# **Python If ... Else**

## **Python Conditions and If statements**

Python supports the usual logical conditions from mathematics:

* Equals: a == b
* Not Equals: a != b
* Less than: a < b
* Less than or equal to: a <= b
* Greater than: a > b
* Greater than or equal to: a >= b

These conditions can be used in several ways, most commonly in "if statements" and loops.

An "if statement" is written by using the if keyword.

### **Example**[**Get your own Python Server**](https://www.w3schools.com/python/python_server.asp)

If statement:

a = 33

b = 200

if b > a:

print("b is greater than a")

In this example we use two variables, a and b, which are used as part of the if statement to test whether b is greater than a. As a is 33, and b is 200, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".

## **Indentation**

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

### **Example**

If statement, without indentation (will raise an error):

a = 33

b = 200

if b > a:

print("b is greater than a") # you will get an error

## **Elif**

The elif keyword is Python's way of saying "if the previous conditions were not true, then try this condition".

### **Example**

a = 33

b = 33

if b > a:

print("b is greater than a")

elif a == b:

print("a and b are equal")

In this example a is equal to b, so the first condition is not true, but the elif condition is true, so we print to screen that "a and b are equal".

## **Else**

The else keyword catches anything which isn't caught by the preceding conditions.

### **Example**

a = 200

b = 33

if b > a:

print("b is greater than a")

elif a == b:

print("a and b are equal")

else:

print("a is greater than b")

In this example a is greater than b, so the first condition is not true, also the elif condition is not true, so we go to the else condition and print to screen that "a is greater than b".

You can also have an else without the elif:

### **Example**

a = 200

b = 33

if b > a:

print("b is greater than a")

else:

print("b is not greater than a")

## **Short Hand If**

If you have only one statement to execute, you can put it on the same line as the if statement.

### **Example**

One line if statement:

if a > b: print("a is greater than b")

## **Short Hand If ... Else**

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

### **Example**

One line if else statement:

a = 2

b = 330

print("A") if a > b else print("B")

This technique is known as **Ternary Operators**, or **Conditional Expressions**.

You can also have multiple else statements on the same line:

### **Example**

One line if else statement, with 3 conditions:

a = 330

b = 330

print("A") if a > b else print("=") if a == b else print("B")

## **And**

The and keyword is a logical operator, and is used to combine conditional statements:

### **Example**

Test if a is greater than b, AND if c is greater than a:

a = 200

b = 33

c = 500

if a > b and c > a:

print("Both conditions are True")

## **Or**

The or keyword is a logical operator, and is used to combine conditional statements:

### **Example**

Test if a is greater than b, OR if a is greater than c:

a = 200

b = 33

c = 500

if a > b or a > c:

print("At least one of the conditions is True")

## **Not**

The not keyword is a logical operator, and is used to reverse the result of the conditional statement:

### **Example**

Test if a is NOT greater than b:

a = 33

b = 200

if not a > b:

print("a is NOT greater than b")

## **Nested If**

You can have if statements inside if statements, this is called *nested* if statements.

### **Example**

x = 41

if x > 10:

print("Above ten,")

if x > 20:

print("and also above 20!")

else:

print("but not above 20.")

## **The pass Statement**

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

### **Example**

a = 33

b = 200

if b > a:

pass

## **Test Yourself With Exercises**

## **Exercise:**

Print "Hello World" if a is greater than b.

a = 50

b = 10

a b

print("Hello World")

[Start the Exercise](https://www.w3schools.com/python/exercise.asp?filename=exercise_ifelse1)

# **Python Lists**

mylist = ["apple", "banana", "cherry"]

## **List**

Lists are used to store multiple items in a single variable.

Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are [Tuple](https://www.w3schools.com/python/python_tuples.asp), [Set](https://www.w3schools.com/python/python_sets.asp), and [Dictionary](https://www.w3schools.com/python/python_dictionaries.asp), all with different qualities and usage.

Lists are created using square brackets:

### **Example**[**Get your own Python Server**](https://www.w3schools.com/python/python_server.asp)

Create a List:

thislist = ["apple", "banana", "cherry"]

print(thislist)

## **List Items**

List items are ordered, changeable, and allow duplicate values.

List items are indexed, the first item has index [0], the second item has index [1] etc.

## **Ordered**

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

**Note:** There are some [list methods](https://www.w3schools.com/python/python_lists_methods.asp) that will change the order, but in general: the order of the items will not change.

## **Changeable**

The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

## **Allow Duplicates**

Since lists are indexed, lists can have items with the same value:

### **Example**

Lists allow duplicate values:

thislist = ["apple", "banana", "cherry", "apple", "cherry"]

print(thislist)

## **List Length**

To determine how many items a list has, use the len() function:

### **Example**

Print the number of items in the list:

thislist = ["apple", "banana", "cherry"]

print(len(thislist))

## **List Items - Data Types**

List items can be of any data type:

### **Example**

String, int and boolean data types:

list1 = ["apple", "banana", "cherry"]

list2 = [1, 5, 7, 9, 3]

list3 = [True, False, False]

A list can contain different data types:

### **Example**

A list with strings, integers and boolean values:

list1 = ["abc", 34, True, 40, "male"]

## **type()**

From Python's perspective, lists are defined as objects with the data type 'list':

<class 'list'>

### **Example**

What is the data type of a list?

mylist = ["apple", "banana", "cherry"]

print(type(mylist))

## **The list() Constructor**

It is also possible to use the list() constructor when creating a new list.

### **Example**

Using the list() constructor to make a List:

thislist = list(("apple", "banana", "cherry")) # note the double round-brackets

print(thislist)

## **Python Collections (Arrays)**

There are four collection data types in the Python programming language:

* **List** is a collection which is ordered and changeable. Allows duplicate members.
* [**Tuple**](https://www.w3schools.com/python/python_tuples.asp) is a collection which is ordered and unchangeable. Allows duplicate members.
* [**Set**](https://www.w3schools.com/python/python_sets.asp) is a collection which is unordered, unchangeable\*, and unindexed. No duplicate members.
* [**Dictionary**](https://www.w3schools.com/python/python_dictionaries.asp) is a collection which is ordered\*\* and changeable. No duplicate members.

\*Set *items* are unchangeable, but you can remove and/or add items whenever you like.

\*\*As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

# **Python Dictionaries**

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

## **Dictionary**

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered\*, changeable and do not allow duplicates.

As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

Dictionaries are written with curly brackets, and have keys and values:

### **Example**[**Get your own Python Server**](https://www.w3schools.com/python/python_server.asp)

Create and print a dictionary:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

print(thisdict)

## **Dictionary Items**

Dictionary items are ordered, changeable, and do not allow duplicates.

Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

### **Example**

Print the "brand" value of the dictionary:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

print(thisdict["brand"])

## **Ordered or Unordered?**

As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.

Unordered means that the items do not have a defined order, you cannot refer to an item by using an index.

## **Changeable**

Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

## **Duplicates Not Allowed**

Dictionaries cannot have two items with the same key:

### **Example**

Duplicate values will overwrite existing values:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964,

"year": 2020

}

print(thisdict)

## **Dictionary Length**

To determine how many items a dictionary has, use the len() function:

### **Example**

Print the number of items in the dictionary:

print(len(thisdict))

## **Dictionary Items - Data Types**

The values in dictionary items can be of any data type:

### **Example**

String, int, boolean, and list data types:

thisdict = {

"brand": "Ford",

"electric": False,

"year": 1964,

"colors": ["red", "white", "blue"]

}

## **type()**

From Python's perspective, dictionaries are defined as objects with the data type 'dict':

<class 'dict'>

### **Example**

Print the data type of a dictionary:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

print(type(thisdict))

## **The dict() Constructor**

It is also possible to use the dict() constructor to make a dictionary.

### **Example**

Using the dict() method to make a dictionary:

thisdict = dict(name = "John", age = 36, country = "Norway")

print(thisdict)

## **Python Collections (Arrays)**

There are four collection data types in the Python programming language:

* [**List**](https://www.w3schools.com/python/python_lists.asp) is a collection which is ordered and changeable. Allows duplicate members.
* [**Tuple**](https://www.w3schools.com/python/python_tuples.asp) is a collection which is ordered and unchangeable. Allows duplicate members.
* [**Set**](https://www.w3schools.com/python/python_sets.asp) is a collection which is unordered, unchangeable\*, and unindexed. No duplicate members.
* **Dictionary** is a collection which is ordered\*\* and changeable. No duplicate members.

\*Set *items* are unchangeable, but you can remove and/or add items whenever you like.

\*\*As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

# **Python While Loops**

## **Python Loops**

Python has two primitive loop commands:

* 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**[**Get your own Python Server**](https://www.w3schools.com/python/python_server.asp)

Print i as long as i is less than 6:

i = 1

while i < 6:

print(i)

i += 1

**Note:** 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.

## **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

## **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)

## **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")

## **Test Yourself With Exercises**

## **Exercise:**

Print i as long as i is less than 6.

i = 1

i < 6

print(i)

i += 1

[Start the Exercise](https://www.w3schools.com/python/exercise.asp?filename=exercise_while_loops1)

# **Python For Loops**

## **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**[**Get your own Python Server**](https://www.w3schools.com/python/python_server.asp)

Print each fruit in a fruit list:

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

for x in fruits:

print(x)

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)

## **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

### **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)

## **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)

## **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)

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)

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)

## **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!")

**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!")

## **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)

## **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

## **Test Yourself With Exercises**

## **Exercise:**

Loop through the items in the fruits list.

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

x fruits

print(x)

[Start the Exercise](https://www.w3schools.com/python/exercise.asp?filename=exercise_for_loops1)

# **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**[**Get your own Python Server**](https://www.w3schools.com/python/python_server.asp)

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()**

## **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"**)

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

## **Parameters or Arguments?**

The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

## **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")

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")

## **Arbitrary Arguments, \*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")

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

## **Keyword Arguments**

You can also send arguments with the *key* = *value* syntax.

This way the order of the arguments does not matter.

### **Example**

def my\_function(child3, child2, child1):

print("The youngest child is " + child3)

my\_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")

The phrase *Keyword Arguments* are often shortened to *kwargs* in Python documentations.

## **Arbitrary Keyword Arguments, \*\*kwargs**

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: \*\* before the parameter name in the function definition.

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

### **Example**

If the number of keyword arguments is unknown, add a double \*\* before the parameter name:

def my\_function(\*\*kid):

print("His last name is " + kid["lname"])

my\_function(fname = "Tobias", lname = "Refsnes")

*Arbitrary Kword Arguments* are often shortened to *\*\*kwargs* in Python documentations.

## **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")

## **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)

## **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))

## **The pass Statement**

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

### **Example**

def myfunction():

pass

## **Positional-Only Arguments**

You can specify that a function can have ONLY positional arguments, or ONLY keyword arguments.

To specify that a function can have only positional arguments, add , / after the arguments:

### **Example**

def my\_function(x, /):

print(x)

my\_function(3)

Without the , / you are actually allowed to use keyword arguments even if the function expects positional arguments:

### **Example**

def my\_function(x):

print(x)

my\_function(x = 3)

But when adding the , / you will get an error if you try to send a keyword argument:

### **Example**

def my\_function(x, /):

print(x)

my\_function(x = 3)

## **Keyword-Only Arguments**

To specify that a function can have only keyword arguments, add \*, *before* the arguments:

### **Example**

def my\_function(\*, x):

print(x)

my\_function(x = 3)

Without the \*, you are allowed to use positionale arguments even if the function expects keyword arguments:

### **Example**

def my\_function(x):

print(x)

my\_function(3)

But when adding the \*, / you will get an error if you try to send a positional argument:

### **Example**

def my\_function(\*, x):

print(x)

my\_function(3)

## **Combine Positional-Only and Keyword-Only**

You can combine the two argument types in the same function.

Any argument *before* the / , are positional-only, and any argument *after* the \*, are keyword-only.

### **Example**

def my\_function(a, b, /, \*, c, d):

print(a + b + c + d)

my\_function(5, 6, c = 7, d = 8)

## **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)

## **Test Yourself With Exercises**

## **Exercise:**

Create a function named my\_function.

:

print("Hello from a function")

[Start the Exercise](https://www.w3schools.com/python/exercise.asp?filename=exercise_functions1)