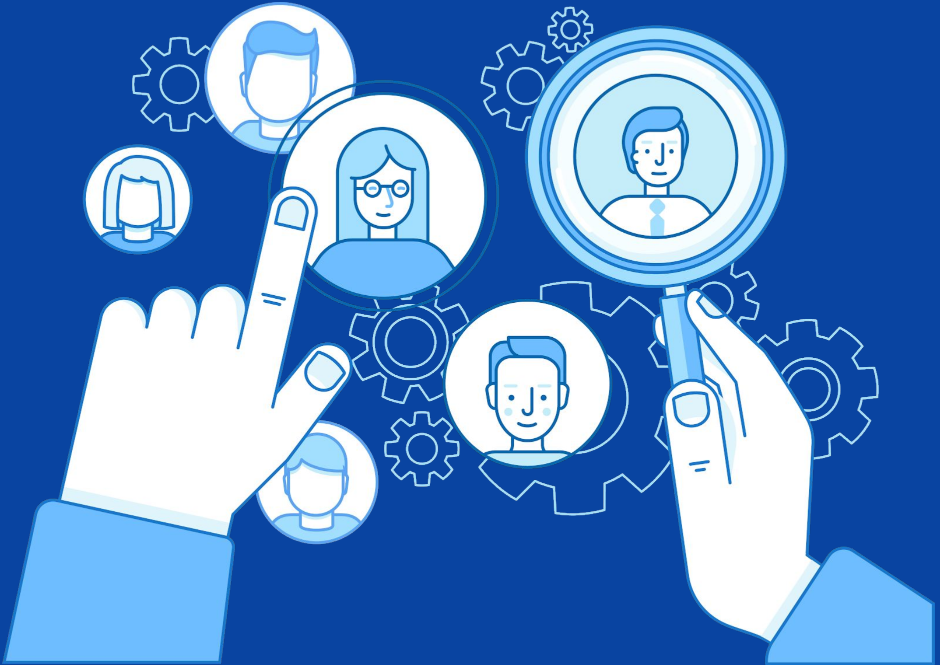
Digital Career Institute

Python Course - Statements & Loops



Digital Career Institute

Topics

- Statements in Python
- The for loop
- The while loop

Statements in Python

Digital Career Institute

A statement

- A **statement** is an instruction that the Python interpreter can execute.

- We have seen two kinds of statements so far:
    - print
    - assignment

A statement

- When you type a statement on the command line, Python **executes** it and displays the result, if there is one.
- The result of a print statement is a value. Assignment statements don't produce a result.
- A script usually contains a **sequence** of statements.
- If there is more than one statement, the results appear one at a time as the statements **execute**.

A statement

- For example, the script

```
print(3)
x = 4
print(x)
```

produces the output:

```
3
4
```

- Again, the assignment statement produces **no output**.

An expression

Digital Career Institute

- An expression is a combination of values, variables, and operators.
- If you type an expression on the command line, the interpreter **evaluates** it and displays the result:

```
>>> 1 + 2

3
```

- Although expressions contain **values**, **variables**, and **operators**, **not every** expression contains all of these elements.

An expression

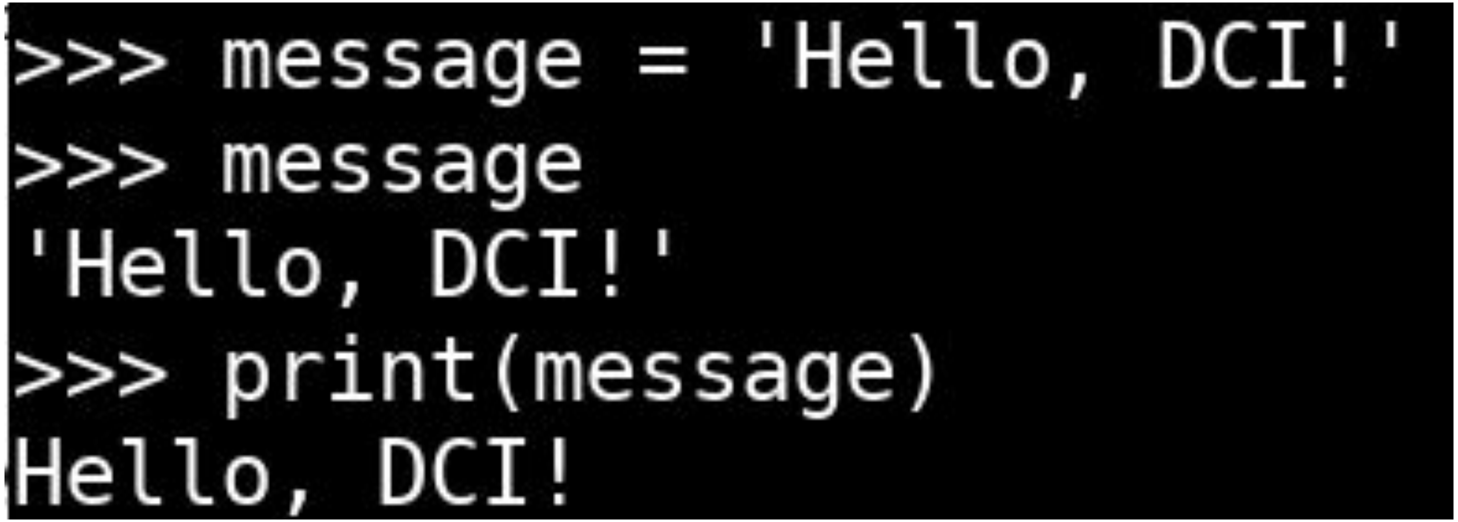
Digital Career Institute

- A **value** all by itself is considered an expression, and so is a **variable**:

```
>>> 23
23
>>> x
2
```

An expression

- Confusingly, evaluating an expression is not quite the same thing as printing a value:



```
>>> message = 'Hello, DCI!'
>>> message
'Hello, DCI!'
>>> print(message)
Hello, DCI!
```

An expression

Digital Career Institute

- When the Python interpreter displays the value of an expression, it uses the same format you would use to enter a value.
- In the case of strings, that means that it **includes** the quotation marks.
- But if you use a **print** statement, Python displays the contents of the string **without** the quotation marks.

Loops

Digital Career Institute

Iteration

Digital Career Institute

- **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

Digital Career Institute DCI

- 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?

- 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

Digital Career Institute

- 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

- 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

Digital Career Institute

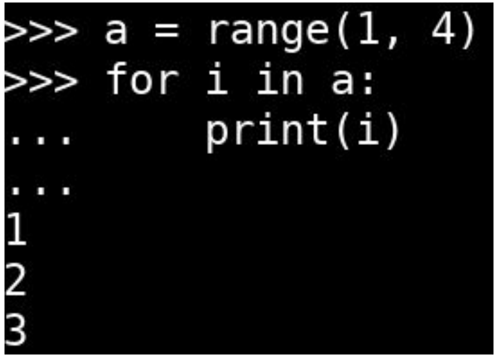
- <**iterable**> is a collection of objects (strings, numbers etc.), for example a sequence of numbers from range() function, list or tuple (two last will be covered later!)

- 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.

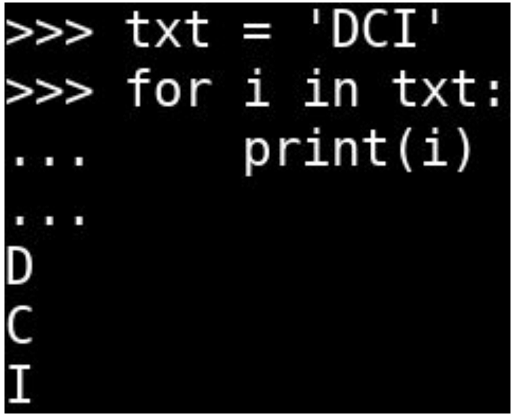


```
>>> a = range(1, 4)
>>> for i in a:
...     print(i)
...
1
2
3
```

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

The Python for loop - example no. 2

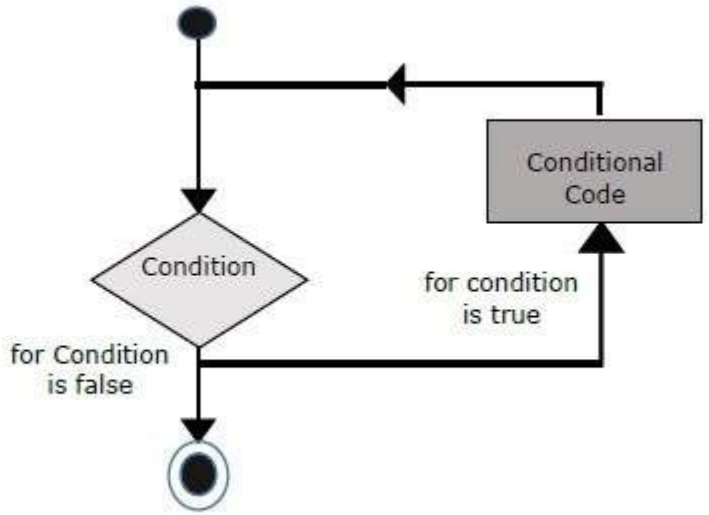
- In this example, <**iterable**> is the string **txt**, and <var> is the variable **i**.
- Each time through the loop, **i** takes on a **successive item** in **txt**, so print() displays the values D, C, I respectively.



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

For loop

Digital Career Institute



Iterables

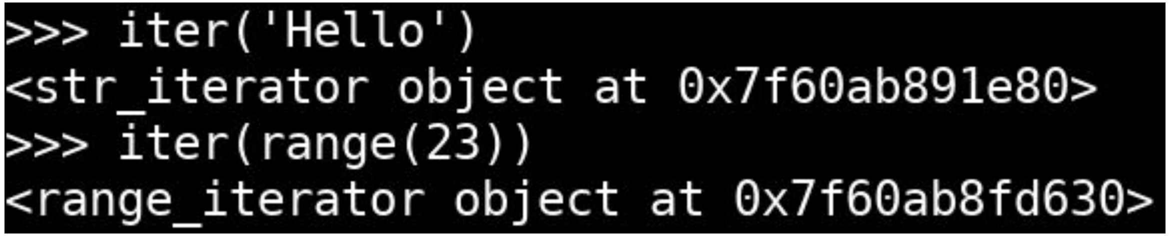
- In Python, **iterable** means an object can be used in iteration. The term is used as:
  - **An adjective:** An object may be described as **iterable**.
  - **A noun:** An object may be characterized as an **iterable**.

- 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:

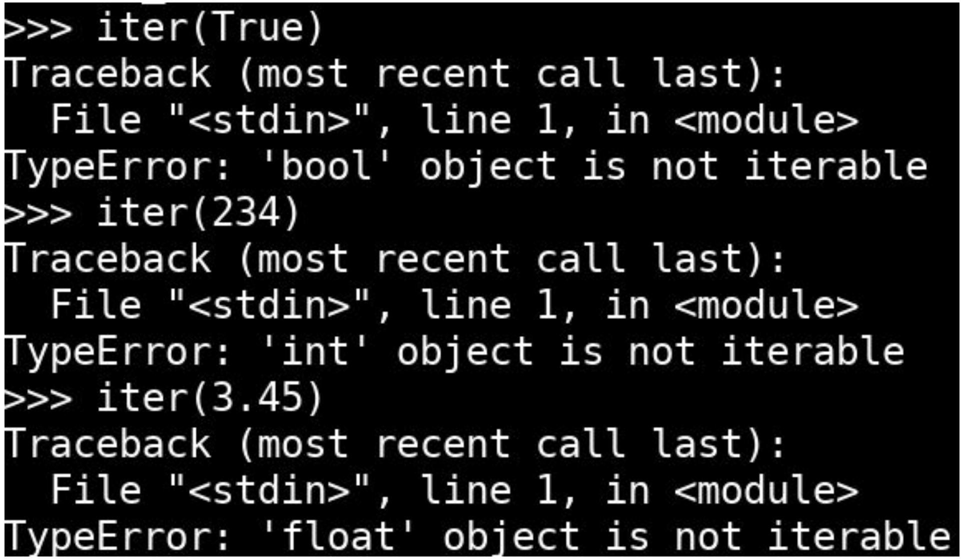


```
>>> iter('Hello')
<str_iterator object at 0x7f60ab891e80>
>>> iter(range(23))
<range_iterator object at 0x7f60ab8fd630>
```

- 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:



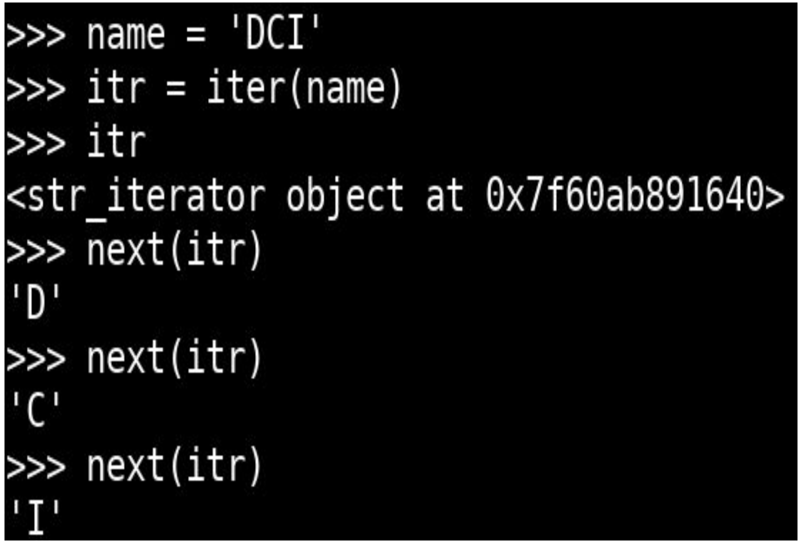
```
>>> iter(True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bool' object is not iterable
>>> iter(234)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
>>> iter(3.45)
Traceback (most recent call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

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.



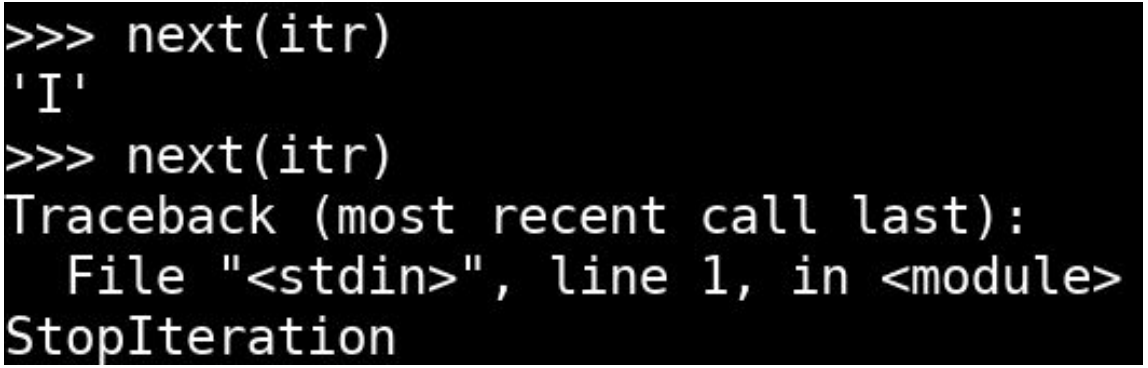
```
>>> name = 'DCI'
>>> itr = iter(name)
>>> itr
<str_iterator object at 0x7f60ab891640>
>>> next(itr)
'D'
>>> next(itr)
'C'
>>> next(itr)
'I'
```

Iterators

- 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**.



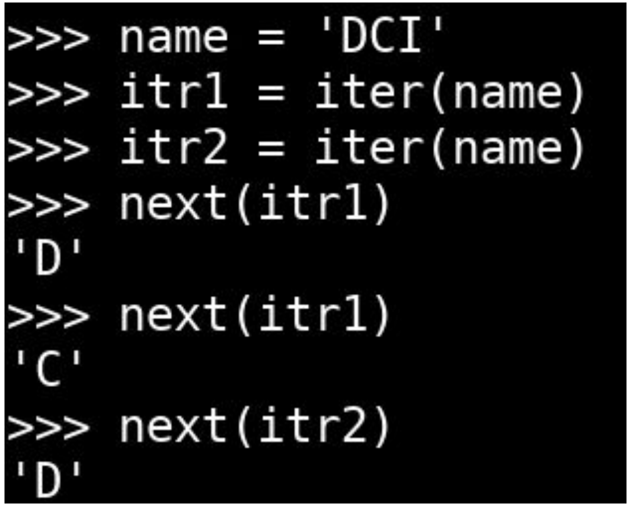
```
>>> next(itr)
'I'
>>> next(itr)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Iterators

- 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

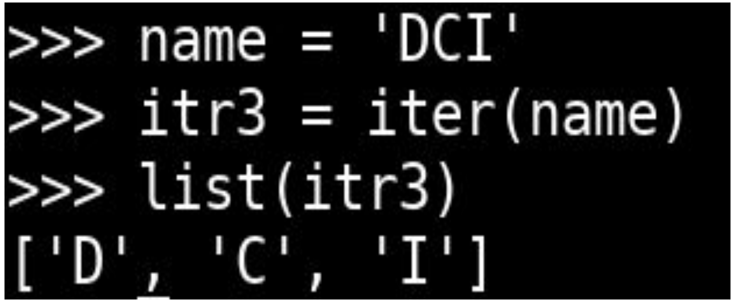
- If you want to grab all the values from an iterator at once, you can use the built-in **list()** function.

- 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”

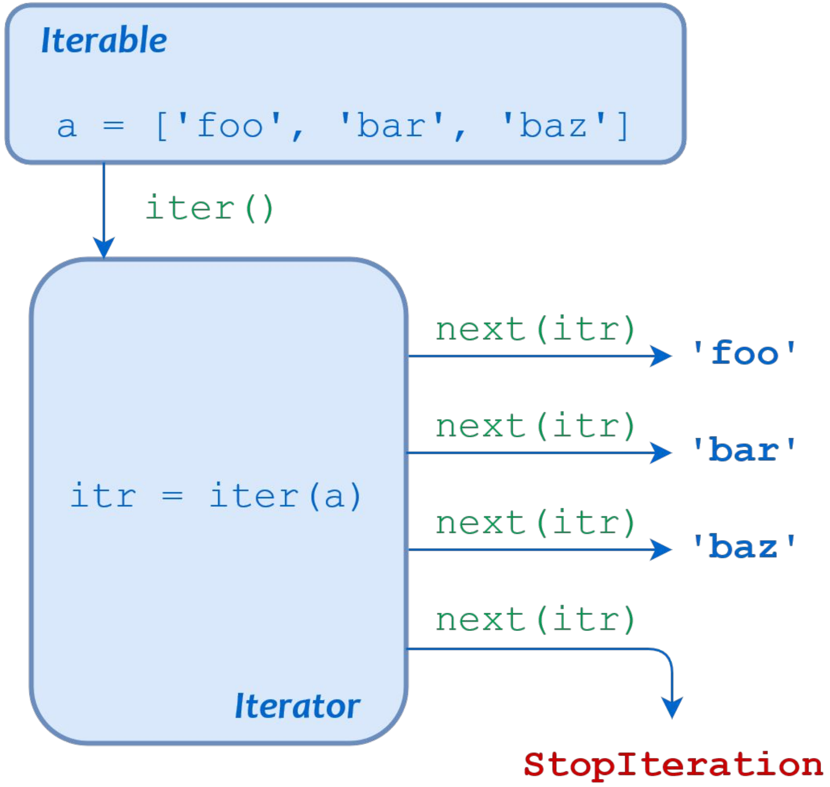
- Part of the elegance of iterators is that they are “**lazy**”
- That means that when you create an iterator, and it doesn’t generate all the items, it can **yield** just them.
- It waits until you ask for them with **next()**.
- Items are not created until they are **requested**.

Iteration in **for** loop

- 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

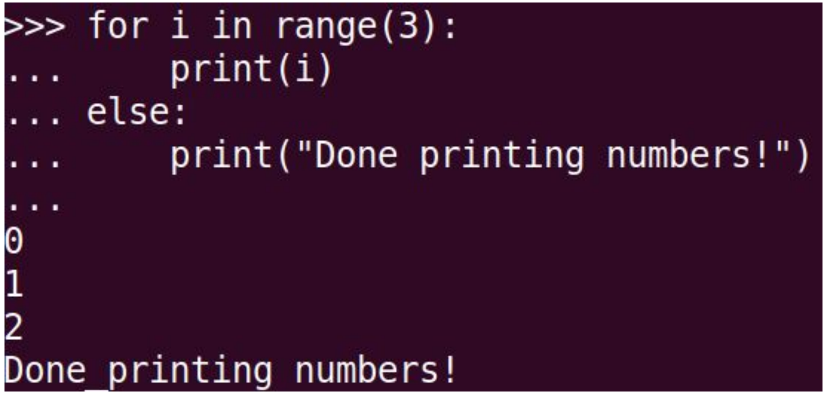
- The loop body is executed **once for each item** next() returns, with loop variable **i** set to the given item for each iteration.

Iteration in **for** loop



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

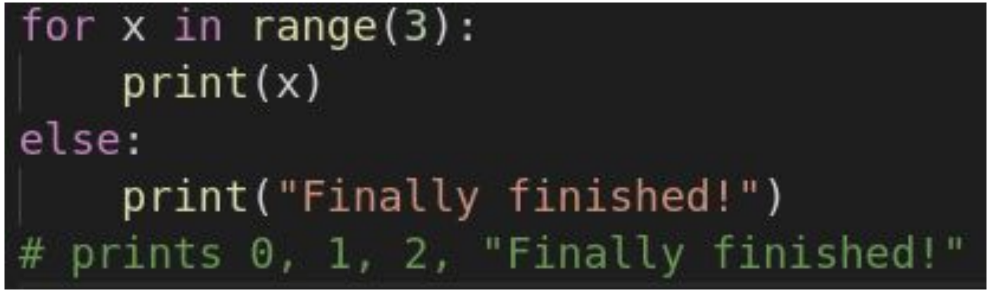
- A for loop can have an **else** clause.
- The else clause will be executed if the loop **terminates** through exhaustion of the iterable:



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

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

- 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!"
```

The **while** loop

While loop

- In Python, **while loops** are used to execute a block of statements repeatedly until a given condition is satisfied.
- Then, the expression is checked again and, if it is **still true**, the body is executed again.
- This continues until the expression becomes **false**.

While loop

- The format of a basic while loop is shown below:

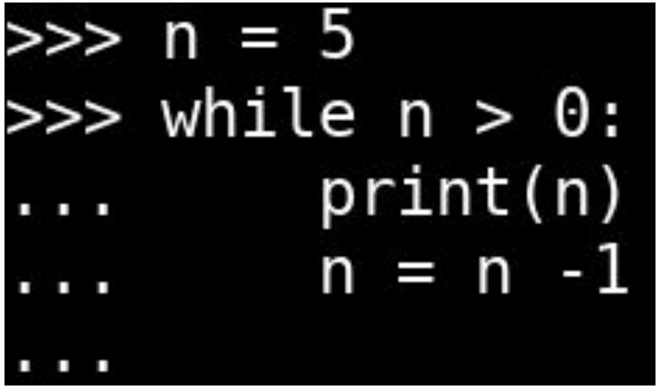
```
while <expr>:
    <statement(s)>
```

- <**statement(s)**> represents the block to be **repeatedly** executed, often referred to as the body of the loop.

- This is denoted with **indentation**

While loop

- The controlling expression, <**expr**>, typically involves one or more variables that are initialized **prior to** starting the loop and then modified somewhere in the loop body:



```
>>> n = 5
>>> while n > 0:
...         print(n)
...         n = n -1
...
```

While loop

- When a while loop is encountered, <**expr**> is first evaluated in **Boolean** context (True or False).

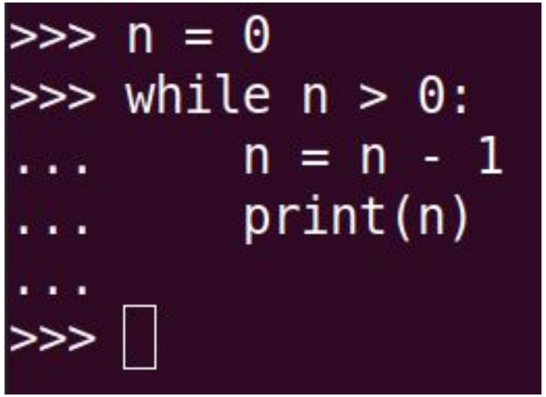
- If it is **true**, the loop body is **executed**.

- Then <expr> is checked **again**, and if still **true**, the body is executed again.

- This continues until <expr> becomes **false**, at which point program execution proceeds to the first statement **beyond** the loop body.

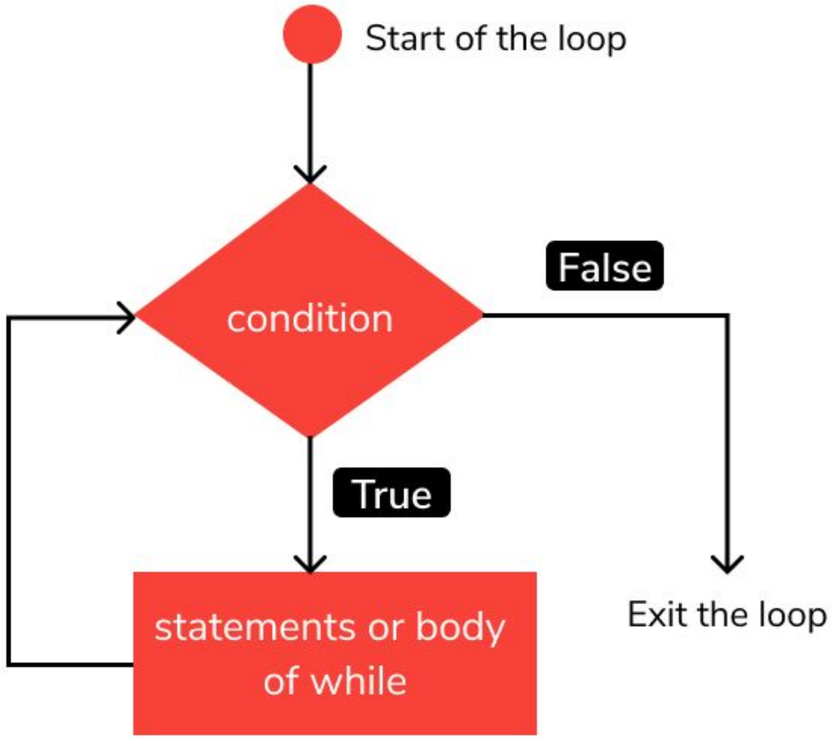
While loop

- Note that the controlling expression of the while loop is tested **first**, before anything else happens.
- If it's false to start with, the loop body will **never** be executed at all:



```
>>> n = 0
>>> while n > 0:
...     n = n - 1
...     print(n)
...
>>> 
```

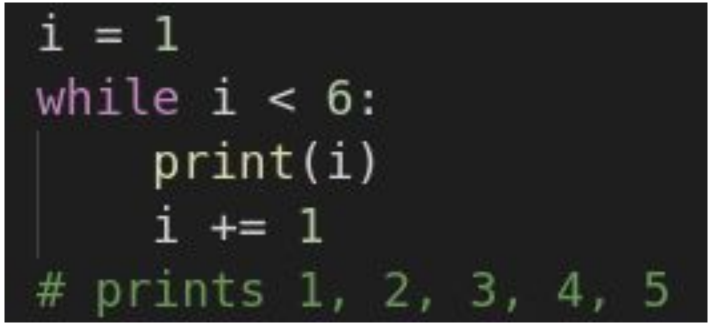
While loop


Start of the loop
condition
False
True
statements or body of while
Exit the loop

While loop - example

- Remember to increment i, or else the loop will continue **forever**.

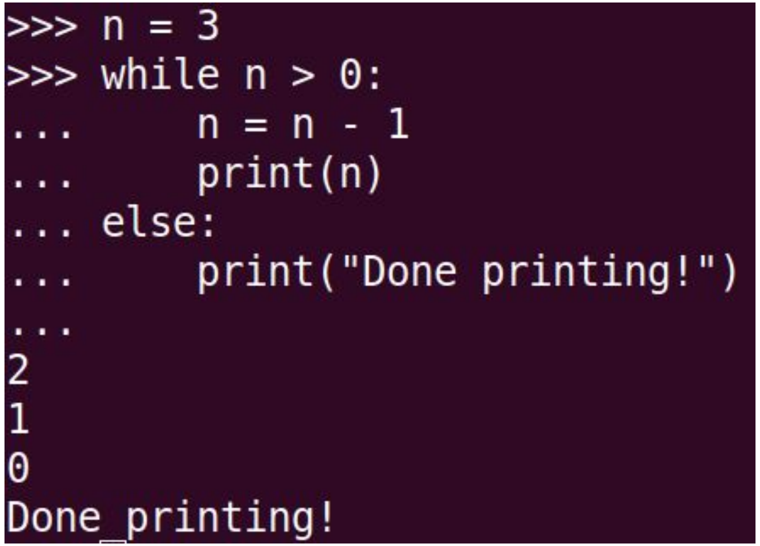
- The while loop **requires** relevant variables to be ready, in this example we need to define an indexing variable **i**, which we set to 1.



```
i = 1
while i < 6:
    print(i)
    i += 1
# prints 1, 2, 3, 4, 5
```

The **else** clause in **while** loop

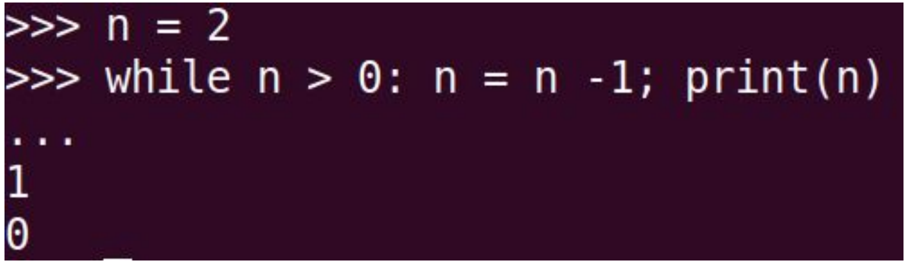
- A while loop can have an **else** clause as well.



```
>>> n = 3
>>> while n > 0:
...     n = n - 1
...     print(n)
... else:
...     print("Done printing!")
...
2
1
0
Done printing!
```

One-line **while** loops

- As with an if statement, a while loop can be specified on **one line**.
- If there are multiple statements in the block that makes up the loop body, they can be separated by semicolons (**;**).



```
>>> n = 2
>>> while n > 0: n = n -1; print(n)
...
1
0
```

One-line **while** loops

- This only works with **simple** statements though.
- You **can’t** combine two compound statements into one line.
- Thus, you can specify a while loop all **on one line** as on previous slide, and you write an if statement **on one line**

THANK YOU

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute