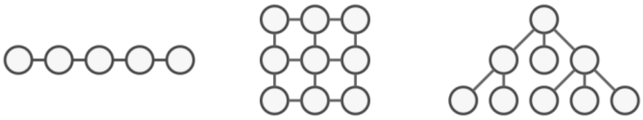
Design Patterns

The Iterator Pattern

The iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements. Contaier types include lists, tuples, dictionaries, and sets, and they can be structured in multiple ways:



Different Types of Collections

The general Iterator design pattern provides a way to access the elements of a collection object sequentially without exposing its underlying representation. The Iterator design pattern allows us to separate out all the logic for iterating over a collection. It allows an object to traverse through a container (collection of objects) without having the container to reveal how the data is structured internally. To achieve this the iterator pattern is designed so that the container object provides a public interface in the form of an interator object for different client objects to access its contents. It consists of two main classes:

- Iterable is a class that provides a way to expose its data to the public.
- Iterator is a class that contains a pointer to the next element in the iteration.

Generic Iterator Pattern

```
class ITERABLE:
      def __init__(self, VALUE):
            self.VALUE = VALUE
      # END Init

      def __iter__(self):
            return ITERATOR(self.VALUE)
      # END Iter
# END ITERABLE.
```

```
class ITERATOR:
      def __init__(self, VALUE):
                  self.VALUE = VALUE
                  self.index = 0
      # END Init
      def __iter__(self):
                  return self
      # END Iter
      def __next__(self):
            if CONDITION:
                  VALUE = SELF.VALUE
                  self.index = self.index + 1
                  return VALUE
            else:
                  raise StopIteration()
            # ENDIF;
      # END Next
# END ITERATOR.
```

This is the general design pattern, not the specific implementation.

Design Patterns

Python Iterator Pattern

To implement the Iterator Pattern, Python provides you with a pair of built-ins:

- `iter()` takes in a container object and builds and returns a new iterator object.
- `next()` takes in the iterator and, each time it is called, returns the next item from the container.When there are no more objects to return, the exception `StopIteration` is raised.

```
class MyCountIterable:
    def __init__(self, Value):
        self.Value = Value
    # END Init

    def __iter__(self):
        return MyCountIteration(self.Value)
    # END Iter
# END MyCountIterable.
```

```
class MyCountIteration:
    def __init__(self, Value):
        self.Index = 0
        self.Value = Value
    # END Init

    def __iter__(self):
    # Iterators are iterables too.
        return self
    # END Iter

    def __next__(self):
       if self.Index < self.Value:
            # THEN
            Index = self.Index
            self.Index += 1
            return Index
       else:
            raise StopIteration()
       # ENDIF;
    # END Next
# END MyCountIteration.
```

Here is the Python code to run the iterator program:

```
FirstCount = MyCountIterable(5)
list(FirstCount)
FirstCountIter = iter(FirstCount)
while True:
    try:
        print(next(FirstCountIter))
    except StopIteration:
          break
# ENDWHILE
```

This is clearly not as easy as a simple FOR loop counting 0 to 4, but it is a standard, and well-known pattern, and is readily recognisable by other programmers.