Loops

Iteration

- **Iteration** means executing the same block of code over and over, potentially many times.

- A programming structure that implements iteration is called a **loop**.

- Python has two primitive loop commands:
  - **while** loops
  - **for** loops

Iterations

- In programming, there are **two types** of iteration, indefinite and definite:
    - With **indefinite iteration**, the number of times the loop is executed isn't specified explicitly in advance. Rather, the designated block is executed repeatedly as long as some condition is met.
    - With **definite iteration**, the number of times the designated block will be executed is specified explicitly at the time the loop starts.

Iterations

- **Definite** iteration loops are frequently referred to as **for** loops, because for is the keyword that is used to introduce them in nearly all programming languages, including Python.

- In Python, indefinite iteration is performed with a **while** loop.

When do we use loops?

Digital Career Institute

- The **for** loops are traditionally used when you have a block of code which you want to repeat a **fixed** number of times.

- The Python **for** statement iterates over the members of a sequence in order, executing the block each time.

- Contrast the for statement with the **"while"** loop, used when a condition needs to be checked **each** iteration, or to repeat a block of code **forever**.

The **for** loop

Digital Career Institute

Numeric range loop

- The most basic for loop is a simple **numeric range** statement with start and end values. The exact format varies **depending** on the language but typically looks something like this:

  for i = 1 to 10

  <loop body>

- Here, the body of the loop is executed ten times. The variable **i** assumes the value 1 on the first iteration, 2 on the second, and so on.

Three-expression loop

- Another form of for loop popularized by the C programming language contains **three** parts:
    - An **initialization**
    - An **expression** specifying an **ending** condition
    - An **action** to be performed at the end of each iteration.

Three-expression loop

- Example:

```
for (i = 1; i <= 10; i++)
    <loop body>
```

- **Note:** In the C programming language, **i++** increments the variable i.

- It is roughly **equivalent** to **i += 1** in Python.

Collection-based or Iterator-based loop

Digital Career Institute

- This type of loop **iterates** over a **collection** of objects (string, numbers, etc.), rather than specifying numeric values or conditions:

```
for i in <collection>
    <loop body>
```

- Each time through the loop, the variable **i** takes on the value of the **next** object in <collection>

- This type of for loop is arguably **the most** generalized and abstract

The Python for loop

- Of the loop types listed above, Python only implements the last: **collection-based iteration**.

- Python **for** loop looks like this:

```
for <var> in <iterable>:
    <statement(s)>
```

The Python for loop

- <**iterable**> is a collection of objects (strings, numbers etc.), for example a sequence of numbers from range() function, list or tuple (two last will be covered later!)

- The <**statement(s)**> in the loop body are denoted by **indentation**, as with all Python control structures, and are executed once for each item in <**iterable**>

- The loop variable <**var**> takes on the value of the next element in <**iterable**> each time through the loop

The Python for loop - example no. 1

- In this example, <**iterable**> is the sequence of numbers **a**, and <var> is the variable **i**.
- Each time through the loop, **i** takes on a **successive item** in **a**, so print() displays the values 1, 2, 3 respectively.

```
>>> a = range(1, 4)
>>> for i in a:
...     print(i)
...
1
2
3
```

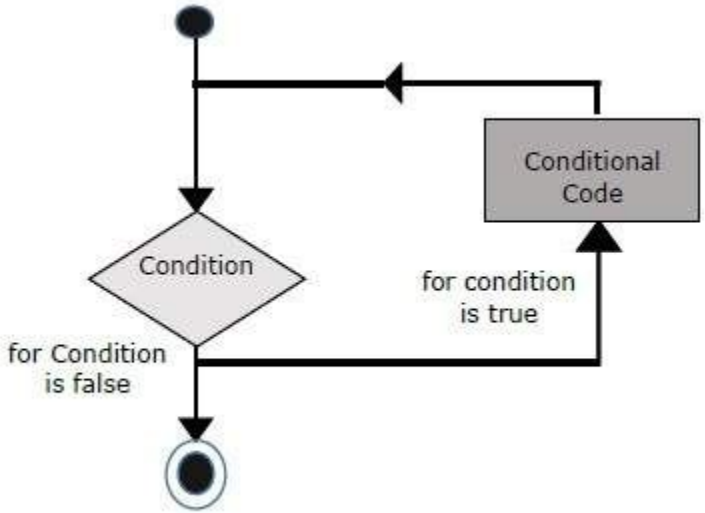
- A for loop like this is the **Pythonic** way to process the items in an iterable.

The Python for loop - example no. 2

- In this example, <**iterable**> is the string **txt**, and <var> is the variable **i**.
- Each time through the loop, **i** takes on a **successive item** in **txt**, so print() displays the values D, C, I respectively.

```
>>> txt = 'DCI'
>>> for i in txt:
...     print(i)
...
D
C
I
```

For loop


Conditional
Code
Condition
for condition
is true
for Condition
is false

Iterables

- In Python, **iterable** means an object can be used in iteration. The term is used as:
  - **An adjective:** An object may be described as **iterable**.
  - **A noun:** An object may be characterized as an **iterable**.

- If an object is iterable, it can be passed to the built-in Python function **iter()**, which returns something called an **iterator**.

- Yes, the terminology gets a bit repetitive, but it all works out in the end 🙂

Iterables

- Some data types known so far are iterable:

```
>>> iter('Hello')
<str_iterator object at 0x7f60ab891e80>
>>> iter(range(23))
<range_iterator object at 0x7f60ab8fd630>
```

- Also iterable are following types: **dict**, **list**, **tuple**, **set**, **frozenset** (you will get to know them **later**!)

Iterables

- Some data types known so far are **not** iterable:

```
>>> iter(True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bool' object is not iterable
>>> iter(234)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
>>> iter(3.45)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object is not iterable
```

Iterables - example

- Many objects that are built into Python or defined in modules are designed to be iterable.
- For example, **open files** in Python are iterable.
- Iterating over an open file object reads data from the file.
- It will be covered later!

Iterators, iterables, ... - terms

Term	Meaning
Iteration	The process of **looping through** the objects or items in a collection
Iterable	An object (or the adjective used to describe an object) that can be **iterated over**
Iterator	The object that **produces** successive items or values from its associated iterable
iter()	The **built-in function** used to obtain an iterator from an iterable

Iterators

- **name** is an iterable string and **itr** is the associated **iterator**, obtained with **iter()**. Each next(itr) call obtains the **next value** from itr.

```
>>> name = 'DCI'
>>> itr = iter(name)
>>> itr
<str_iterator object at 0x7f60ab891640>
>>> next(itr)
'D'
>>> next(itr)
'C'
>>> next(itr)
'I'
```

Iterators

Digital Career Institute

- An iterator **retains** its state internally.
- It knows which values have been obtained **already**, so when you call next(), it knows what value to return **next**.
- What happens when the iterator runs out of values?
- If all the values from an iterator have been returned already, a subsequent next() call raises a **StopIteration exception**.

Iterators

- Any further attempts to obtain values from the iterator will **fail**.

```
>>> next(itr)
'I'
>>> next(itr)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Iterators

Digital Career Institute

- You can only obtain values from an iterator in **one direction**.
- You can’t go backward. There is **no** prev() function.
- But you can define two independent iterators on the same iterable object.
- Each iterator maintains its own internal state, independent of the other.

Iterators

- Even when iterator itr1 is already at the end of the list, itr2 is still at the beginning.

```
>>> name = 'DCI'
>>> itr1 = iter(name)
>>> itr2 = iter(name)
>>> next(itr1)
'D'
>>> next(itr1)
'C'
>>> next(itr2)
'D'
```

Iterators

- If you want to grab all the values from an iterator at once, you can use the built-in **list()** function.

- Among other possible uses, list() takes an iterator as its **argument**, and returns a list consisting of all the values that the iterator yielded.

Iterators

- If you want to grab all the values from an iterator at once, you can use the built-in **list()** function.

- Among other possible uses, list() takes an iterator as its **argument**, and returns a list consisting of all the values that the iterator yielded:

```
>>> name = 'DCI'
>>> itr3 = iter(name)
>>> list(itr3)
['D', 'C', 'I']
```

Iterators “laziness”

Digital Career Institute

- Part of the elegance of iterators is that they are “**lazy**”
- That means that when you create an iterator, and it doesn’t generate all the items, it can **yield** just them.
- It waits until you ask for them with **next()**.
- Items are not created until they are **requested**.

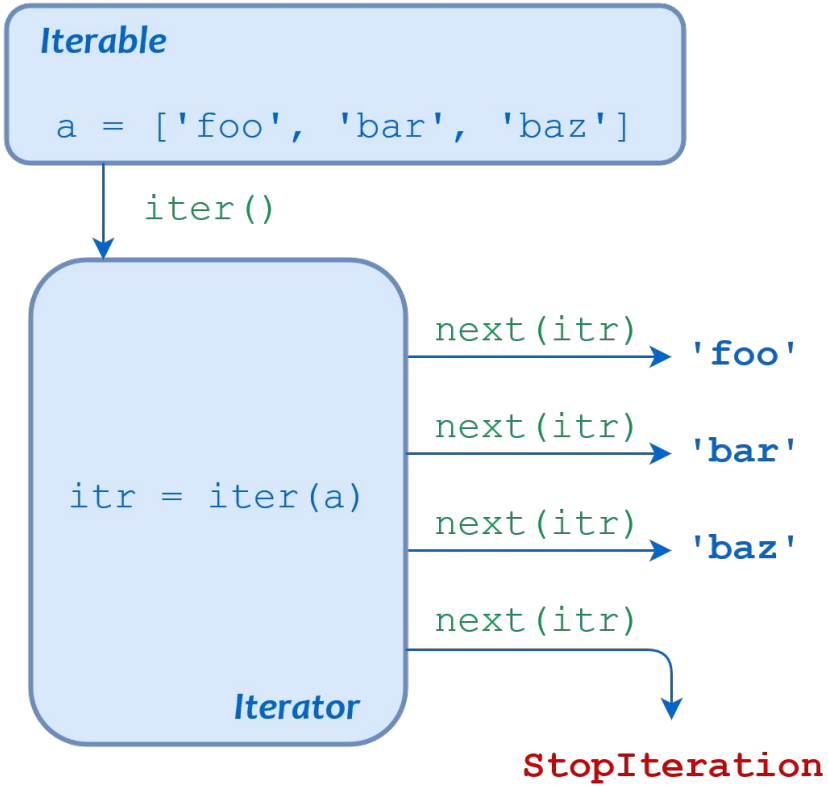
Iteration in **for** loop

Digital Career Institute

- To carry out the iteration this for loop describes, Python does the following:
  - Calls **iter()** to obtain an iterator for a
  - Calls **next()** repeatedly to obtain each item from the iterator in turn
  - Terminates the loop when next() raises the **StopIteration** exception

- The loop body is executed **once for each item** next() returns, with loop variable **i** set to the given item for each iteration.

Iteration in **for** loop



The **else** clause in **for** loop

- A for loop can have an **else** clause.
- The else clause will be executed if the loop **terminates** through exhaustion of the iterable:

```
>>> for i in range(3):
...     print(i)
... else:
...     print("Done printing numbers!")
...
0
1
2
Done printing numbers!
```

The **else** clause in **for** loop

Digital Career Institute

- Another example of using **else** in **for** loop:

```
for x in range(3):
    print(x)
else:
    print("Finally finished!")
# prints 0, 1, 2, "Finally finished!"
```