Python iter

Summary: in this tutorial, you'll learn how to use the Python `iter()` built-in function effectively.

Introduction to the Python iter function

The `iter()` function returns an iterator of a given object:

```
iter(object)
```

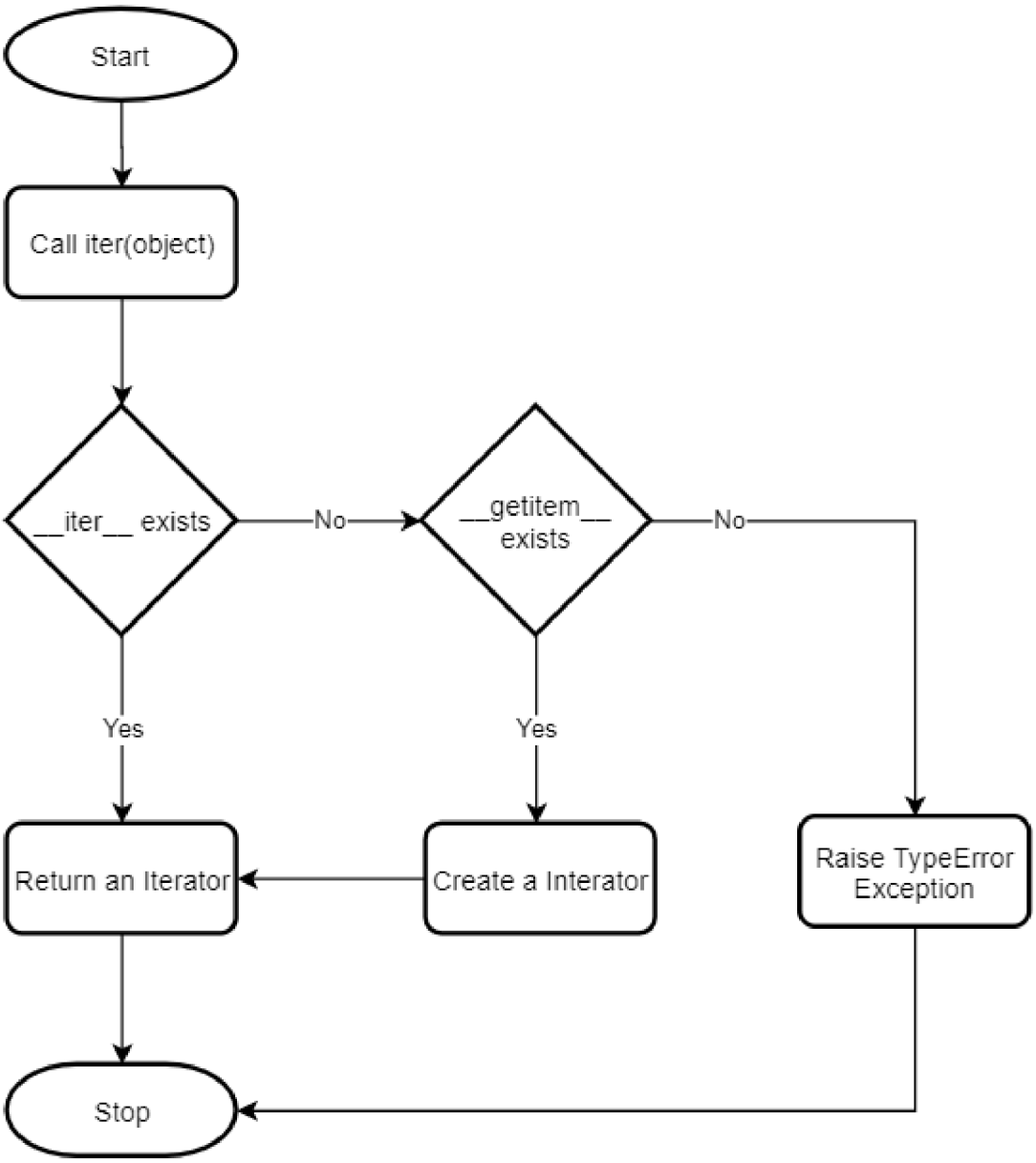
The `iter()` function requires an argument that can be an iterable (https://www.pythontutorial.net/python-basics/python-iterables/) or a sequence (https://www.pythontutorial.net/advanced-python/python-sequences/) . In general, the object argument can be any object that supports either iteration or sequence protocol.

When you call the `iter()` function on an object, the function first looks for an `__iter__()` method of that object.

If the `__iter__()` method exists, the `iter()` function calls it to get an iterator. Otherwise, the `iter()` function will look for a `__getitem__()` method.

If the `__getitem__()` is available, the `iter()` function creates an iterator object and returns that object. Otherwise, it raises a `TypeError` exception.

The following flowchart illustrates how the `iter()` function works:



Python iter() function examples

The following example defines a simple `Counter` class and uses the `iter()` function to get an iterator of the counter object:

```
class Counter:
    def __init__(self):
        self.__current = 0
```

```
counter = Counter()
iterator = iter(counter)
```

It'll raise a `TypeError` because the `counter` object is not an iterable:

```
TypeError: 'Counter' object is not iterable
```

The following adds the `__getitem__()` method to the `Counter` class:

```
class Counter:
    def __init__(self):
        self.current = 0

    def __getitem__(self, index):
        if isinstance(index, int):
            self.current += 1
            return self.current
```

Because the `Counter` implements the `__getitem__()` method that returns an element based on an index, it's a sequence.

Now, you can use the `iter()` function to get the iterator of the counter:

```
counter = Counter()

iterator = iter(counter)
print(type(iterator))
```

Output:

```
<class 'iterator'>
```

In this case, Python creates an iterator object and returns it. Hence, you can use the iterator object to iterate the `counter`:

```
for _ in range(1, 4):
    print(next(iterator))
```

The following adds the `CounterIterator` class to the `Counter` class and implement the iterable protocol:

```
class Counter:
    def __init__(self):
        self.current = 0

    def __getitem__(self, index):
        if isinstance(index, int):
            self.current += 1
            return self.current

    def __iter__(self):
        return self.CounterIterator(self)

    class CounterIterator:
        def __init__(self, counter):
            self.__counter = counter

        def __iter__(self):
            return self

        def __next__(self):
            self.__counter.current += 1
            return self.__counter.current
```

How it works.

- The Counter class implements the `__iter__()` method that returns an iterator. The return iterator is a new instance of the `CounterIterator` .
- The `CounterIterator` class supports the iterator protocol by implementing the `__iter__()` and `__next__()` methods.

When both `__iter__()` and `__getitem__()` methods exist, the `iter()` function always uses the `__iter__()` method:

```
counter = Counter()

iterator = iter(counter)
print(type(iterator))
```

Output:

```
<class '__main__.Counter.CounterIterator'>
1
2
3
```

In this example, the `iter()` function calls the `__iter__()` method instead of `__getitem__()` method. That's why you see the `CounterIterator` in the output.

The second form of the Python iter() function

The following shows the second form of the `iter()` function:

```
iter(callable, sentinel)
```

The `iter(callable,sentinel)` will call a callable when the `next()` method is called.

It'll return the value returned by the callable or raise the StopIteration exception if the result is equal to the sentinel value.

Let's take an example to understand how the `iter(callable, sentinel)` works.

First, define a function that returns closure:

```
def counter():
    count = 0

    def increase():
        nonlocal count
        count += 1
        return count

    return increase
```

The `counter()` function returns a closure. And the closure returns a new integer starting from one when it's called.

Second, use the `counter()` function to show the numbers from 1 to 3:

```
cnt = counter()

while True:
    current = cnt()
    print(current)
    if current == 3:
        break
```

Output:

```
1
2
3
```

To make it more generic, you can use an iterator instead.

Third, define a new counter iterator:

```
class CounterIterator:
    def __init__(self, fn, sentinel):
        self.fn = fn
        self.sentinel = sentinel

    def __iter__(self):
        return self

    def __next__(self):
        current = self.fn()
        if current == self.sentinel:
            raise StopIteration

        return current
```

The `CounterIterator` 's constructor accepts a callable `fn` and a `sentinel` .

The `__next__()` method returns the value returned by the callable (`fn`) or raise a `StopIteration` exception if the return value equals the sentinel.

The following shows how to use the CounterIterator:

```
cnt = counter()
iterator = CounterIterator(cnt, 4)
for count in iterator:
    print(count)
```

Output:

```
1
2
3
```

Instead of defining a new iterator every time you want to iterate values returned by the callable, you can use the `iter(callable, sentinel)` function:

```
cnt = counter()
iterator = iter(cnt, 4)

for count in iterator:
    print(count)
```

Output:

```
1
2
3
```

Use Python iter() function to test if an object is iterable

To determine whether an object is iterable, you can check if it implements the `__iter__()` or `__getitem__()` method.

However, you can use the `iter()` function to test if an object is iterable as follows:

```
def is_iterable(object):
    try:
        iter(object)
    except TypeError:
        return False
    else:
        return True
```

If the object doesn't implement neither `__iter__()` method nor `__getitem__()` method, the `iter()` function raises the `TypeError` exception.

The following shows how to use the `is_iterable()` function:

```
print(is_iterable([1, 2, 3]))
print(is_iterable('Python iter'))
print(is_iterable(100))
```

Output:

```
True
True
False
```

Summary

- Use the Python `iter()` function to get an iterator of an object.