Iterator Design Pattern

Iterators in Python

Iterator looks incredibly simple, but its quite important and a good example of applying the patterns concept.

What Can You Use in a for Loop?

```
for x in ___________:
    dowork(x)
```

or

```
result = [f(x) for x in ___________]
```

Name some <u>types</u> that can go in the blank

Iterator Design Pattern

Pattern Name: **Iterator**

Context

We need to access elements of a collection or data src.

Motivation (Forces)

We want to access elements of a collection without the need to know the underlying structure of the collection.

Solution

Each collection provides an iterator with a method to get the next element.

Consequences

Application is not coupled to the kind of collection.
Collection type can be changed w/o changing other code.

Using an Iterator

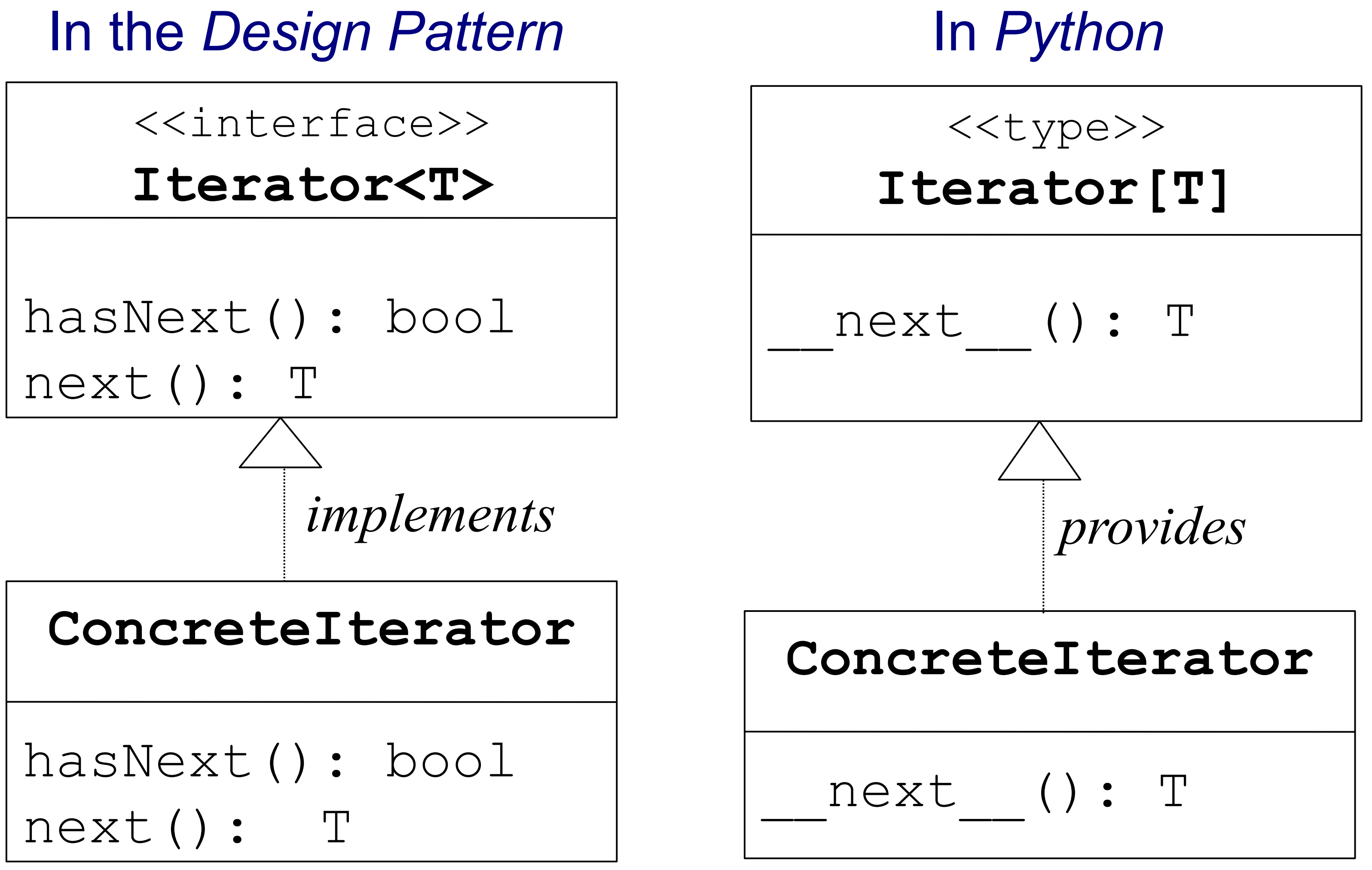
In Python you rarely use iterators directly, but you can.

```
>>> fruit = ["Apple", "Banana", "Durian", ...]
>>> iter = iter(fruit)   # create an iterator

>>> next(iter)           # calls iter.__next__()
'Apple'
>>> next(iter)           # calls iter.__next__()
'Banana'

>>> str_iter = iter("Hello")
>>> next(str_iter)
'H'
>>> next(str_iter)
'e'
```

Diagram for Iterator



`T` is a ***type parameter*.**

Interface

Interface = specify some required behavior (methods), but not the implementation of the behavior.

Python does not really have an *Interface* type.

In Python, an *abstract class* serves as an *interface.*

```
from collections.abc import ABC, abstractmethod
class Iterator(ABC):

    @abstractmethod
    def __next__(self):
        """Return the next element."""
        pass
```

Iterator in Python

`collections.abc.Iterator` - abstract base class

`typing.Iterator` - type hint, which has a parameter:

`Iterator[date]` = an iterator for date objects.

Example: an Appointments class provides iterators

```
class Appointments(Iterator[date])
```

How do you Get an Iterator?

Context:

We want to create an Iterator.

Forces:

We don't want our code to be coupled to a particular collection type. We want to always create iterators in the *same way*.

Create an Iterator in Python

The `__iter__` method creates an iterator.

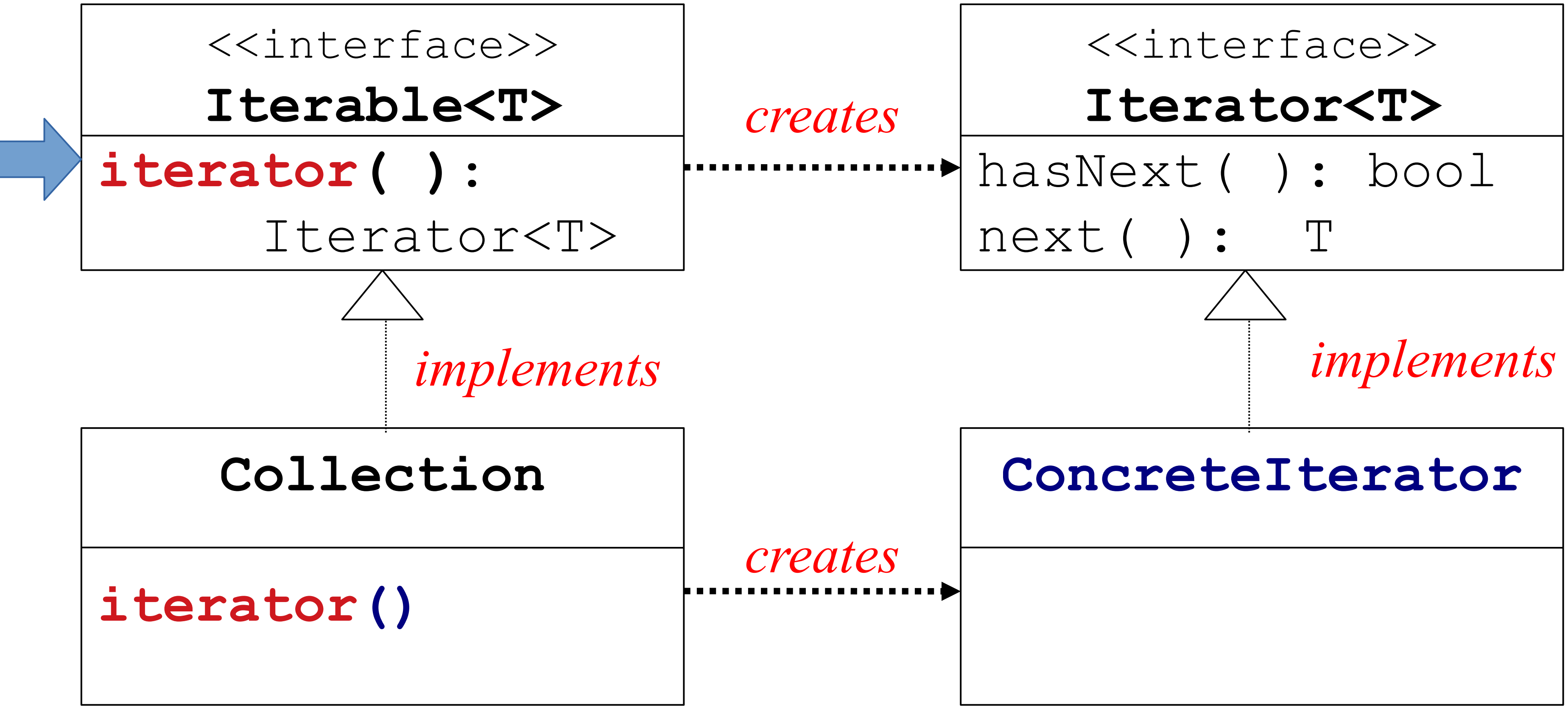
```
>>> fruit = ["Apple", "Banana", "Durian", ...]
>>> iter = fruit.__iter__()   # same as iter(fruit)

>>> next(iter)                # calls iter.__next__()
'Apple'
>>> next(iter)                # calls iter.__next__()
'Banana'
```

You should write `iter(fruit)`, not `fruit.__iter__()`.

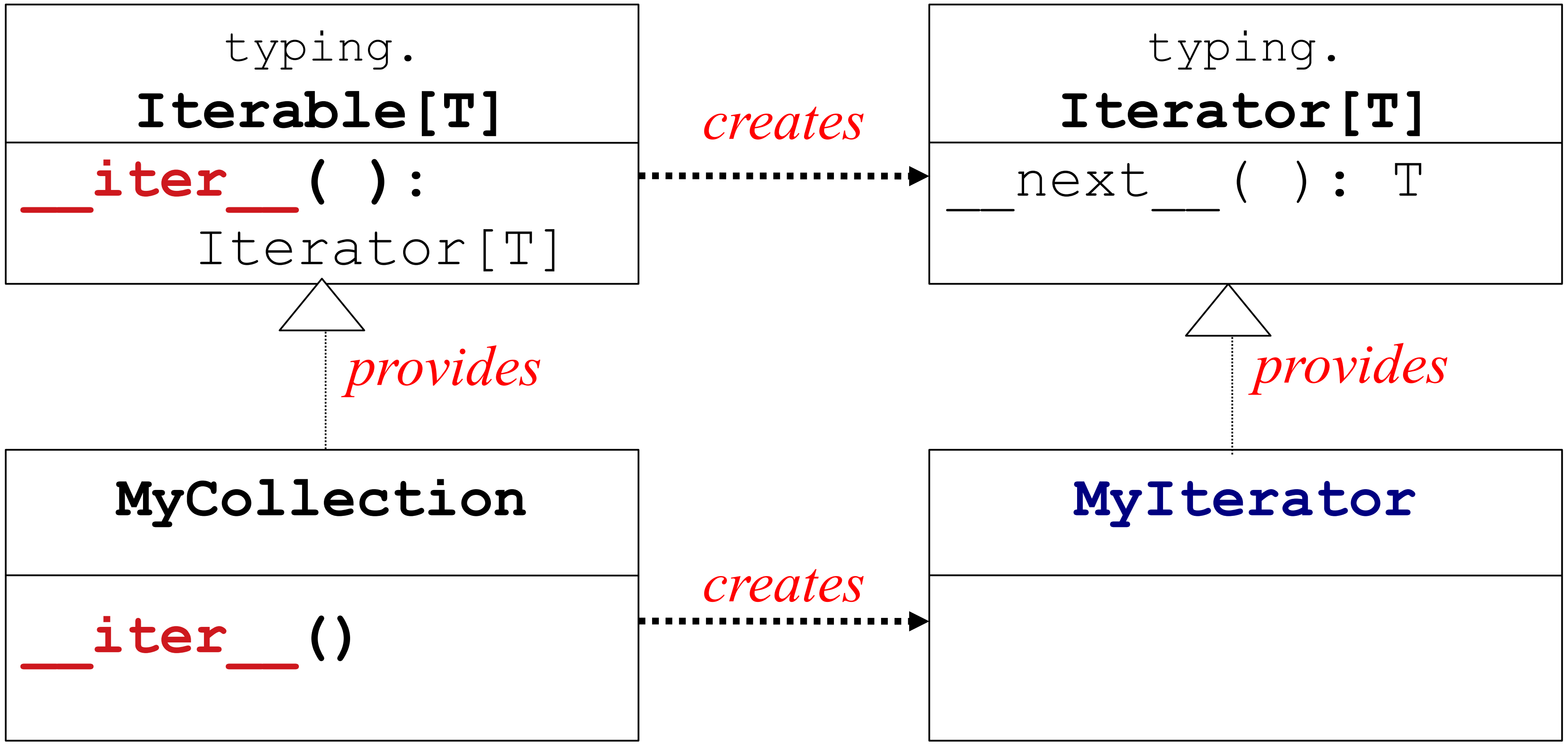
Solution: Define a *Factory Method*

Define a method that creates an Iterator.



Iterable in Python

In Python, an *Iterable* has a `__iter__` method that returns an Iterator.



What *Uses* an Iterable?

Anything that is *Iterable* or *Iterator* can be used as the data source in a "for" loop, list comprehension, or map.

for loop:

```
for x in iterable:
```

list comprehension

```
[f(x) for x in iterable if condition(x)]
```

map function:

```
map( function, iterable)
```

builtin functions:

```
max(iterable), min(iterable),
  sum(iterable), any(iterable), ...
```

What objects are Iterable?

`list`

`set`

`dict` (iterator over keys)

`file`

  `f = open("somefile.txt")`. Iterator returns lines

`string`

Generators

What *Django* classes are Iterable?

You can check you answer:

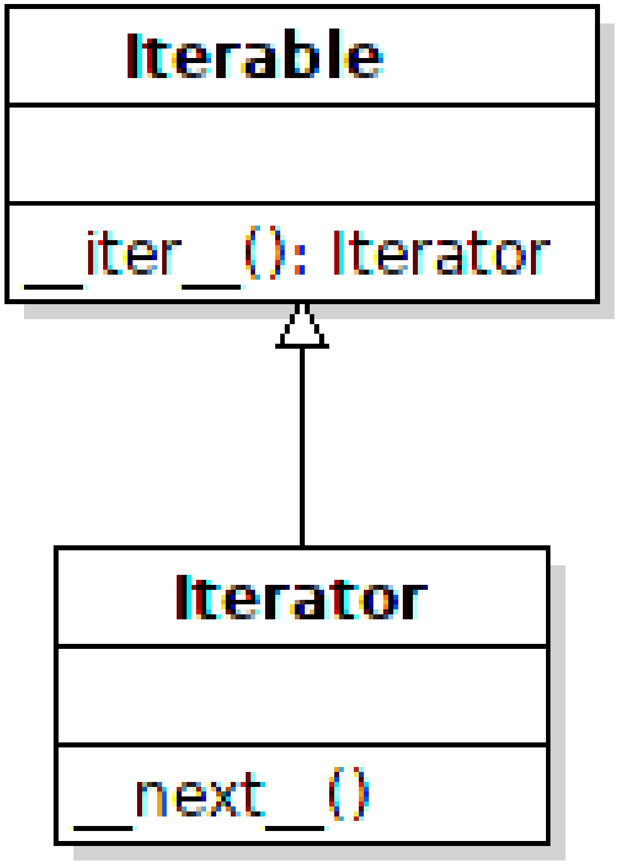
If `foo` is *Iterable* then:

```
>>> isinstance(foo, Iterable)
True
>>> it = iter(foo)
>>> next(it)
```

should return first element of `foo`

Python is Unusual

In Python collections.abc, *Iterable* is a **subtype** of *Iterator*

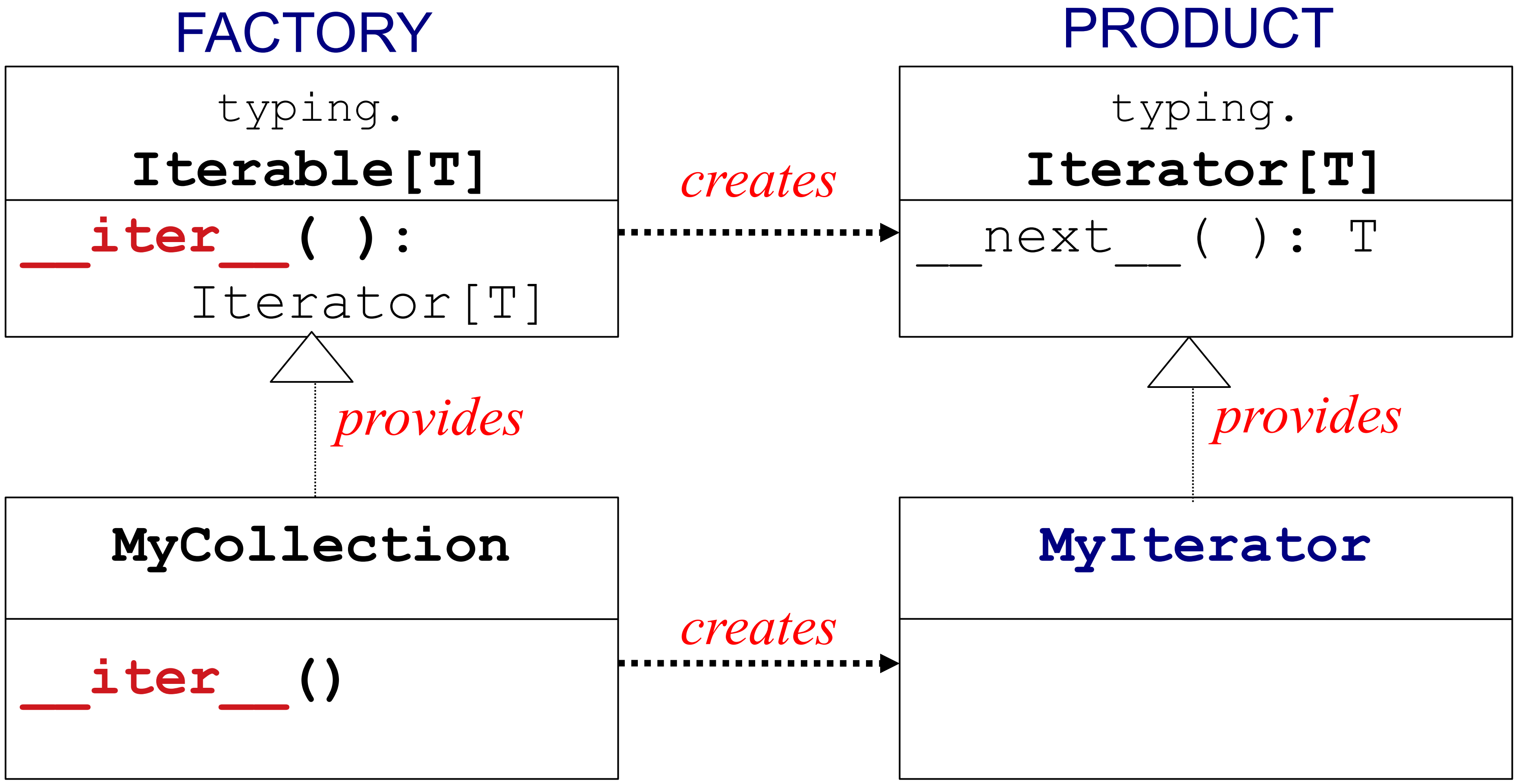


Iterators can create new iterators.
Just call `iter(iterator)`

`May not always work!!`

Another Design Pattern

Iterable & *Iterator* are an example of ***Factory Method Pattern***


FACTORY
PRODUCT
typing.
Iterable[T]
__iter__( ): Iterator[T]
creates
typing.
Iterator[T]
__next__( ): T
provides
provides
MyCollection
__iter__()
creates
MyIterator