

Python For Loops

[< Previous](#)[Next >](#)

Python For Loops

A **for** loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the **for** keyword in other programming language, and works more like an iterator method as found in other object-orientated programming languages.

With the **for** loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Example

Print each fruit in a fruit list:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

[Run example »](#)

The **for** loop does not require an indexing variable to set beforehand.

Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

Example

Loop through the letters in the word "banana":

```
for x in "banana":  
    print(x)
```

[Run example »](#)

The break Statement

With the **break** statement we can stop the loop before it has looped through all the items:

Example

Exit the loop when **x** is "banana":

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    print(x)  
    if x == "banana":  
        break
```

[Run example »](#)

Example

Exit the loop when **x** is "banana", but this time the break comes before the print:

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    if x == "banana":  
        break  
    print(x)
```

[Run example »](#)

The continue Statement

With the `continue` statement we can stop the current iteration of the loop, and continue with the next:

Example

Do not print banana:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        continue
    print(x)
```

[Run example »](#)

The range() Function

To loop through a set of code a specified number of times, we can use the `range()` function,

The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example

Using the range() function:

```
for x in range(6):
    print(x)
```

[Run example »](#)

Note that `range(6)` is not the values of 0 to 6, but the values 0 to 5.

The `range()` function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: `range(2, 6)`, which means values from 2 to 6 (but not including 6):

Example

Using the start parameter:

```
for x in range(2, 6):  
    print(x)
```

[Run example »](#)

The `range()` function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: `range(2, 30, 3)`:

Example

Increment the sequence with 3 (default is 1):

```
for x in range(2, 30, 3):  
    print(x)
```

[Run example »](#)

Else in For Loop

The `else` keyword in a `for` loop specifies a block of code to be executed when the loop is finished:

Example

Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x in range(6):  
    print(x)
```

```
else:  
    print("Finally finished!")
```

[Run example »](#)

Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

Print each adjective for every fruit:

```
adj = ["red", "big", "tasty"]  
fruits = ["apple", "banana", "cherry"]  
  
for x in adj:  
    for y in fruits:  
        print(x, y)
```

[Run example »](#)

Recursion

Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

In this example, `tri_recursion()` is a function that we have defined to call itself ("recurse"). We use the `k` variable as the data, which decrements (`-1`) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

Example

Recursion Example

```
def tri_recursion(k):
    if(k>0):
        result = k+tri_recursion(k-1)
        print(result)
    else:
        result = 0
    return result

print("\n\nRecursion Example Results")
tri_recursion(6)
```

[Run example »](#)

Test Yourself With Exercises

Exercise:

Loop through the items in the `fruits` list.

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

[Submit Answer »](#)

[Start the Exercise](#)

[◀ Previous](#)

[Next ▶](#)

[Copyright 1999-2018](#) by Refsnes Data. All Rights Reserved.