Iterators

Iterator protocol

In Python, an iterator is an **object** that implements the **iterator protocol**, which consists of the methods `__iter__()` and `__next__()`. An iterator is used to *iterate over a sequence of values*, such as the items in a list or the characters in a string.

The `__iter__()` method returns the *iterator object itself*, while the `__next__()` method returns *the next value in the sequence* or *raises the `StopIteration` exception* if there are no more values.

Here's an example of using an iterator to iterate over the items in a list:

```
my_list = [1, 2, 3]
my_iter = iter(my_list)

print(next(my_iter))  # Output: 1
print(next(my_iter))  # Output: 2
print(next(my_iter))  # Output: 3

# Raises StopIteration because there are no more values
print(next(my_iter))
```

In the example above, we use the **`iter()`**[1] function to *create an iterator object for the list `[1, 2, 3]`*. We then use the **`next()`**[2] function to iterate over the items in the list, printing each value as we go. When there are no more values to iterate over, the `next()` function raises the `StopIteration` exception.

```
my_list = [1, 2, 3]
my_iter = iter(my_list)
```

[1] the `iter()` function creates an iterator object from an iterable object. The `iter()` function takes a single argument, which is the iterable object to be converted to an iterator. When the `iter()` function is called on an iterable object, it returns an iterator object that knows how to traverse the iterable object one item at a time.

[2] Note that calling the `next()` function on an iterator is equivalent to calling the iterator's `__next__()` method. In fact, the `next()` function is a built-in function in Python that simply calls the `__next__()` method on the iterator object passed as its argument.

```
while True:
    try:
        print(next(my_iter))
    except StopIteration:
        break
```

In addition to lists, iterators can be used to iterate over a variety of other Python objects, including *strings*, *dictionaries*, and *sets*. You can also *define your own iterators* by implementing the iterator protocol in your own classes.

Here's an example of a simple class that implements the iterator protocol in Python:

```
class MyIterator:
    def __init__(self, values):
        self.values = values
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index >= len(self.values):
            raise StopIteration
        value = self.values[self.index]
        self.index += 1
        return value
```

In this example, we define a class `MyIterator` that takes a sequence of values as input and implements the iterator protocol using the `__iter__()` and `__next__()` methods.

The `__init__()` method initializes the object with the given sequence of values and sets the initial index to `0`.

The `__iter__()` method simply returns the iterator object itself.

The `__next__()` method returns the next value in the sequence or raises the `StopIteration` exception if there are no more values.

Here's an example of using this class to iterate over a sequence of values:

```
values = [1, 2, 3, 4, 5]
my_iter = MyIterator(values)

for value in my_iter:
    print(value)
```

In this example, we create an instance of the `MyIterator` class with the sequence `[1, 2, 3, 4, 5]`. We then use a for loop to iterate over the sequence, printing

each value as we go. The for loop *implicitly* calls the `__iter__()` method to obtain the iterator object, and then calls the `__next__()` method repeatedly to retrieve the values in the sequence.

Iterators are used extensively in Python, both directly and indirectly. For example, many Python libraries and functions use iterators internally to process large datasets or generate sequences of values on-the-fly. Understanding how iterators work and how to use them effectively is an important part of writing efficient and effective Python code.

Iterator vs Iterable

In Python, an iterable is *an object that can be iterated over*, which means it can be used as a source of values in a *for loop* or other type of iteration. An iterator, on the other hand, is *an object that actually performs the iteration*, returning the next value in the sequence each time its `__next__()` method is called.

In other words, an iterable is a collection of items, while an iterator is an object that knows how to traverse that collection of items one by one.

An object is iterable if it implements the `__iter__()` method, which returns an iterator object. The iterator object itself implements the `__next__()` method, which returns the next item in the sequence, or raises the `StopIteration` exception if there are no more items to return.

Here's an example to illustrate the difference:

```
my_list = [1, 2, 3]

# my_list is an iterable because it has an __iter__()
# method that returns an iterator object
for item in my_list:
    print(item)

# We can also manually create an iterator object for
# my_list using the iter() function
my_iter = iter(my_list)

# my_iter is an iterator because it has a __next__() method
# that returns the next item in the sequence
print(next(my_iter))  # Output: 1
print(next(my_iter))  # Output: 2
print(next(my_iter))  # Output: 3
```

In this example, `my_list` is an iterable because it has an `__iter__()` method that returns an iterator object. We can use a for loop to iterate over the items in the list, or we can manually create an iterator object using the `iter()` function and retrieve each item using the `next()` function. The iterator object `my_iter`

is the actual object that performs the iteration by returning the next item in the sequence each time its `__next__()` method is called.

Here's an example of implementing two classes, one that is iterable and the other that is an iterator, and shows how to use them together to iterate over a sequence of items:

```
class MyIterable:
    def __init__(self, items):
        self.items = items

    def __iter__(self):
        return MyIterator(self.items)


class MyIterator:
    def __init__(self, items):
        self.index = 0
        self.items = items

    def __next__(self):
        if self.index >= len(self.items):
            raise StopIteration
        value = self.items[self.index]
        self.index += 1
        return value


# create an iterable object
my_iterable = MyIterable([1, 2, 3, 4, 5])

# iterate over the items in the iterable
for item in my_iterable:
    print(item)
```

In this example, we define two classes: `MyIterable`, which is an iterable object, and `MyIterator`, which is an iterator object.

The `MyIterable` class has an `__init__()` method that takes a list of items as its argument and initializes an instance variable called items to store the list of items.

The `MyIterable` class also has an `__iter__()` method that returns an instance of the `MyIterator` class initialized with the list of items stored in self.items.

The `MyIterator` class has an `__init__()` method that takes a list of items as its argument and initializes an instance variable called index to `0` to keep track of the current index of the iterator, and an instance variable called items to store

the list of items.

The `MyIterator` class also has a `__next__()` method that returns the next item in the list of items each time it is called, or raises a `StopIteration` exception if there are no more items to return.

To use these classes, we create an instance of the `MyIterable` class with the sequence `[1, 2, 3, 4, 5]`, and then use a for loop to iterate over the items in the iterable object. During each iteration, the for loop calls the `__iter__()` method of the iterable object to obtain an instance of the `MyIterator` class, and then repeatedly calls the `__next__()` method of the iterator object to retrieve the next item in the sequence. The loop exits when the iterator's `__next__()` method raises a `StopIteration` exception.

src: chatgpt

How `list` class in Python implemented in terms of iterator

When the `__iter__()` method is called on a list object, it returns an iterator object that is created by calling the `iter()` built-in function on the list object itself. The returned iterator object is an instance of the built-in `listiterator` class, which is a specialized iterator class that knows how to traverse the list one item at a time.

Here's an excerpt from the Python source code for `list` and `listiterator`, showing the implementation of the iterator protocol:

```
/* List object */
typedef struct {
    PyObject_VAR_HEAD
    PyObject **ob_item;
    Py_ssize_t allocated;
} PyListObject;

/* List iterator object */
typedef struct {
    PyObject_HEAD
    PyListObject *it_list;
    Py_ssize_t it_index;
} listiterator;

/* List type object */
PyTypeObject PyList_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "list",
    sizeof(PyListObject),
```

```
    ...
    (reprfunc)list_repr,                        /* tp_repr */
    ...
    (getiterfunc)list_iter,                     /* tp_iter */
    0,                                          /* tp_iternext */
    ...
};

/* Iterator function for list objects */
static PyObject *
list_iter(PyObject *seq)
{
    PyListObject *list = (PyListObject *)seq;
    listiterator *it;

    if (list == NULL) {
        PyErr_BadInternalCall();
        return NULL;
    }

    it = PyObject_GC_New(listiterator, &PyListIter_Type);
    if (it == NULL)
        return NULL;

    Py_INCREF(list);
    it->it_list = list;
    it->it_index = 0;
    _PyObject_GC_TRACK(it);
    return (PyObject *)it;
}

/* Next method for list iterator objects */
static PyObject *
listiter_next(listiterator *it)
{
    PyListObject *seq = it->it_list;
    Py_ssize_t i = it->it_index;

    if (i < Py_SIZE(seq)) {
        PyObject *r = seq->ob_item[i];
        it->it_index = i + 1;
        Py_INCREF(r);
        return r;
    }
    return NULL;
```

```
}
```

In this code, we see the definition of the `PyListObject` struct, which represents a list object in Python. The `PyListObject` struct contains a variable-sized array called `ob_item`, which stores the list's items.

We also see the definition of the `listiterator` struct, which represents an iterator object for a list. The `listiterator` struct contains a pointer to the list object being iterated over, and an integer index indicating the current position of the iterator within the list.

The `PyList_Type` object represents the list type in Python. We see that the `tp_iter` field of `PyList_Type` is set to the `list_iter()` function, which is called to create an iterator object for a list. The `tp_iternext` field is set to `0`, indicating that the `next()` method for the iterator will be implemented separately.

The `list_iter()` function takes a PyObject representing a list object as its argument. It creates a new `listiterator` object and initializes it with a pointer to the list object and an initial index of `0`. It returns a pointer to the new `listiterator` object.

The `listiter_next()` function is the implementation of the `next()` method for `listiterator` objects. It takes a pointer to a `listiterator` object as its argument, and retrieves the next item in the list using the iterator's `it_index` field to index into the `ob_item` array of the list object. If there are no more items in the list, it returns `NULL`. If it retrieves an item, it increments the iterator's `it_index` field and returns a pointer to the item.

src: chatgpt

Further Examples:

Fibonacci sequence

```
class FibonacciSequence:
    def __init__(self, n):
        self.n = n

    def __iter__(self):
        return FibonacciIterator(self.n)

class FibonacciIterator:
    def __init__(self, n):
        self.n = n
        self.current = 0
        self.next = 1
        self.index = 0
```

```
    def __next__(self):
        if self.index >= self.n:
            raise StopIteration
        result = self.current
        self.current, self.next = self.next, self.current + self.next
        self.index += 1
        return result

# usage
fibonacci_sequence = FibonacciSequence(10)
for number in fibonacci_sequence:
    print(number)
```

An Image Related WebApp

Suppose you are building a web application that serves a large number of images to users. You want to optimize the performance of the application by loading images on demand, rather than loading all the images at once. You can use an iterator and iterable to accomplish this.

First, you can define an `Image` class that represents a **single image**:

```
class Image:
    def __init__(self, filename):
        self.filename = filename
        self.data = None

    def load(self):
        if self.data is None:
            with open(self.filename, 'rb') as f:
                self.data = f.read()

    def display(self):
        if self.data is None:
            self.load()
        # Display the image
```

This class has a `load()` method that loads the image data from disk, and a `display()` method that displays the image.

Next, you can define an `ImageCollection` class that represents a **collection of images**. This class should be an **iterable**, so it has an `__iter__()` method that returns an *iterator object*:

```
# - Only stores the name of the files
# - Implements iterator protocol to load/display images on demand as user
#   iterates over each file name. So no Image is stored. So With iterator
```

```
#   protocol we told python what to do when we iterate over items in list
#   that contains potentially large/numerous items.
class ImageCollection:
    def __init__(self, filenames):
        self.filenames = filenames

    def __iter__(self):
        return ImageIterator(self.filenames)

class ImageIterator:
    def __init__(self, filenames):
        self.filenames = filenames
        self.index = 0

    def __next__(self):
        if self.index >= len(self.filenames):
            raise StopIteration
        filename = self.filenames[self.index]
        image = Image(filename)
        self.index += 1
        return image
```

Now, you can use the `ImageCollection` class to load and display images on demand in your web application:

```
image_filenames = ['image1.jpg', 'image2.jpg', 'image3.jpg', ...]
image_collection = ImageCollection(image_filenames)

for image in image_collection:
    image.display()
```

Using other data structures as the underlying data to be iterated over.

In addition to lists, there are many other data structures that can be used with iterators and iterables in Python. Here's an example using a **dictionary**:

Suppose you have a dictionary that maps *names* to *email addresses*:

```
emails = {
    'Alice': 'alice@example.com',
    'Bob': 'bob@example.com',
    'Charlie': 'charlie@example.com',
    'Dave': 'dave@example.com',
}
```

You can define an `Emails` class that represents the *email addresses* in this dictionary. This class should be an iterable, so it has an `__iter__()` method that returns an iterator object:

```
class Emails:
    def __init__(self, emails):
        self.emails = emails

    def __iter__(self):
        return iter(self.emails.values()) # values: email address
```

Now, you can use the `Emails` class to iterate over the email addresses:

```
emails_iter = Emails(emails)

for email in emails_iter:
    print(email)
```

This example shows that any data structure that implements the iterable protocol can be used with iterators and iterables in Python. This includes not just sequences like lists and tuples, but also sets, dictionaries, and custom classes that implement the iterable protocol.