



# Python Loops

- `while` loops
- `for` loops

## The while Loop

With the `while` loop we can execute a set of statements as long as a condition is true.

### Example

Print i as long as i is less than 6:

```
i = 1
while i < 6:
    print(i)
    i += 1
```

[Try it Yourself »](#)

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.

## The break Statement

With the `break` statement we can stop the loop even if the while condition is true:

### Example

Exit the loop when i is 3:

```
i = 1
while i < 6:
    print(i)
    if i == 3:
        break
    i += 1
```

[Try it Yourself »](#)



# The continue Statement

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

## Example

Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```

[Try it Yourself »](#)

# The else Statement

With the `else` statement we can run a block of code once when the condition no longer is true:

## Example

Print a message once the condition is false:

```
i = 1
while i < 6:
    print(i)
    i += 1
else:
    print("i is no longer less than 6")
```

[Try it Yourself »](#)



# 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 languages, 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)
```

[Try it Yourself »](#)

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)
```

[Try it Yourself »](#)



# 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
```

[Try it Yourself »](#)

## 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)
```

[Try it Yourself »](#)

# 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)
```

[Try it Yourself »](#)

## 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)
```

[Try it Yourself »](#)

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)
```

[Try it Yourself »](#)

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)
```

[Try it Yourself »](#)

## 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!")
```

[Try it Yourself »](#)

**Note:** The `else` block will NOT be executed if the loop is stopped by a `break` statement.

## Example

Break the loop when `x` is 3, and see what happens with the `else` block:



```
for x in range(6):
    if x == 3: break
    print(x)
else:
    print("Finally finished!")
```

[Try it Yourself »](#)

## 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)
```

[Try it Yourself »](#)

## The pass Statement

`for` loops cannot be empty, but if you for some reason have a `for` loop with no content, put in the `pass` statement to avoid getting an error.

### Example

```
for x in [0, 1, 2]:
    pass
```

[Try it Yourself »](#)



# Python Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

## Creating a Function

In Python a function is defined using the `def` keyword:

### Example

```
def my_function():  
    print("Hello from a function")
```

## Calling a Function

To call a function, use the function name followed by parenthesis:

### Example

```
def my_function():  
    print("Hello from a function")
```

```
my_function()
```

[Try it Yourself »](#)



# Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

## Example

```
def my_function(fname):  
    print(fname + " Refsnes")
```

```
my_function("Emil")  
my_function("Tobias")  
my_function("Linus")
```

[Try it Yourself »](#)

*Arguments* are often shortened to *args* in Python documentations.

# Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

## Example

This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):  
    print(fname + " " + lname)
```

```
my_function("Emil", "Refsnes")
```

[Try it Yourself »](#)



If you try to call the function with 1 or 3 arguments, you will get an error:

## Example

This function expects 2 arguments, but gets only 1:

```
def my_function(fname, lname):  
    print(fname + " " + lname)
```

```
my_function("Emil")
```

[Try it Yourself »](#)

## \*args

If you do not know how many arguments that will be passed into your function, add a `*` before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

## Example

If the number of arguments is unknown, add a `*` before the parameter name:

```
def my_function(*kids):  
    print("The youngest child is " + kids[2])
```

```
my_function("Emil", "Tobias", "Linus")
```

[Try it Yourself »](#)

## Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:



## Example

```
def my_function(country = "Norway"):  
    print("I am from " + country)
```

```
my_function("Sweden")  
my_function("India")  
my_function()  
my_function("Brazil")
```

[Try it Yourself »](#)

## Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

## Example

```
def my_function(food):  
    for x in food:  
        print(x)
```

```
fruits = ["apple", "banana", "cherry"]
```

```
my_function(fruits)
```

[Try it Yourself »](#)



# Return Values

To let a function return a value, use the `return` statement:

## Example

```
def my_function(x):  
    return 5 * x
```

```
print(my_function(3))  
print(my_function(5))  
print(my_function(9))
```

[Try it Yourself »](#)



Python  
PYTHON

shajan mohammed  
SHAJAN MOHAMMED