

# **Python Iterators**



Next >

## **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.

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. 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

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

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

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
Try it Yourself »
```

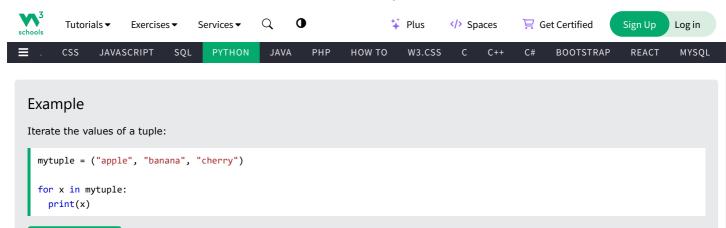
Even strings are iterable objects, and can return an iterator:

#### Example

Strings are also iterable objects, containing a sequence of characters:

```
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
TryitYourself»
```



### Example

Try it Yourself »

Iterate the characters of a string:

```
mystr = "banana"

for x in mystr:
    print(x)
```

Try it Yourself »

The for loop actually creates an iterator object and executes the next() method for each loop.

#### Create an Iterator

To create an object/class as an iterator you have to implement the methods \_\_iter\_\_() and \_\_next\_\_() to your object.

As you have learned in the <u>Python Classes/Objects</u> chapter, all classes have a function called <u>\_\_init\_\_()</u>, which allows you to do some initializing when the object is being created.

The \_\_iter\_\_() method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.

The \_\_next\_\_() method also allows you to do operations, and must return the next item in the sequence.

### Example

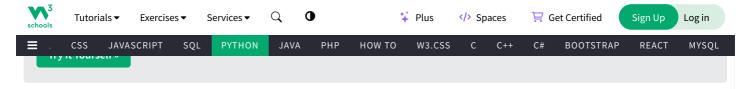
Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

```
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self

def __next__(self):
        x = self.a
        self.a += 1
        return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
```



## Stoplteration

Try it Yourself »

The example above would continue forever if you had enough next() statements, or if it was used in a for loop.

To prevent the iteration from going on forever, we can use the **StopIteration** statement.

In the \_\_next\_\_() method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

### Example Stop after 20 iterations: class MyNumbers: def \_\_iter\_\_(self): self.a = 1return self def \_\_next\_\_(self): if self.a <= 20:</pre> x = self.a self.a += 1 return x else: raise StopIteration myclass = MyNumbers() myiter = iter(myclass) for x in myiter: print(x)









