡ by GitHub view raw

Output:

```
<class 'dict_items'>
name Ventsislav
age 24
```

Many people use Python to solve **Data Science** problems. In some cases, the data you work with can be very large. In this cases, we can't load all the data in the memory.

The solution is to **load** the **data in chunks**, then perform the desired operation/s on each chunk, discard the chunk and load the next chunk of data. Said in other words we need to create an iterator. We can achieve this by using the `read_csv` function in **pandas**. We just need to specify the **chunksize**.

## Large DataSets Example

In this example, we'll see the idea with a small dataset called "iris species", but the same concept will work with very large datasets, too. I've changed the column names, you can find my version here.

```python
import pandas as pd

# Initialize an empty dictionary
counts_dict = {}

# Iterate over the file chunk by chunk
for chunk in pd.read_csv("iris.csv", chunksize = 10):
    # Iterate over the "species" column in DataFrame
    for entry in chunk["species"]:
        if entry in counts_dict.keys():
            counts_dict[entry] += 1
        else:
            counts_dict[entry] = 1

# Print the populated dictionary
print(counts_dict)
```

Loading a CSV file in chunks.py hosted with ♡ by GitHub view raw

Output:

```
{'Iris-setosa': 50, 'Iris-versicolor': 50, 'Iris-virginica': 50}
```

There are a lot of iterator objects in the Python standard library and in third-party libraries.

## Creating a custom iterator with defining a Class

In some cases, we may want to create a **custom iterator**. We can do that by defining a class that has **__init__**, **__next__**, and **__iter__** methods.

Let's try to create a custom iterator class that generate numbers between min value and max value.

```python
class generate_numbers:
    def __init__(self, min_value, max_value):
        self.current = min_value
        self.high = max_value

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

numbers = generate_numbers(40, 50)
print(type(numbers))
print(next(numbers))
print(next(numbers))
print(next(numbers))
```

custom iterator example.py hosted with ♡ by GitHub view raw

Output:

```
<class '__main__.generate_numbers'>
40
41
42
```

We can see that this works. However, it is much easier to use a **generator function** or **generator expression** to create a custom iterator.

## Generator Functions and Generator Expressions

Usually, we use a **generator function** or **generator expression** when we want to create a custom iterator. They are **simpler** to use and need **less code** to achieve the same result.

### Generator Functions

Let's see what is a generator function from the Python docs.

> *A function which returns a generator iterator. It looks like a normal function except that it contains `yield` **expressions** for producing a **series of values** usable in a for-loop or that can be retrieved **one at a time** with the `next()` **function**.*

Source: https://docs.python.org/3.7/glossary.html#term-generator

Now, we can try to re-create our custom iterator using a generator function.

```python
def generate_numbers(min_value, max_value):
    while min_value < max_value:
        yield min_value
        min_value += 1

numbers = generate_numbers(10, 20)
print(type(numbers))
print(next(numbers))
print(next(numbers))
print(next(numbers))
```

Generator Function Example.py hosted with ♡ by GitHub view raw

Output:

```
<class 'generator'>
10
11
12
```

The **yield expression** is the thing that separates a **generation function** from a normal function. This expression is helping us to use the iterator's **laziness**.

> *Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the* generator iterator *resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).*

Source: https://docs.python.org/3.7/glossary.html#term-generator-iterator

## Generator Expressions

The generator expressions are **very similar to** the **list comprehensions**. Just like a list comprehension, the general expressions are **concise**. In most cases, they are written in one line of code.

> *An expression that returns an iterator. It looks like a normal expression followed by a `for` expression defining a loop variable, range, and an optional `if` expression.*

Source: https://docs.python.org/3.7/glossary.html#term-generator-expression

The **general formula** is:
**(output expression for iterator variable in iterable)**

Let's see how we can define a simple generator expression.

```python
numbers = [1, 2, 3, 4, 5]
squares = (number**2 for number in numbers)
print(type(squares))
print(next(squares))
print(next(squares))
print(next(squares))
```

Generator Expression Example.py hosted with ♡ by GitHub view raw

Output:

```
<class 'generator'>
1
4
9
```

We can also add a **conditional expression on** the **iterable**. We can do it like this:

```python
numbers = [1, 2, 3, 4, 5]
squares = (number**2 for number in numbers if number % 2 == 0)
print(type(squares))
print(list(squares))
```

Generator Expression Example 2.py hosted with ♡ by GitHub view raw

Output:

```
<class 'generator'>
[4, 16]
```

They can be **multiple conditional expressions on** the **iterable** for more complex filtering.

```python
numbers = [1, 2, 3, 4, 5]
squares = (number**2 for number in numbers if number % 2 == 0 if number % 4 == 0)
print(type(squares))
print(list(squares))
```

Generator Expression Example 3.py hosted with ♡ by GitHub view raw

Output:

```
<class 'generator'>
[16]
```

Also, we can add an **if-else clause on** the **output expression** like this:

```python
numbers = [1, 2, 3, 4, 5]
result = ("even" if number % 2 == 0 else "odd" for number in numbers)
print(type(result))
print(list(result))
```

Generator Expression Example 4.py hosted with ♡ by GitHub view raw

Output:

```
<class 'generator'>
['odd', 'even', 'odd', 'even', 'odd']
```

## Takeaways / Summary

- An **iterable** is something you can **loop over**.
- **Sequences** are a very common type of **iterable**.
- Many things in Python are iterables, but not all of them are sequences.
- An **iterator** is an object representing a **stream of data**. It does the iterating over an iterable. You can use an iterator to get the **next value** or to **loop over it**. **Once**, you loop over an iterator, there are no more stream values.
- Iterators use the **lazy evaluation** approach.
- **Many built-in** classes in Python are **iterators**.
- A **generator function** is a function which returns an **iterator**.
- A **generator expression** is an expression that returns an **iterator**.

## What to read next?

- You can look at the itertools library. This library includes functions creating iterators for efficient looping.
- You can also read the docs or read/watch some of the resources below.

- My next article about list comprehensions in Python.

## Resources

- https://opensource.com/article/18/3/loop-better-deeper-look-iteration-python
- https://www.youtube.com/watch?v=V2PkkMS2Ack
- https://www.datacamp.com/community/tutorials/python-iterator-tutorial
- https://www.datacamp.com/courses/python-data-science-toolbox-part-2
- https://en.wikipedia.org/wiki/Foreach_loop
- https://docs.python.org/3.7/glossary.html#term-sequence
- https://docs.python.org/3.7/glossary.html#term-generator
- https://docs.python.org/3.7/library/stdtypes.html#iterator-types
- https://docs.python.org/3/howto/functional.html?#iterators

## Other Blog Posts by Me

You can also check my previous blog posts.

- Jupyter Notebook Shortcuts
- Python Basics for Data Science
- Python Basics: List Comprehensions
- Data Science with Python: Intro to Data Visualization with Matplotlib
- Data Science with Python: Intro to Loading, Subsetting, and Filtering Data with pandas
- Introduction to Natural Language Processing for Text

## Newsletter

If you want to be notified when I post a new blog post you can subscribe to my newsletter.

## LinkedIn

Here is my LinkedIn profile in case you want to connect with me. I'll be happy to be connected with you.

## Final Words

Thank you for the read. I hope that you have enjoyed the article. If you like it, please hold the clap button and share it with your friends. I'll be happy to hear your feedback. If you have some questions, feel free to ask them. 😉

Python Programming Iteration Iterators Python Programming