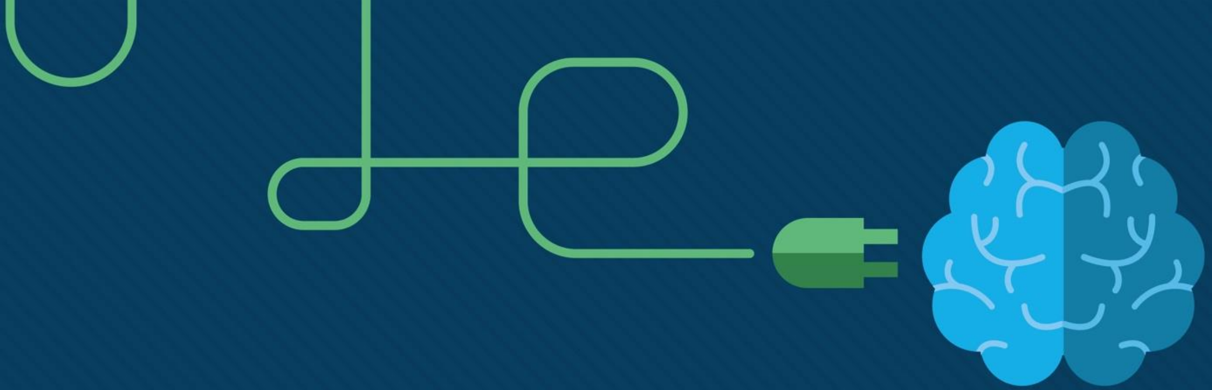


Programming Essentials in Python

Programming Essentials in
Python

Module 3: Boolean values, conditional execution, loops, lists and list processing, logical and bitwise operations

Module Objectives

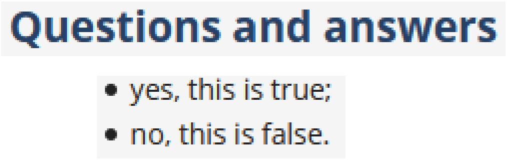
Module Title: Boolean values, conditional execution, loops, lists and list processing, logical and bitwise operations

Objectives
Boolean values;
if-elif-else instructions;
the while and for loops;
flow control;
logical and bitwise operations;
lists and arrays.

Boolean values
Comparison operators

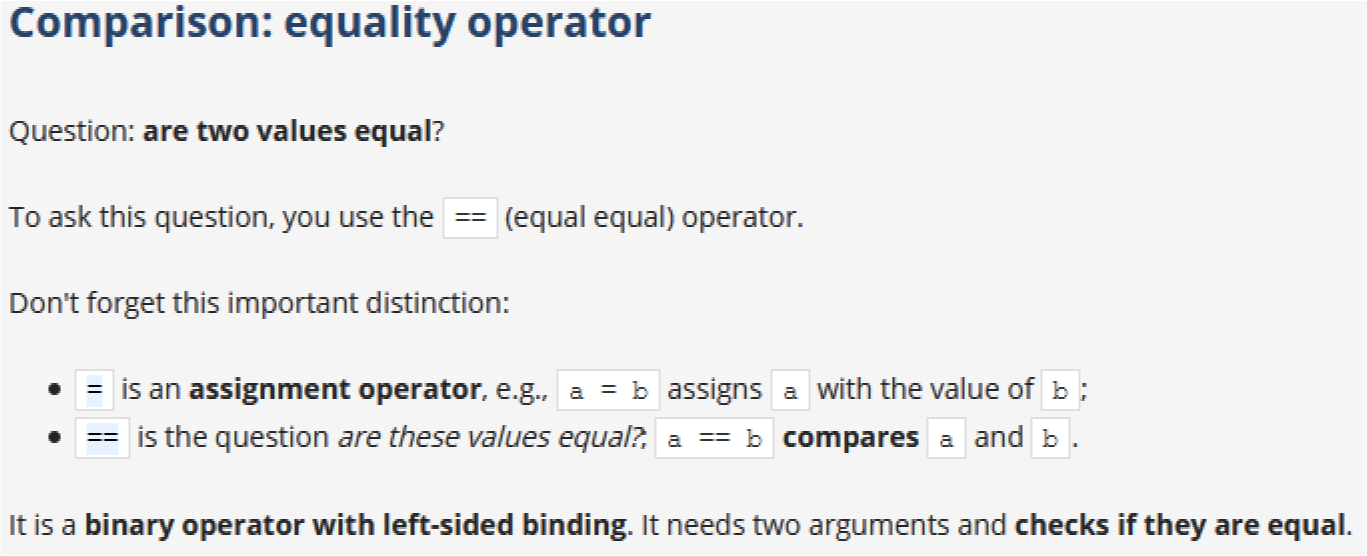
Boolean

- Can only be True of False



Questions and answers

- yes, this is true;
- no, this is false.



Comparison: equality operator

Question: **are two values equal**?

To ask this question, you use the `==` (equal equal) operator.

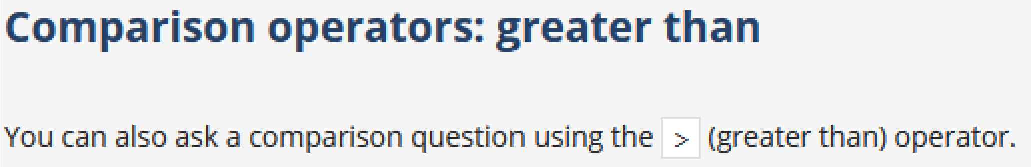
Don't forget this important distinction:

- `=` is an **assignment operator**, e.g., `a = b` assigns `a` with the value of `b`;
- `==` is the question *are these values equal?*; `a == b` **compares** `a` and `b`.

It is a **binary operator with left-sided binding**. It needs two arguments and **checks if they are equal**.

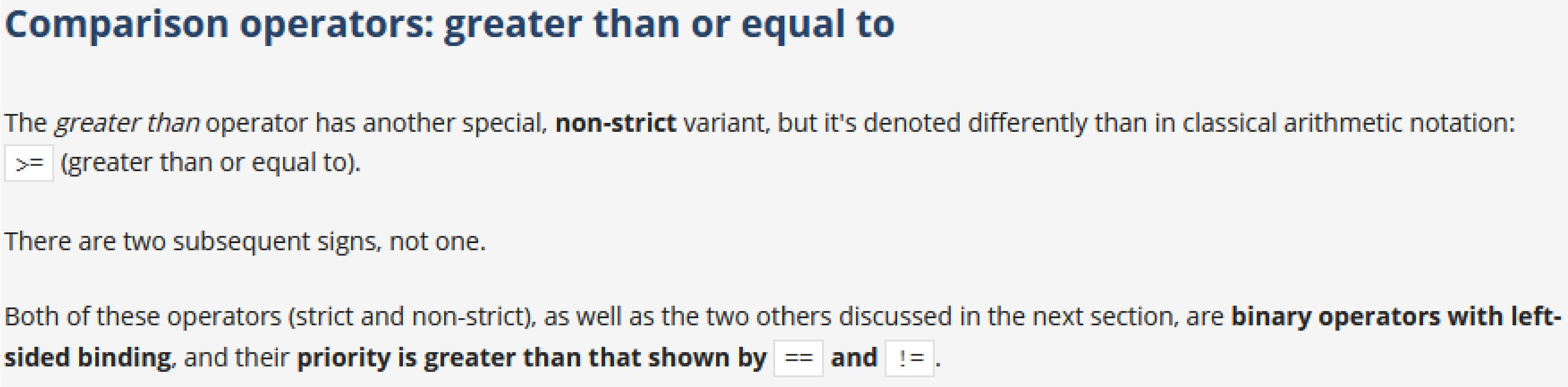
Boolean values

Comparison operators



Comparison operators: greater than

You can also ask a comparison question using the `>` (greater than) operator.

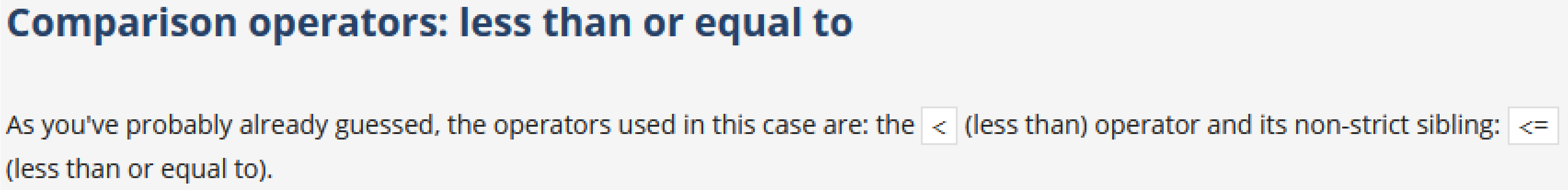


Comparison operators: greater than or equal to

The *greater than* operator has another special, **non-strict** variant, but it's denoted differently than in classical arithmetic notation: `>=` (greater than or equal to).

There are two subsequent signs, not one.

Both of these operators (strict and non-strict), as well as the two others discussed in the next section, are **binary operators with left-sided binding**, and their **priority is greater than that shown by** `==` **and** `!=`.

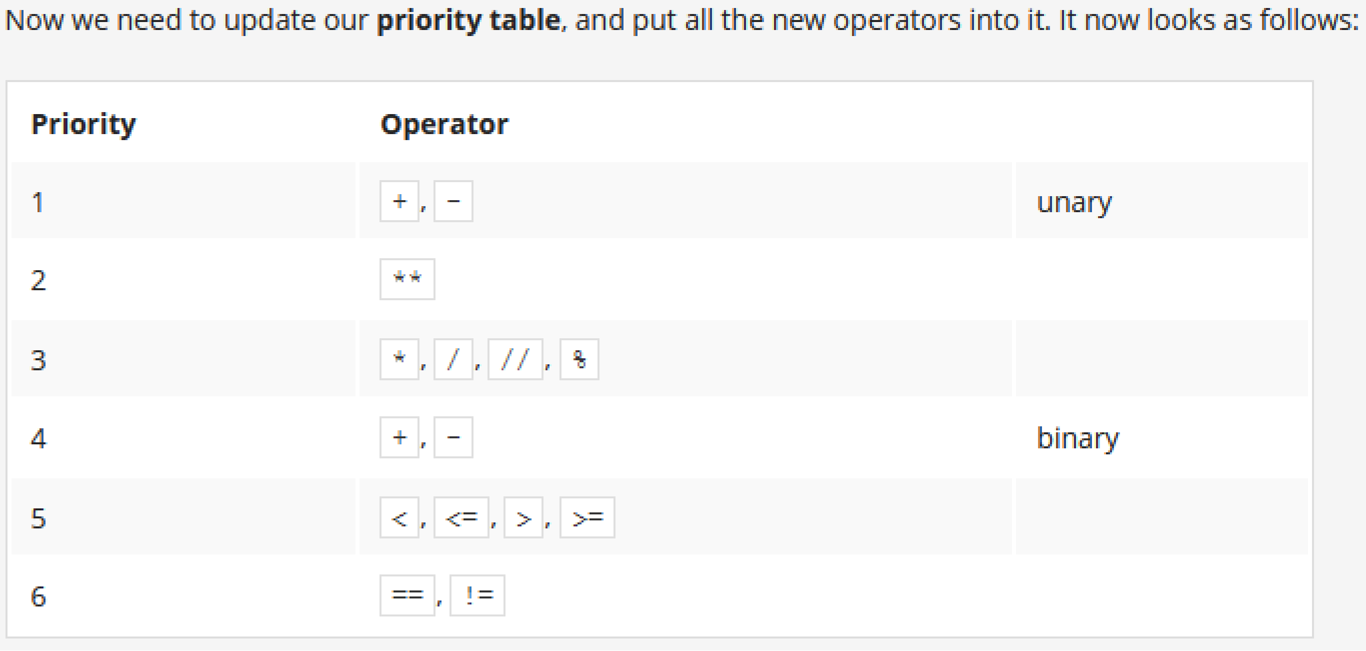


Comparison operators: less than or equal to

As you've probably already guessed, the operators used in this case are: the `<` (less than) operator and its non-strict sibling: `<=` (less than or equal to).

Boolean values

Comparison operators



Now we need to update our **priority table**, and put all the new operators into it. It now looks as follows:

Priority	Operator	
1	`+`, `-`	unary
2	`**`	
3	`*`, `/`, `//`, `%`	
4	`+`, `-`	binary
5	`<`, `<=`, `>`, `>=`	
6	`==`, `!=`	

Boolean values

Comparison operators - Comparators

Operator	Description
==	Equal to
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
!=	Not equal

Boolean values

Comparison operators – Logic Operators

Operator	Description
and	Evaluates to True if both statements are true, otherwise evaluates to False.
or	Evaluates to True if either of the statements is true, otherwise evaluates to False.
not	Evaluates to the opposite of the statement.

Order of Operations of Booleans
The order of operations for Boolean algebra, from highest to lowest priority is **NOT**, then **AND**, then **OR**. Expressions inside brackets are always evaluated first.

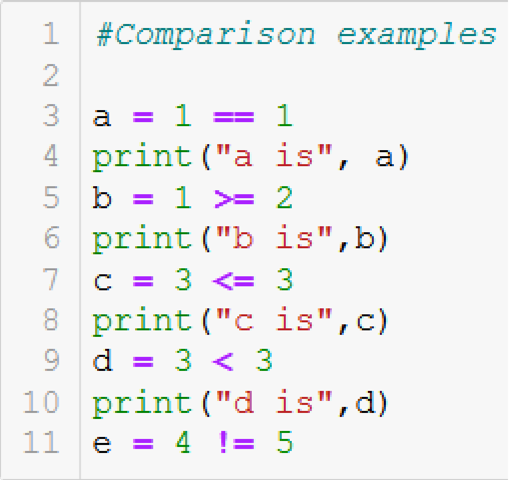
Boolean values

Comparison operators - Boolean Operators

Truth Table	
True and True is True True and False is False False and True is False False and False is False	True or True is True True or False is True False or True is True False or False is False
Not True is False Not False is True	

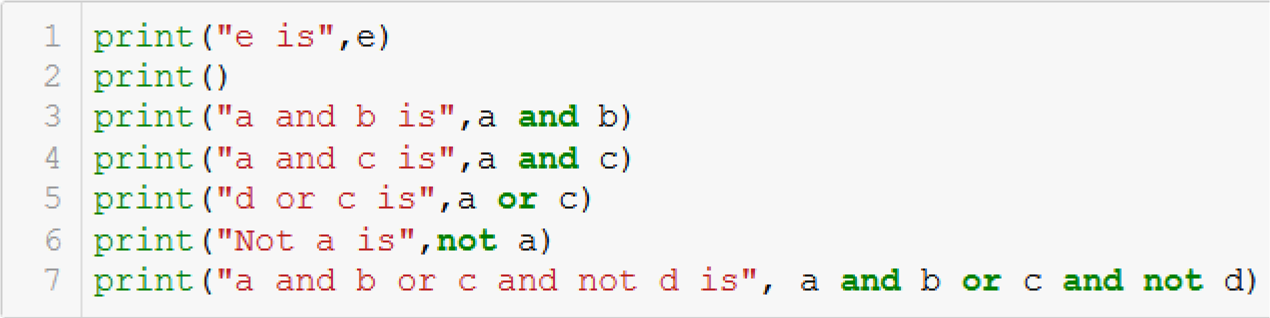
Boolean values

Comparison operators - Examples



```python
#Comparison examples

a = 1 == 1
print("a is", a)
b = 1 >= 2
print("b is",b)
c = 3 <= 3
print("c is",c)
d = 3 < 3
print("d is",d)
e = 4 != 5
```

```
a is True
b is False
c is True
d is False
```



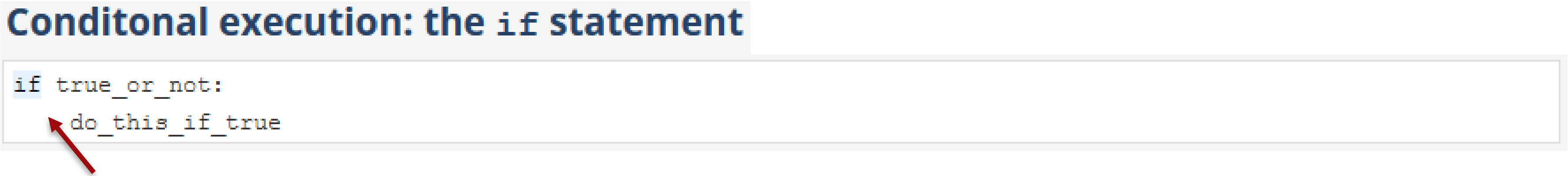
```python
print("e is",e)
print()
print("a and b is",a and b)
print("a and c is",a and c)
print("d or c is",a or c)
print("Not a is",not a)
print("a and b or c and not d is", a and b or c and not d)
```

```
e is True

a and b is False
a and c is True
d or c is True
Not a is False
a and b or c and not d is True
```

Demo
Boolean values and comparison

Conditionals
if, if-else, if-elif-else

Conditional execution
if-elif-else instructions

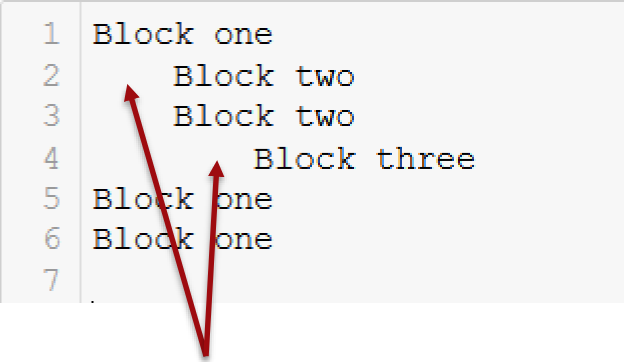


Conditonal execution: the `if` statement

```
if true_or_not:
    do_this_if_true
```

Indentation

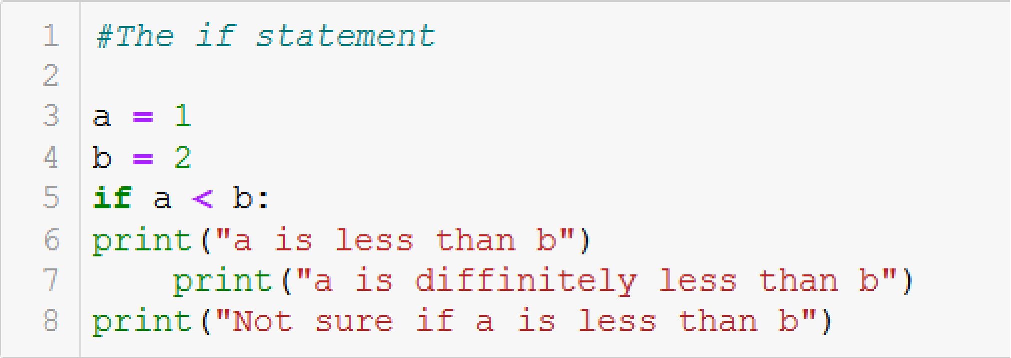
Code Blocks



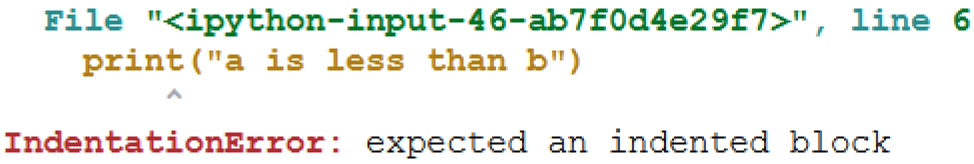
```
Block one
    Block two
    Block two
        Block three
Block one
Block one
```

4 spaces or single tab

Spacing problems



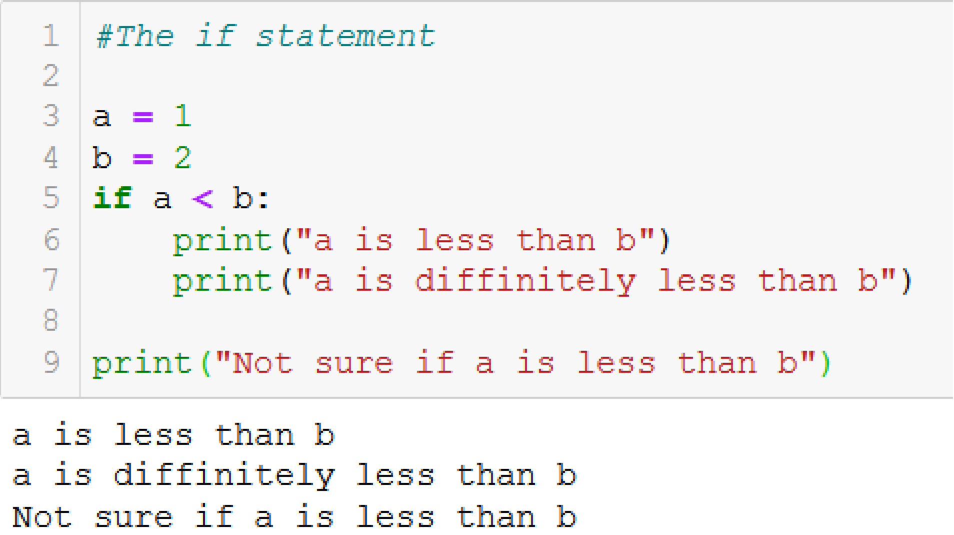
```
1  #The if statement
2
3  a = 1
4  b = 2
5  if a < b:
6  print("a is less than b")
7      print("a is diffinitely less than b")
8  print("Not sure if a is less than b")
```



```
  File "<ipython-input-46-ab7f0d4e29f7>", line 6
    print("a is less than b")
        ^
IndentationError: expected an indented block
```

Conditional execution
if-elif-else instructions

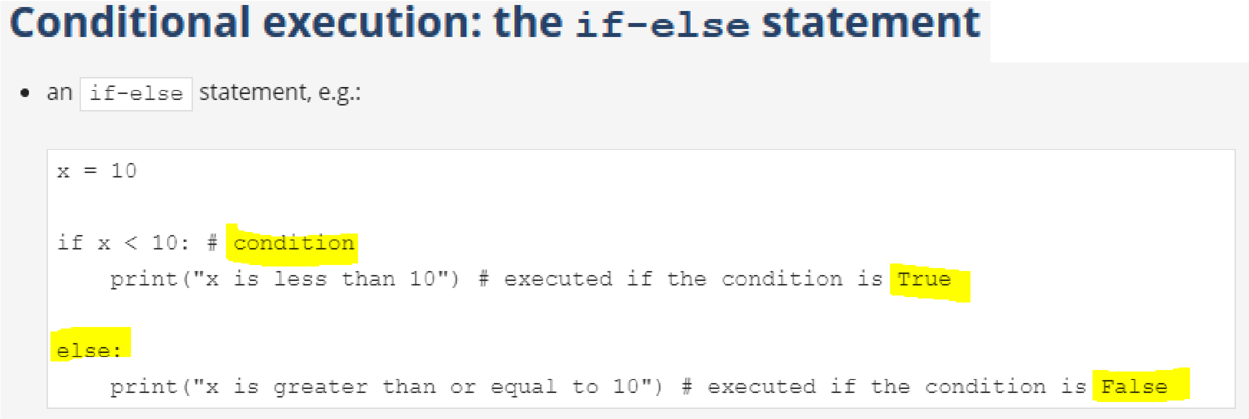
Example:



```python
#The if statement

a = 1
b = 2
if a < b:
    print("a is less than b")
    print("a is diffinitely less than b")

print("Not sure if a is less than b")
```

```
a is less than b
a is diffinitely less than b
Not sure if a is less than b
```

Conditional execution
if-elif-else instructions



Conditional execution: the `if-else` statement

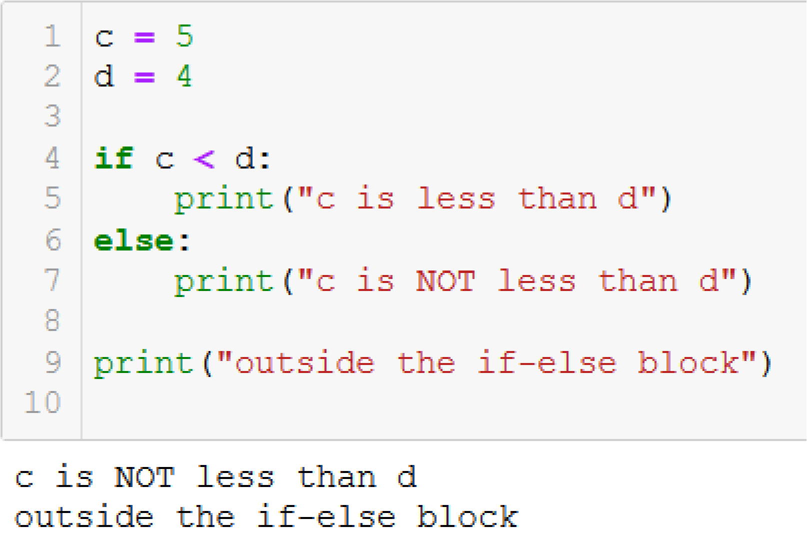
- an `if-else` statement, e.g.:

```
x = 10

if x < 10: # condition
    print("x is less than 10") # executed if the condition is True

else:
    print("x is greater than or equal to 10") # executed if the condition is False
```

Conditional execution
if-elif-else instructions

Conditional execution: the if-else statement

Example:

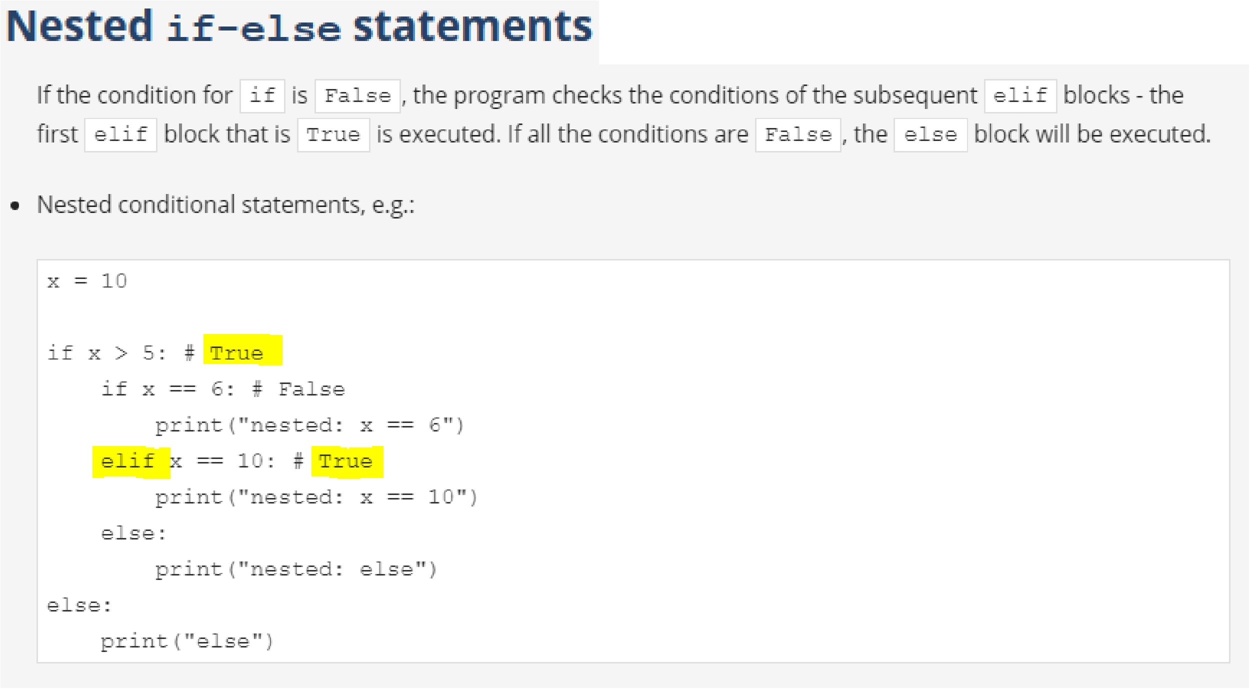


```
c = 5
d = 4

if c < d:
    print("c is less than d")
else:
    print("c is NOT less than d")

print("outside the if-else block")

```

```
c is NOT less than d
outside the if-else block
```

Conditional execution

if-elif-else instructions



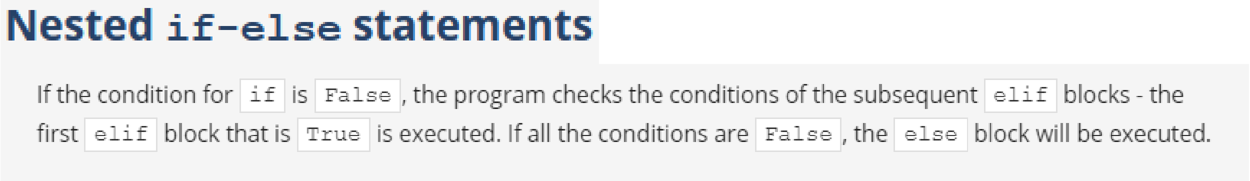
Nested `if-else` statements

If the condition for `if` is `False`, the program checks the conditions of the subsequent `elif` blocks - the first `elif` block that is `True` is executed. If all the conditions are `False`, the `else` block will be executed.

- Nested conditional statements, e.g.:

```
x = 10

if x > 5: # True
    if x == 6: # False
        print("nested: x == 6")
    elif x == 10: # True
        print("nested: x == 10")
    else:
        print("nested: else")
else:
    print("else")
```

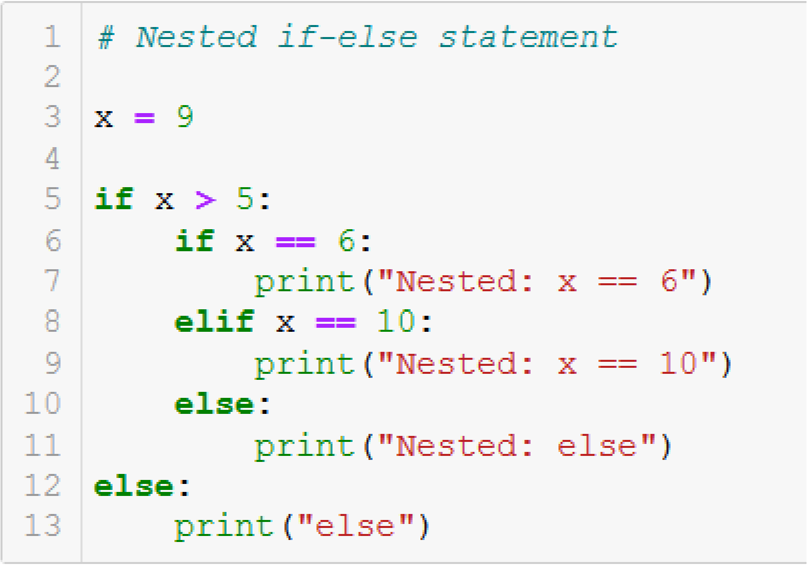
Conditional execution
if-elif-else instructions



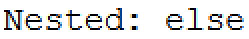
Nested `if-else` statements

If the condition for `if` is `False`, the program checks the conditions of the subsequent `elif` blocks - the first `elif` block that is `True` is executed. If all the conditions are `False`, the `else` block will be executed.

Example:



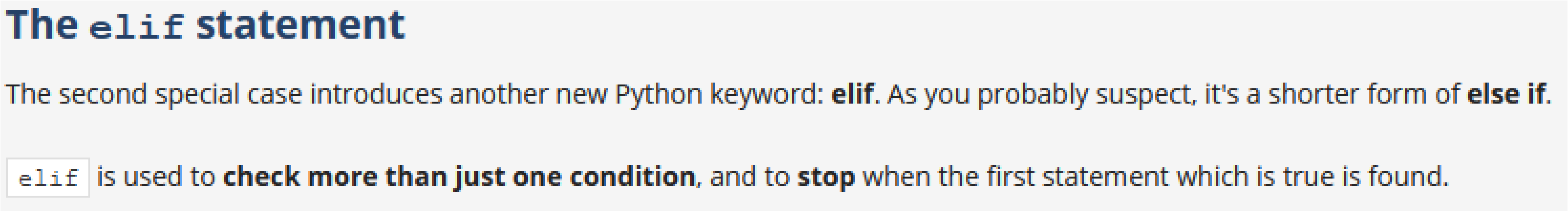
```python
# Nested if-else statement

x = 9

if x > 5:
    if x == 6:
        print("Nested: x == 6")
    elif x == 10:
        print("Nested: x == 10")
    else:
        print("Nested: else")
else:
    print("else")
```



```
Nested: else
```

Conditional execution

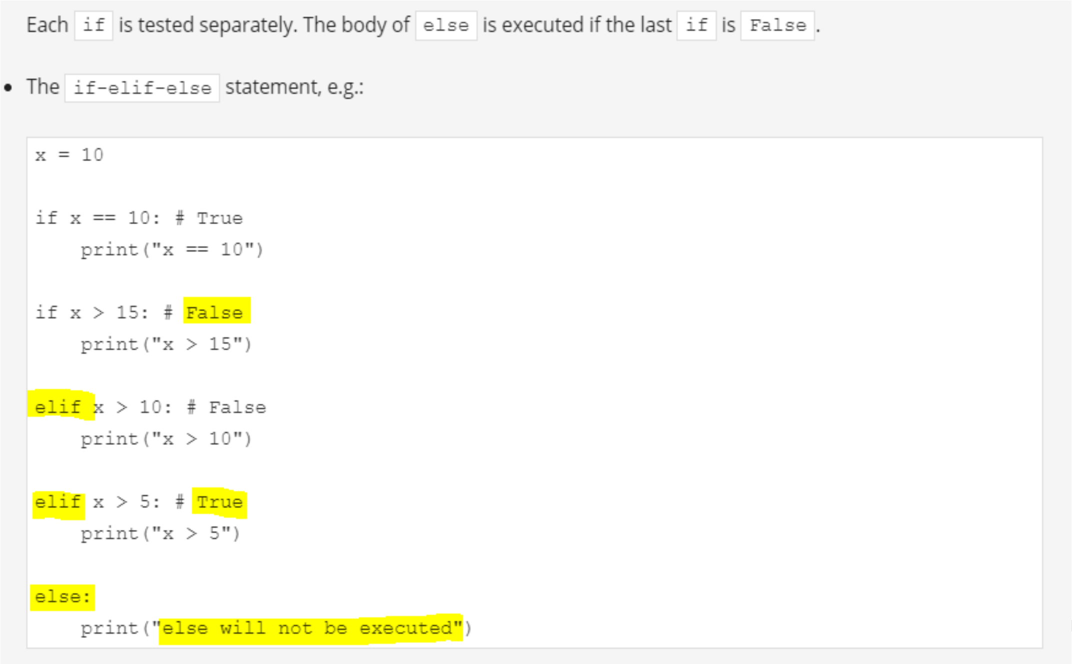
if-elif-else instructions



The `elif` statement

The second special case introduces another new Python keyword: **elif**. As you probably suspect, it's a shorter form of **else if**.

`elif` is used to **check more than just one condition**, and to **stop** when the first statement which is true is found.



Each `if` is tested separately. The body of `else` is executed if the last `if` is `False`.

- The `if-elif-else` statement, e.g.:

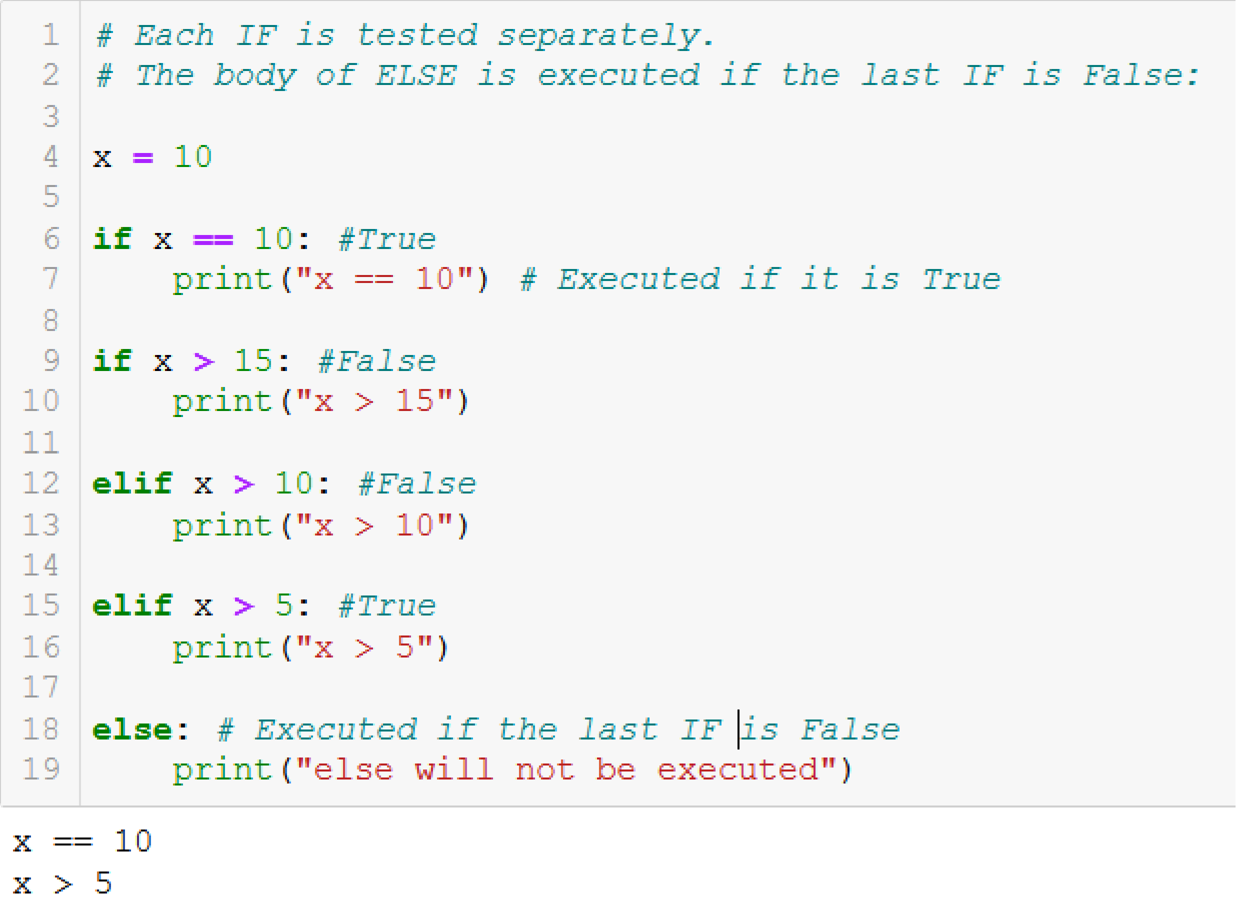
```
x = 10

if x == 10: # True
    print("x == 10")

if x > 15: # False
    print("x > 15")

elif x > 10: # False
    print("x > 10")

elif x > 5: # True
    print("x > 5")

else:
    print("else will not be executed")
```

Conditional execution

if-elif-else instructions

The `elif` statement

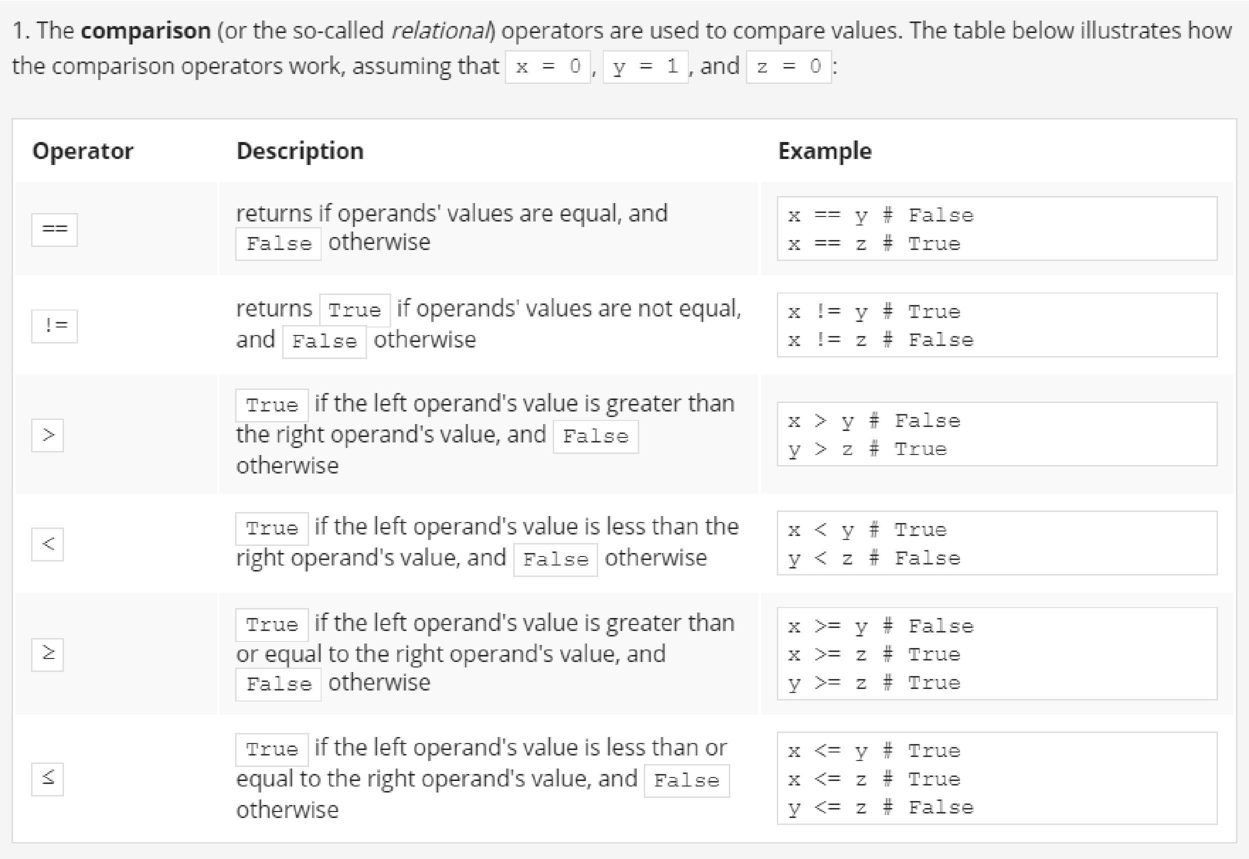
Example:



```python
# Each IF is tested separately.
# The body of ELSE is executed if the last IF is False:

x = 10

if x == 10: #True
    print("x == 10") # Executed if it is True

if x > 15: #False
    print("x > 15")

elif x > 10: #False
    print("x > 10")

elif x > 5: #True
    print("x > 5")

else: # Executed if the last IF is False
    print("else will not be executed")
```

```
x == 10
x > 5
```

Conditional execution
if-elif-else instructions



1. The **comparison** (or the so-called *relational*) operators are used to compare values. The table below illustrates how the comparison operators work, assuming that `x = 0`, `y = 1`, and `z = 0`:

Operator	Description	Example

`==`	returns if operands' values are equal, and `False` otherwise	`x == y # False`
		`x == z # True`

`!=`	returns `True` if operands' values are not equal, and `False` otherwise	`x != y # True`
		`x != z # False`

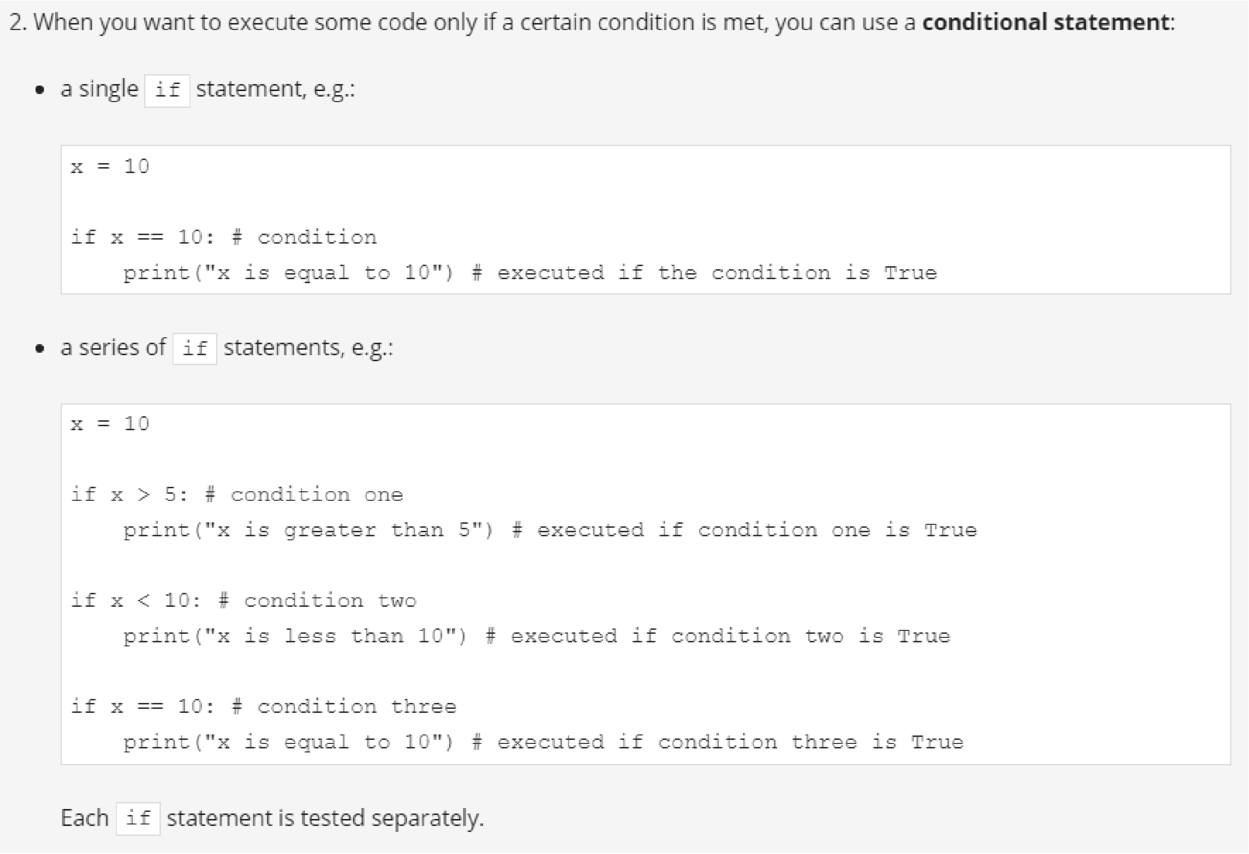
`>`	`True` if the left operand's value is greater than the right operand's value, and `False` otherwise	`x > y # False`
		`y > z # True`

`<`	`True` if the left operand's value is less than the right operand's value, and `False` otherwise	`x < y # True`
		`y < z # False`

`≥`	`True` if the left operand's value is greater than or equal to the right operand's value, and `False` otherwise	`x >= y # False`
		`x >= z # True`
		`y >= z # True`

`≤`	`True` if the left operand's value is less than or equal to the right operand's value, and `False` otherwise	`x <= y # True`
		`x <= z # True`
		`y <= z # False`

Conditional execution
if-elif-else instructions



2. When you want to execute some code only if a certain condition is met, you can use a **conditional statement**:

- a single `if` statement, e.g.:

```
x = 10

if x == 10: # condition
    print("x is equal to 10") # executed if the condition is True
```

- a series of `if` statements, e.g.:

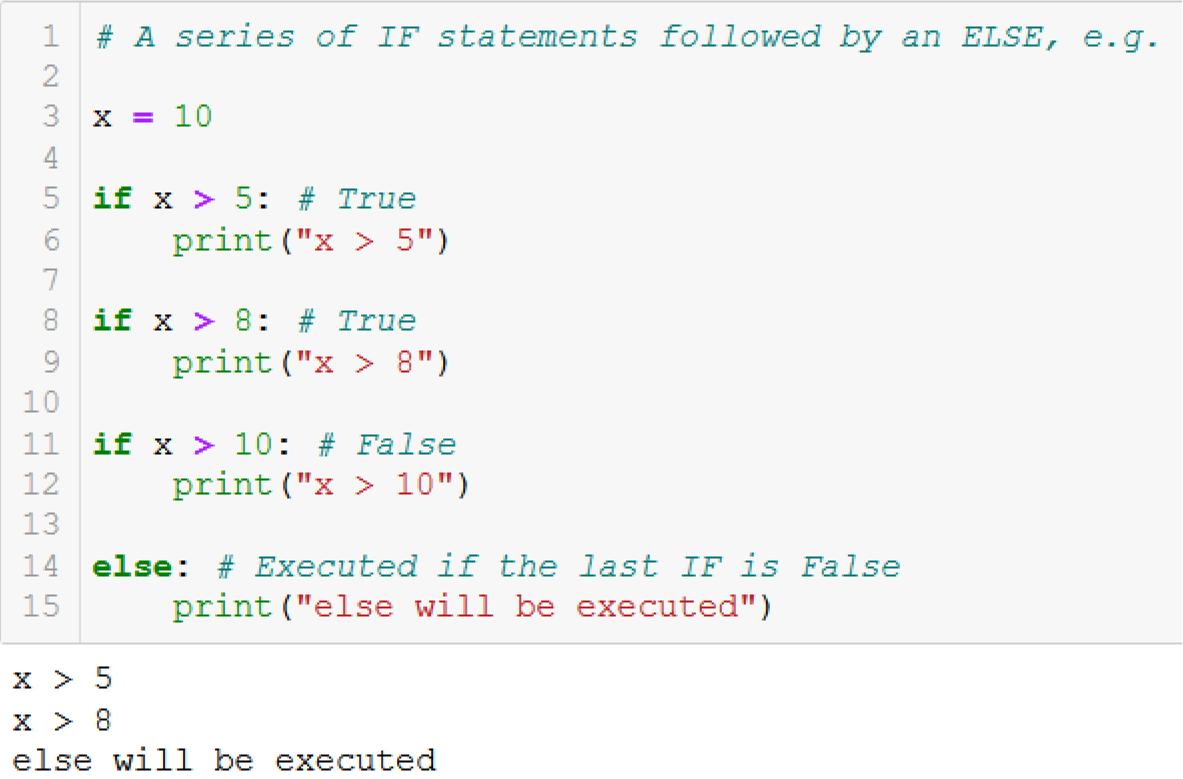
```
x = 10

if x > 5: # condition one
    print("x is greater than 5") # executed if condition one is True

if x < 10: # condition two
    print("x is less than 10") # executed if condition two is True

if x == 10: # condition three
    print("x is equal to 10") # executed if condition three is True
```

Each `if` statement is tested separately.

Conditional execution

if-elif-else instructions

Example:



```python
# A series of IF statements followed by an ELSE, e.g.

x = 10

if x > 5: # True
    print("x > 5")

if x > 8: # True
    print("x > 8")

if x > 10: # False
    print("x > 10")

else: # Executed if the last IF is False
    print("else will be executed")
```

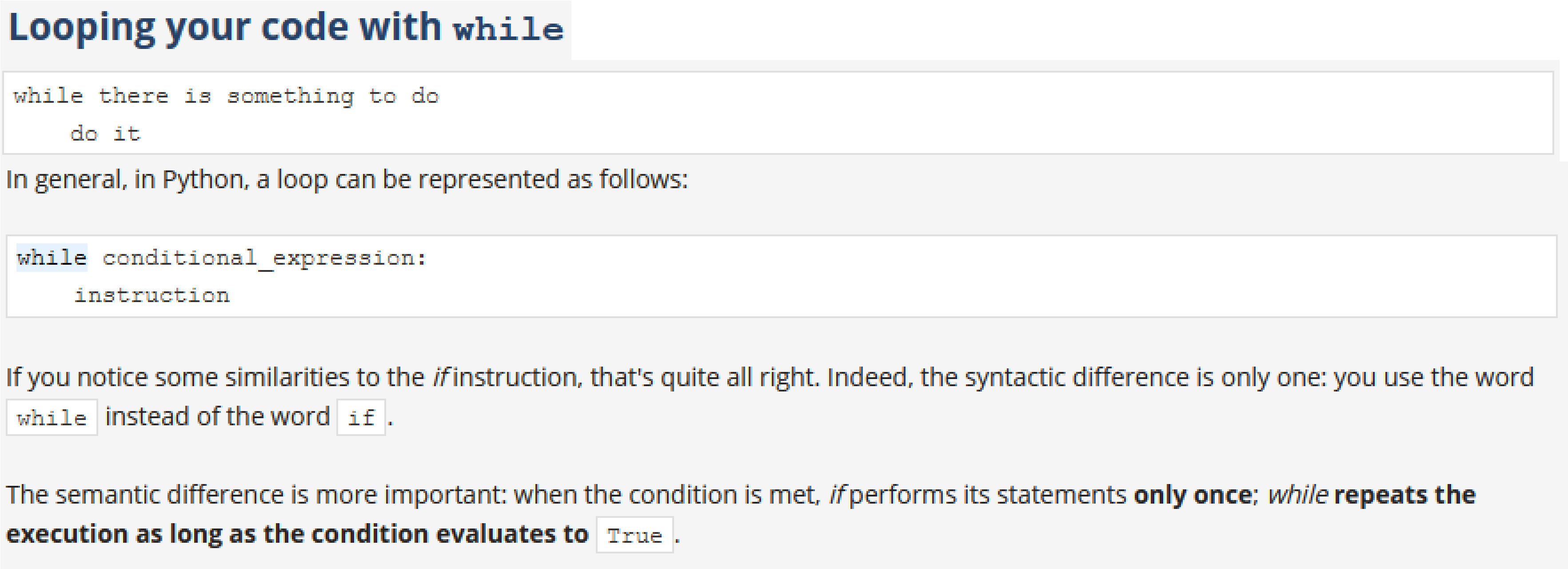
```
x > 5
x > 8
else will be executed
```

Demo
if, if-else, if-elif-else

Loops
while, for, break and continue

Loops

while and for loops



Looping your code with `while`

```
while there is something to do
    do it
```

In general, in Python, a loop can be represented as follows:

```
while conditional_expression:
    instruction
```

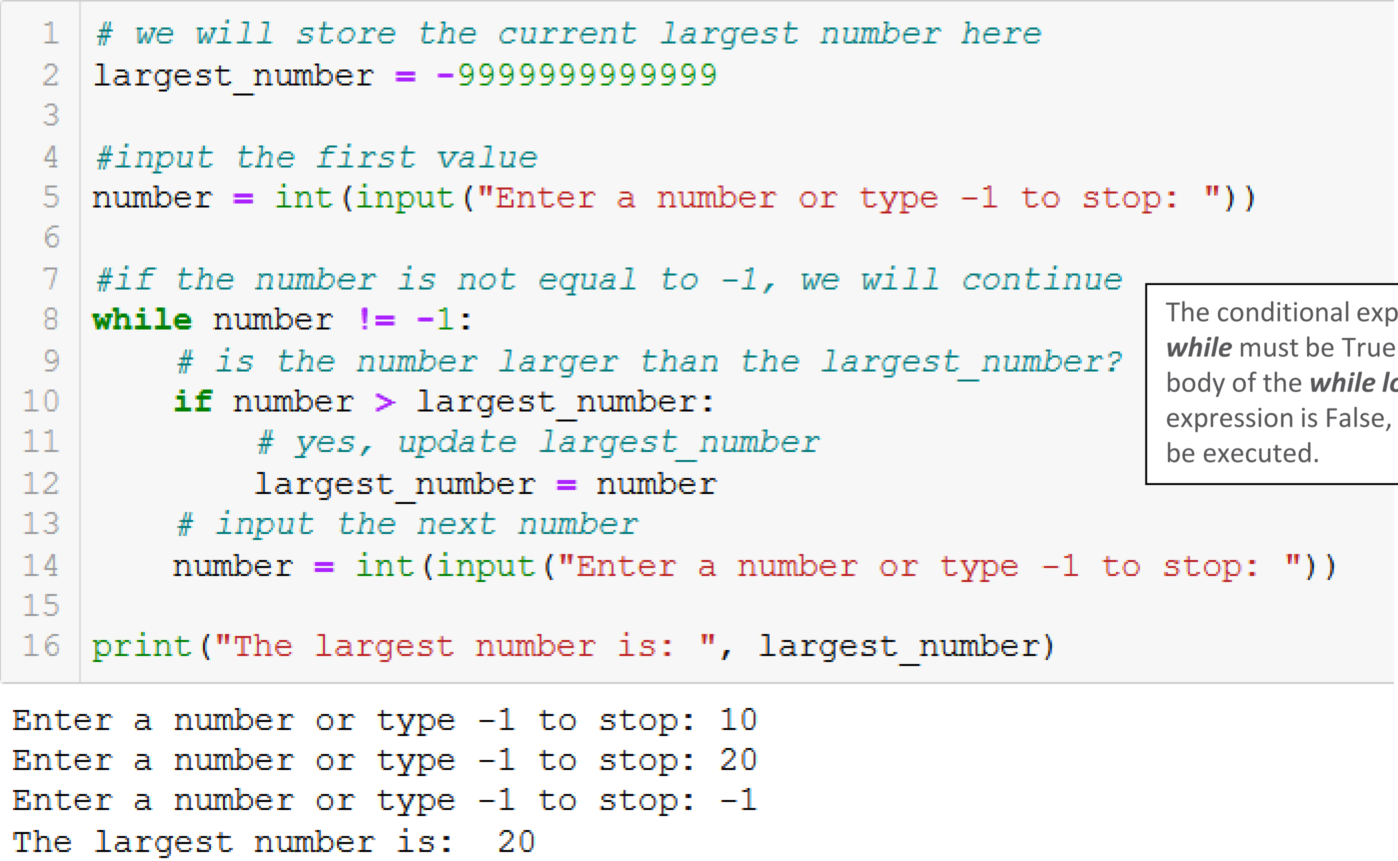
If you notice some similarities to the *if* instruction, that's quite all right. Indeed, the syntactic difference is only one: you use the word `while` instead of the word `if`.

The semantic difference is more important: when the condition is met, *if* performs its statements **only once**; *while* **repeats the execution as long as the condition evaluates to** `True`.

Loops

while and for loops

Looping your code with `while`



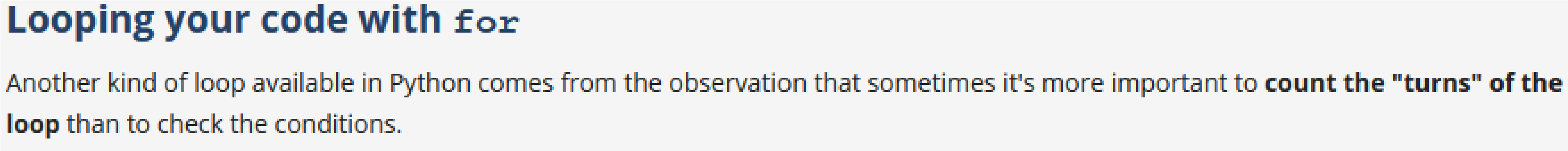
```python
# we will store the current largest number here
largest_number = -999999999999

#input the first value
number = int(input("Enter a number or type -1 to stop: "))

#if the number is not equal to -1, we will continue
while number != -1:
    # is the number larger than the largest_number?
    if number > largest_number:
        # yes, update largest_number
        largest_number = number
    # input the next number
    number = int(input("Enter a number or type -1 to stop: "))

print("The largest number is: ", largest_number)
```

```
Enter a number or type -1 to stop: 10
Enter a number or type -1 to stop: 20
Enter a number or type -1 to stop: -1
The largest number is:  20
```

The conditional expression in the ***while*** must be True to execute the body of the ***while loop***. If condition expression is False, the body will not be executed.

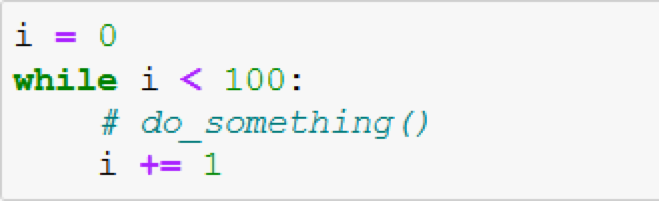
Loops

while and for loops



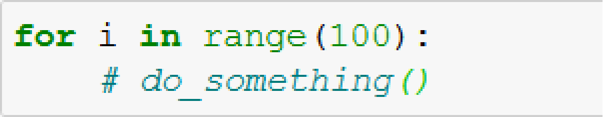
Looping your code with `for`

Another kind of loop available in Python comes from the observation that sometimes it's more important to **count the "turns" of the loop** than to check the conditions.

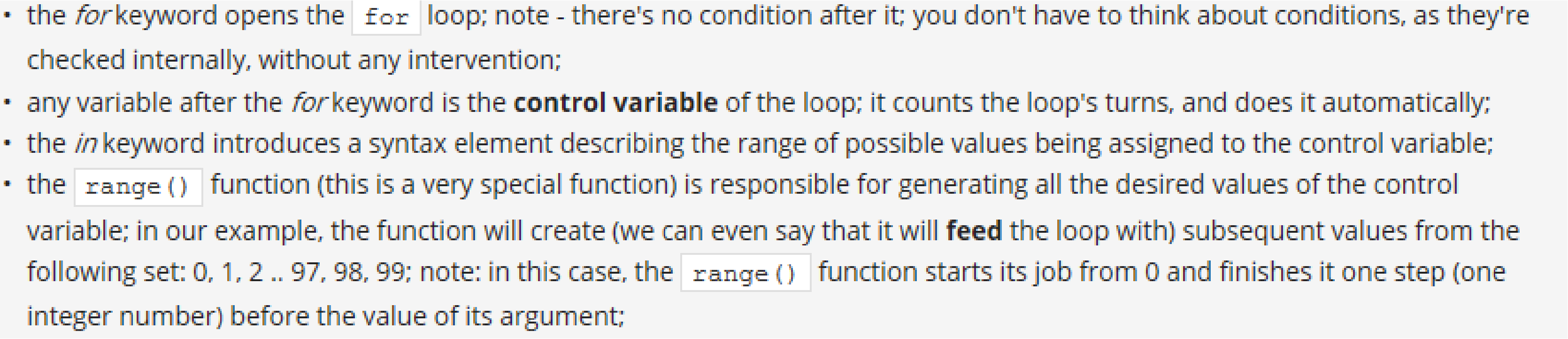


```
i = 0
while i < 100:
    # do_something()
    i += 1
```

or



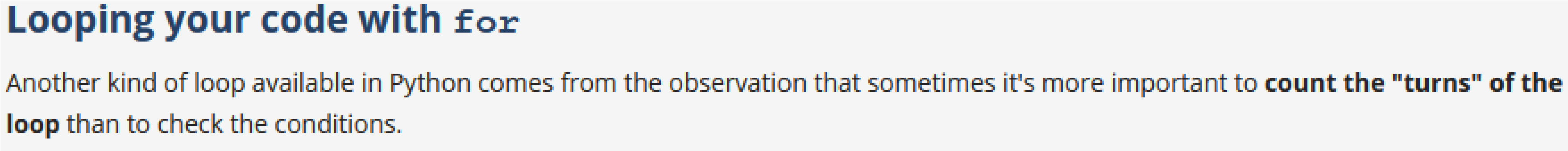
```
for i in range(100):
    # do_something()
```



- the *for* keyword opens the `for` loop; note - there's no condition after it; you don't have to think about conditions, as they're checked internally, without any intervention;
- any variable after the *for* keyword is the **control variable** of the loop; it counts the loop's turns, and does it automatically;
- the *in* keyword introduces a syntax element describing the range of possible values being assigned to the control variable;
- the `range()` function (this is a very special function) is responsible for generating all the desired values of the control variable; in our example, the function will create (we can even say that it will **feed** the loop with) subsequent values from the following set: 0, 1, 2 .. 97, 98, 99; note: in this case, the `range()` function starts its job from 0 and finishes it one step (one integer number) before the value of its argument;

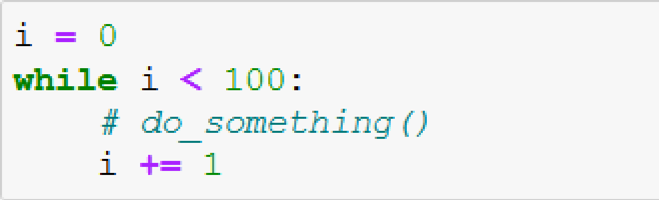
Loops

while and for loops



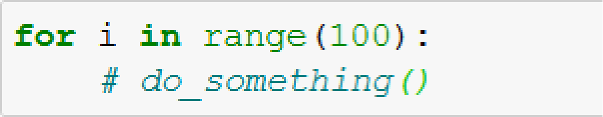
Looping your code with `for`

Another kind of loop available in Python comes from the observation that sometimes it's more important to **count the "turns" of the loop** than to check the conditions.

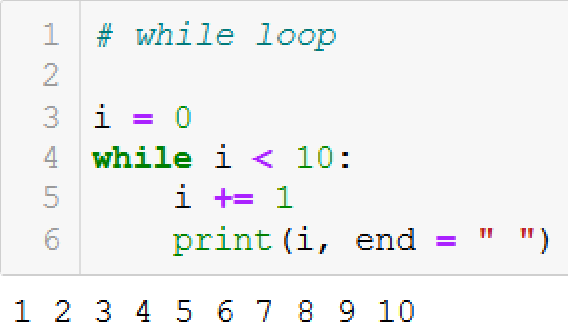


```
i = 0
while i < 100:
    # do_something()
    i += 1
```

or

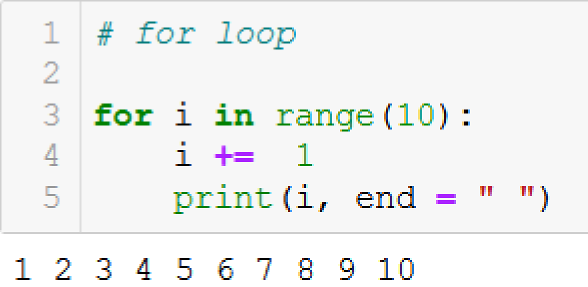


```
for i in range(100):
    # do_something()
```



```
# while loop

i = 0
while i < 10:
    i += 1
    print(i, end = " ")
```

```
1 2 3 4 5 6 7 8 9 10
```



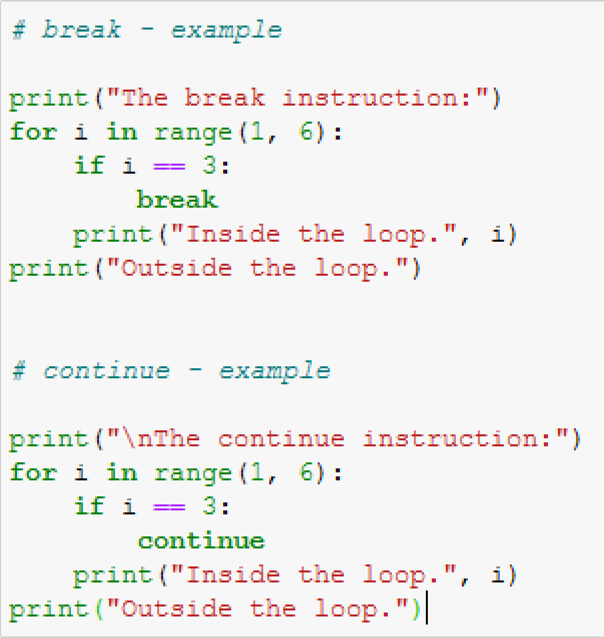
```
# for loop

for i in range(10):
    i +=  1
    print(i, end = " ")
```

```
1 2 3 4 5 6 7 8 9 10
```

Loops

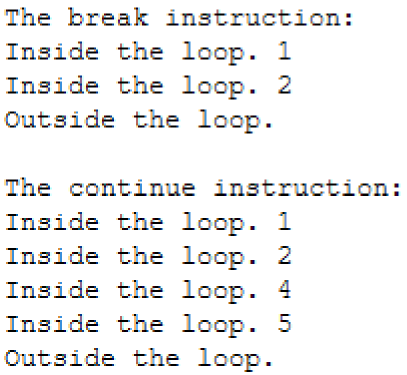
while and for loops

The break and continue statements



```
# break - example

print("The break instruction:")
for i in range(1, 6):
    if i == 3:
        break
    print("Inside the loop.", i)
print("Outside the loop.")


# continue - example

print("\nThe continue instruction:")
for i in range(1, 6):
    if i == 3:
        continue
    print("Inside the loop.", i)
print("Outside the loop.")
```

Output

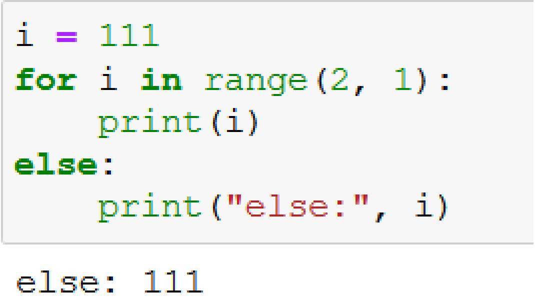


```
The break instruction:
Inside the loop. 1
Inside the loop. 2
Outside the loop.

The continue instruction:
Inside the loop. 1
Inside the loop. 2
Inside the loop. 4
Inside the loop. 5
Outside the loop.
```

Loops
while and for loops

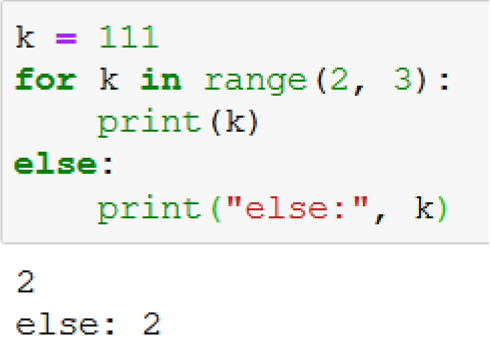
The `for` loop and the `else` branch

Examples:



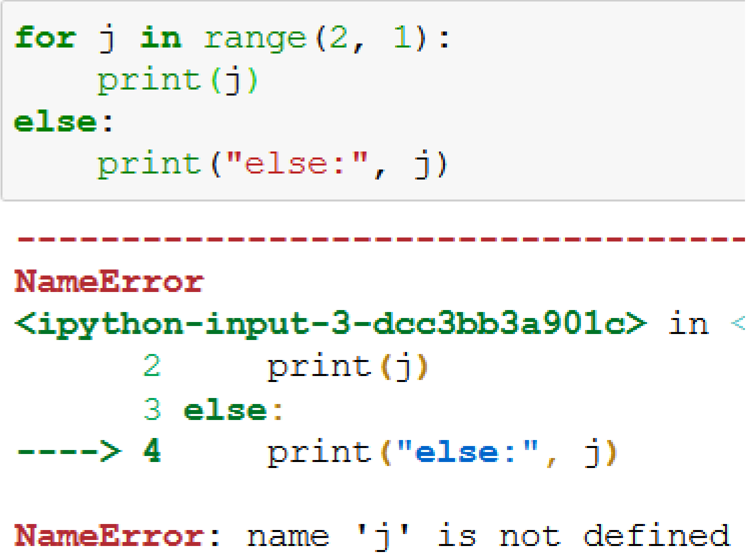
```
i = 111
for i in range(2, 1):
    print(i)
else:
    print("else:", i)
```

```
else: 111
```



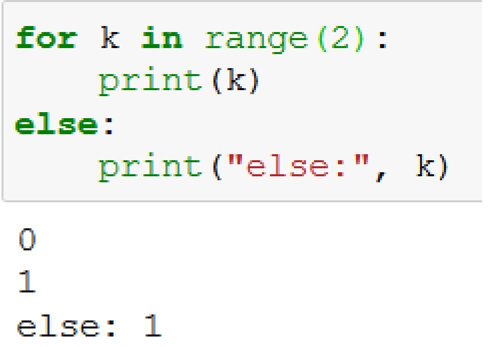
```
k = 111
for k in range(2, 3):
    print(k)
else:
    print("else:", k)
```

```
2
else: 2
```



```
for j in range(2, 1):
    print(j)
else:
    print("else:", j)
```

```
---------------------------------------
NameError
<ipython-input-3-dcc3bb3a901c> in <
      2     print(j)
      3 else:
----> 4     print("else:", j)

NameError: name 'j' is not defined
```



```
for k in range(2):
    print(k)
else:
    print("else:", k)
```

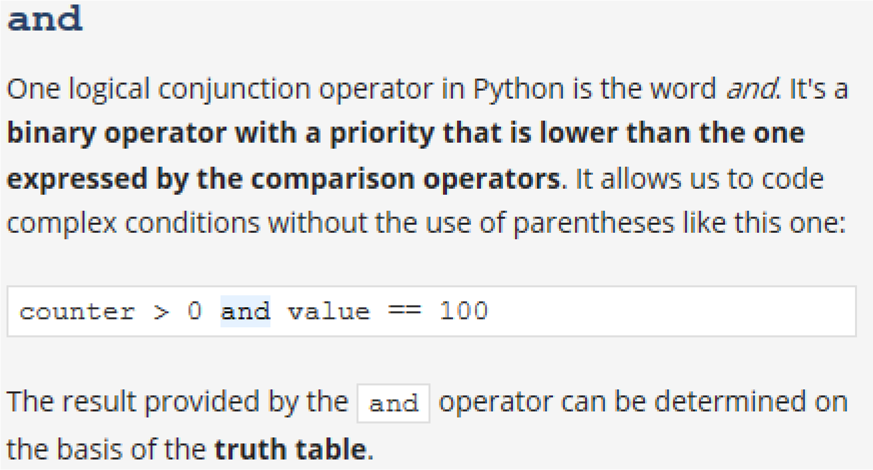
```
0
1
else: 1
```

Demo
while, for, break and continue

Logic and bit operations and, or, not

Logic and Bit operations

Computer Logic operators

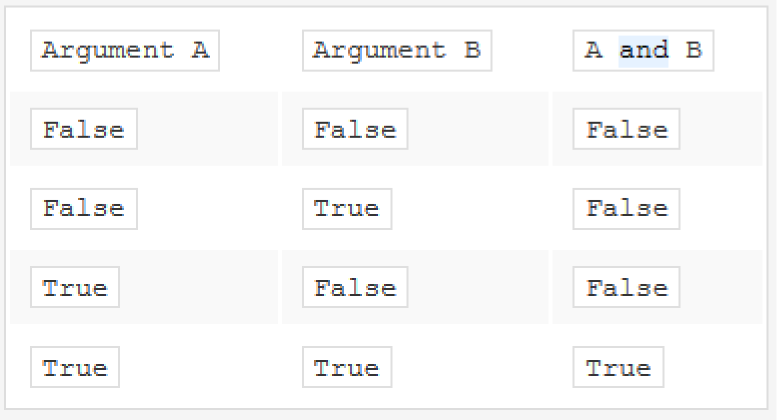


and

One logical conjunction operator in Python is the word *and*. It's a **binary operator with a priority that is lower than the one expressed by the comparison operators**. It allows us to code complex conditions without the use of parentheses like this one:

```
counter > 0 and value == 100
```

The result provided by the `and` operator can be determined on the basis of the **truth table**.

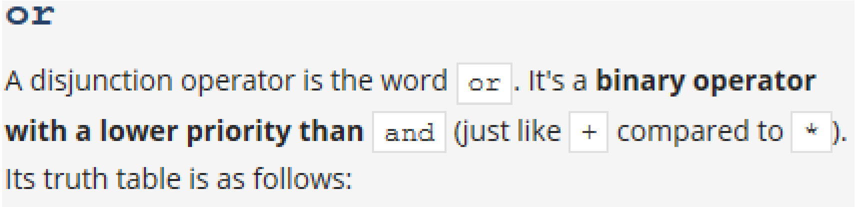


Argument A	Argument B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

“**and**” operator is called conjunction

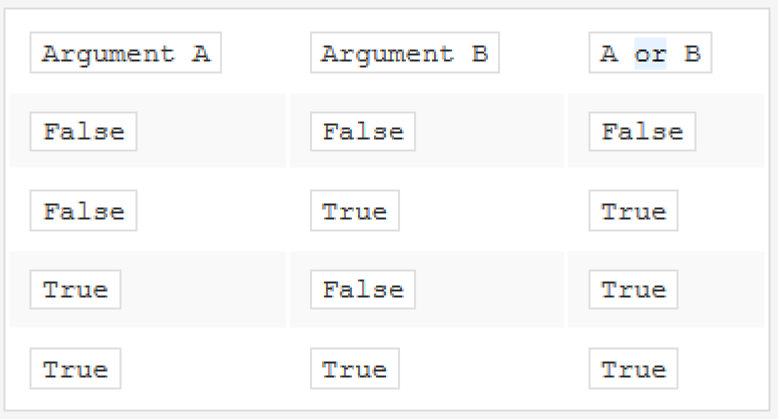
Logic and Bit operations

Computer Logic operators



or

A disjunction operator is the word `or`. It's a **binary operator with a lower priority than** `and` (just like `+` compared to `*`). Its truth table is as follows:

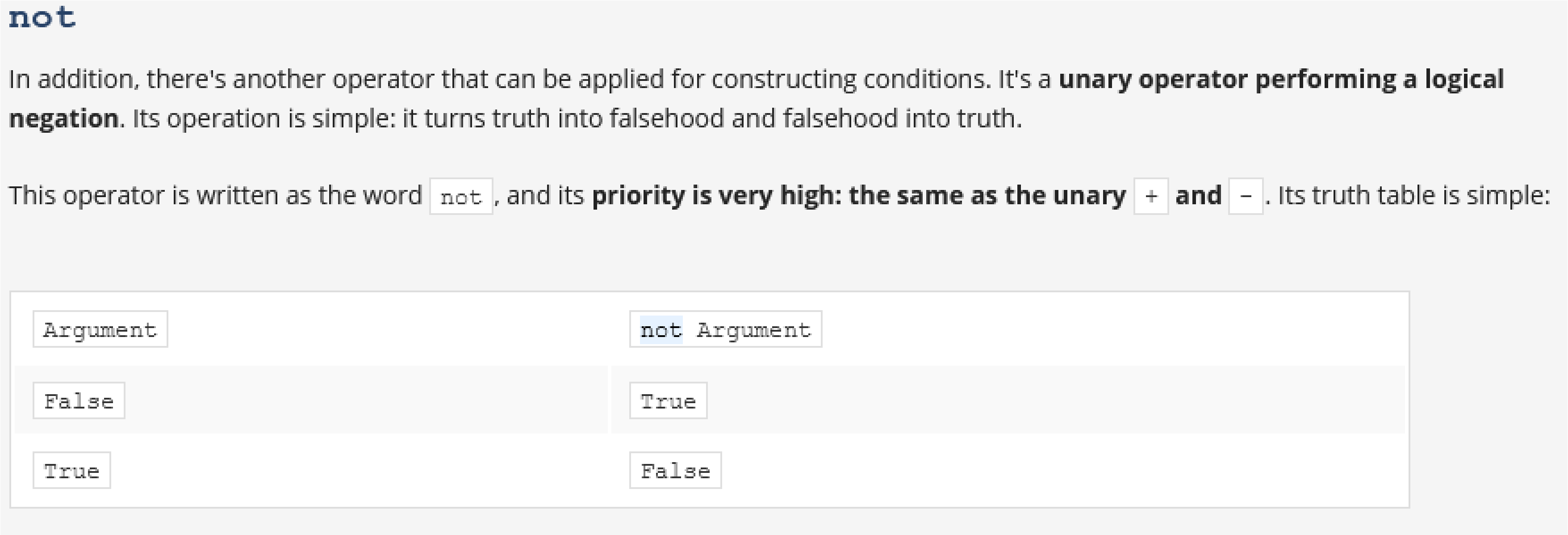


`Argument A`	`Argument B`	`A or B`
`False`	`False`	`False`
`False`	`True`	`True`
`True`	`False`	`True`
`True`	`True`	`True`

“**or**” operator is called disjunction

Logic and Bit operations

Computer Logic operators



not

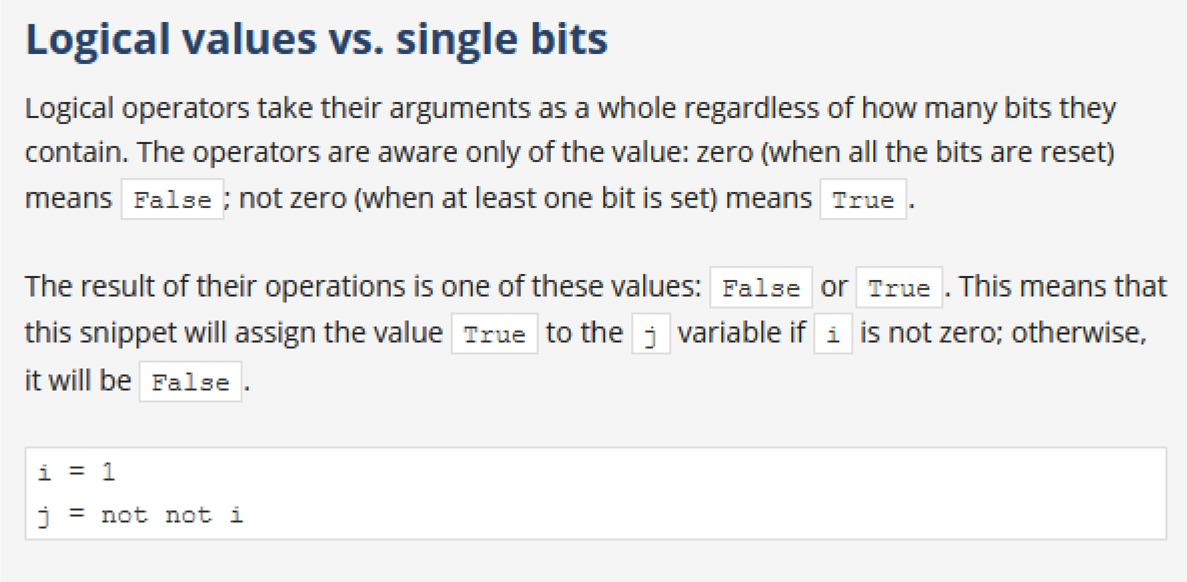
In addition, there's another operator that can be applied for constructing conditions. It's a **unary operator performing a logical negation**. Its operation is simple: it turns truth into falsehood and falsehood into truth.

This operator is written as the word `not`, and its **priority is very high: the same as the unary** `+` **and** `-`. Its truth table is simple:

`Argument`	`not Argument`
`False`	`True`
`True`	`False`

“**not**” operator is called negation

Logic and Bit operations
Computer Logic operators



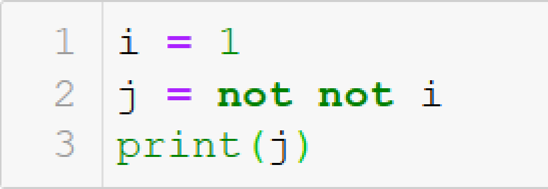
Logical values vs. single bits

Logical operators take their arguments as a whole regardless of how many bits they contain. The operators are aware only of the value: zero (when all the bits are reset) means `False`; not zero (when at least one bit is set) means `True`.

The result of their operations is one of these values: `False` or `True`. This means that this snippet will assign the value `True` to the `j` variable if `i` is not zero; otherwise, it will be `False`.

```
i = 1
j = not not i
```

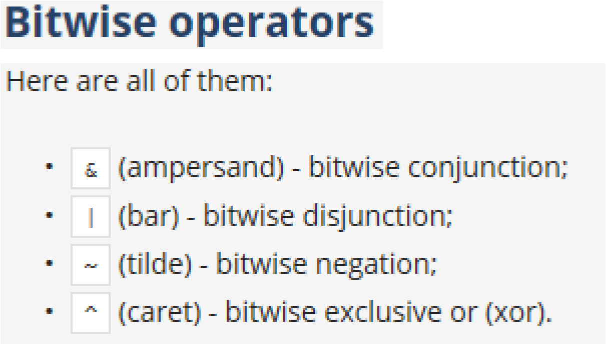
Example:



```
i = 1
j = not not i
print(j)
```

```
True
```

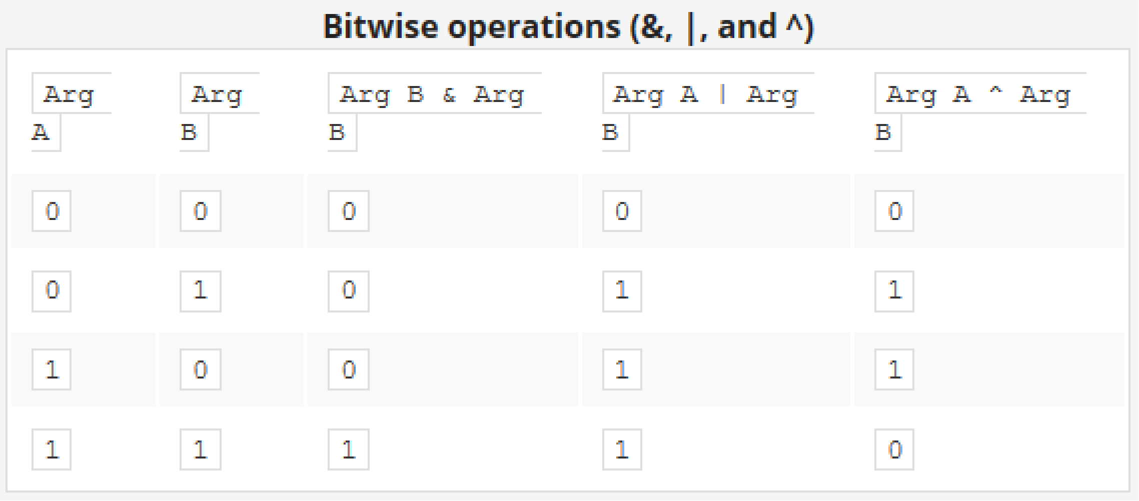
Bitwise operators



Bitwise operators

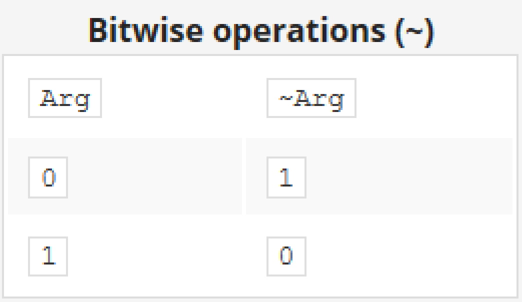
Here are all of them:

- `&` (ampersand) - bitwise conjunction;
- `|` (bar) - bitwise disjunction;
- `~` (tilde) - bitwise negation;
- `^` (caret) - bitwise exclusive or (xor).



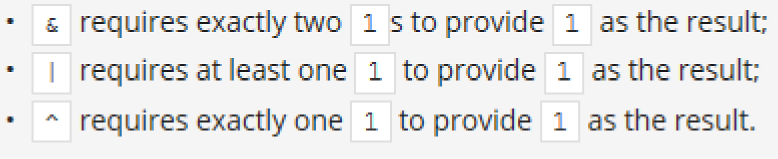
Bitwise operations (&, |, and ^)

`Arg A`	`Arg B`	`Arg B & Arg B`	`Arg A | Arg B`	`Arg A ^ Arg B`
`0`	`0`	`0`	`0`	`0`
`0`	`1`	`0`	`1`	`1`
`1`	`0`	`0`	`1`	`1`
`1`	`1`	`1`	`1`	`0`



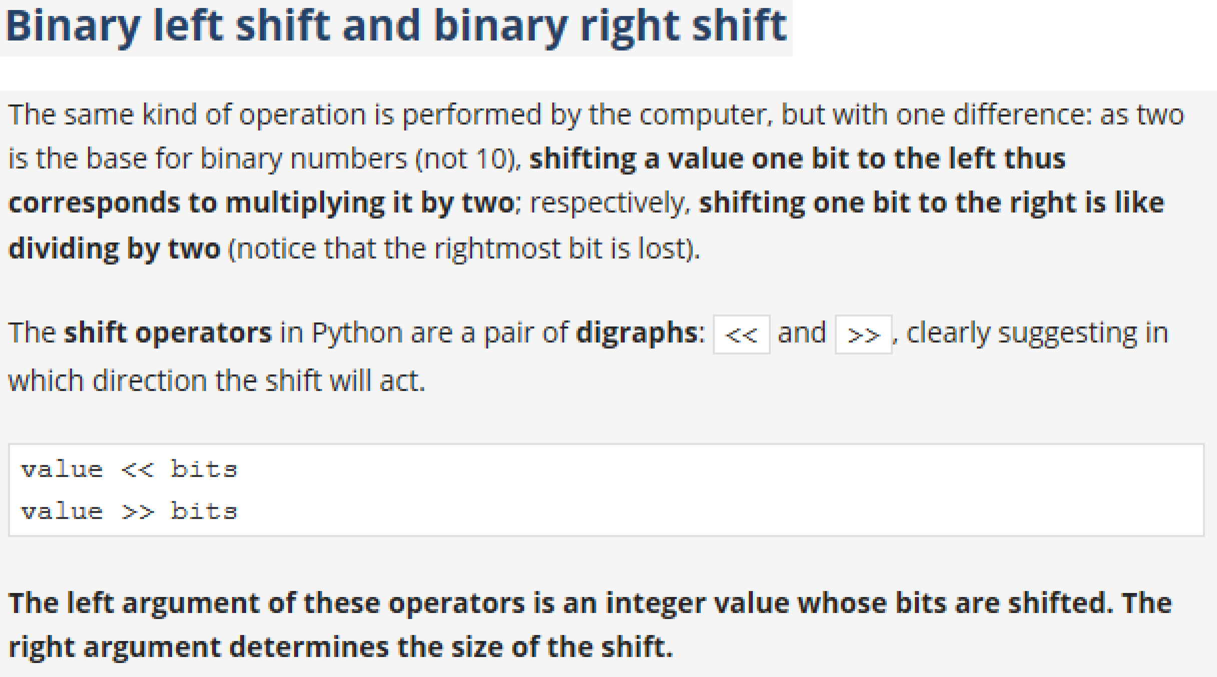
Bitwise operations (~)

`Arg`	`~Arg`
`0`	`1`
`1`	`0`



- `&` requires exactly two `1`s to provide `1` as the result;
- `|` requires at least one `1` to provide `1` as the result;
- `^` requires exactly one `1` to provide `1` as the result.

Logic and Bit operations
Bit shifting



Binary left shift and binary right shift

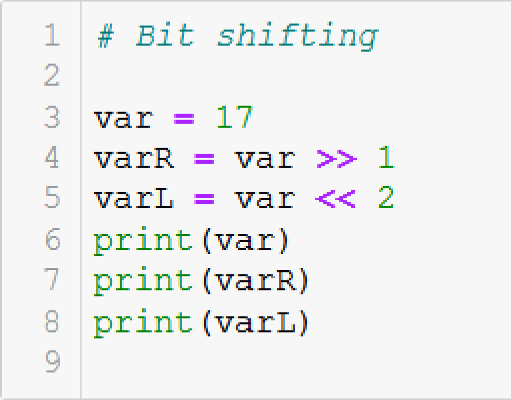
The same kind of operation is performed by the computer, but with one difference: as two is the base for binary numbers (not 10), **shifting a value one bit to the left thus corresponds to multiplying it by two**; respectively, **shifting one bit to the right is like dividing by two** (notice that the rightmost bit is lost).

The **shift operators** in Python are a pair of **digraphs**: `<<` and `>>`, clearly suggesting in which direction the shift will act.

```
value << bits
value >> bits
```

The left argument of these operators is an integer value whose bits are shifted. The right argument determines the size of the shift.

Example:



```python
# Bit shifting

var = 17
varR = var >> 1
varL = var << 2
print(var)
print(varR)
print(varL)

```

```
17
8
68
```

Logic and bit operations
and, or, not

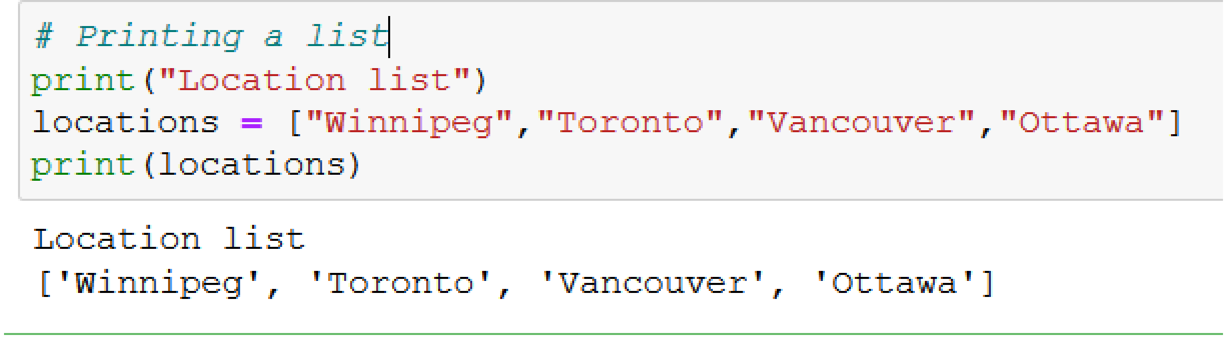
Lists

Lists – Collections of Data
Indexing

List is a collection of elements, but each element is a **scalar.**
Scalar – declared variables that are able to store exactly one given value at a time.

The value inside the brackets which selects one element of the list is called an **index**, while the operation of selecting an element from the list is known as **indexing**.

Example:

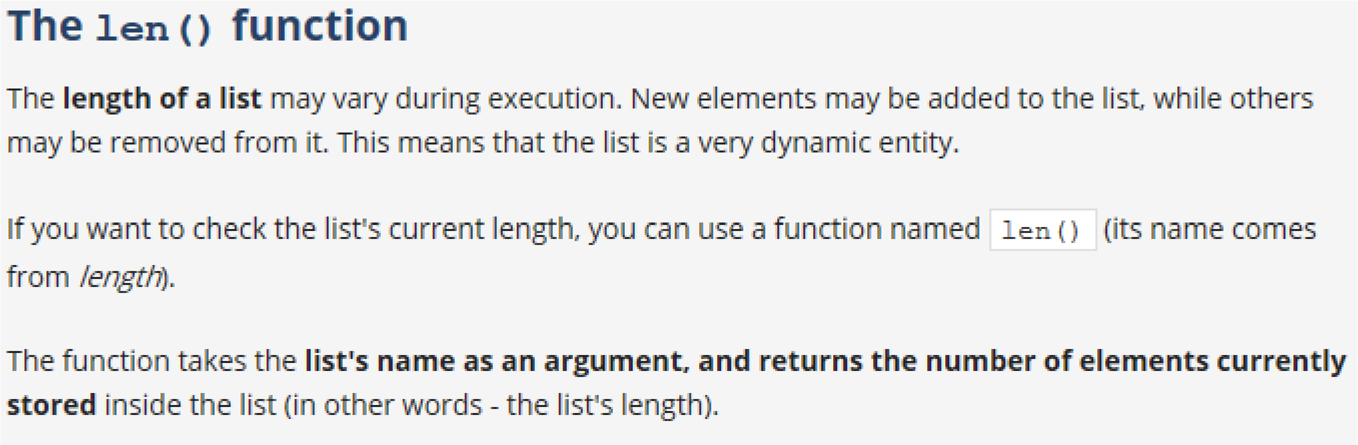


```
# Printing a list
print("Location list")
locations = ["Winnipeg","Toronto","Vancouver","Ottawa"]
print(locations)
```

```
Location list
['Winnipeg', 'Toronto', 'Vancouver', 'Ottawa']
```

Lists – Collections of Data

Indexing

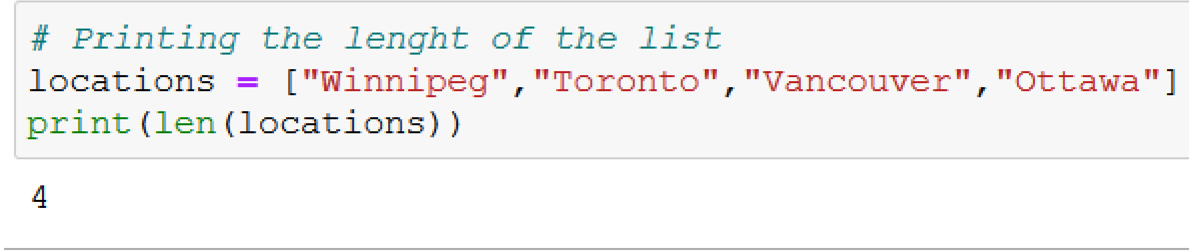


The `len()` function

The **length of a list** may vary during execution. New elements may be added to the list, while others may be removed from it. This means that the list is a very dynamic entity.

If you want to check the list's current length, you can use a function named `len()` (its name comes from *length*).

The function takes the **list's name as an argument, and returns the number of elements currently stored** inside the list (in other words - the list's length).



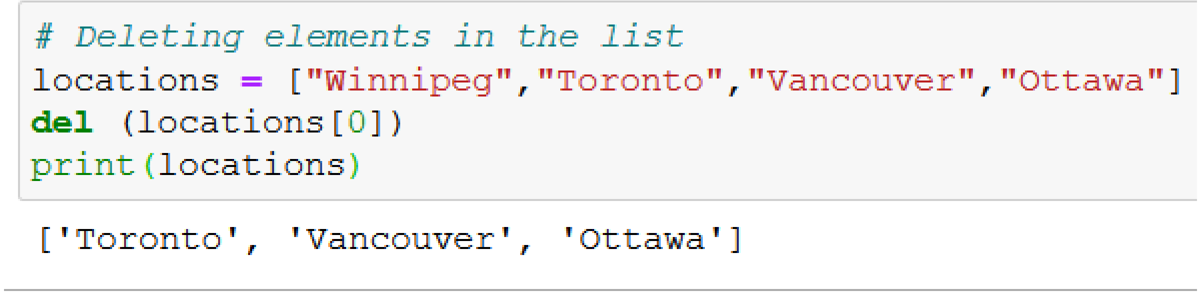
```
# Printing the lenght of the list
locations = ["Winnipeg","Toronto","Vancouver","Ottawa"]
print(len(locations))
```

```
4
```

Lists – Collections of Data

Operations on list

Examples:



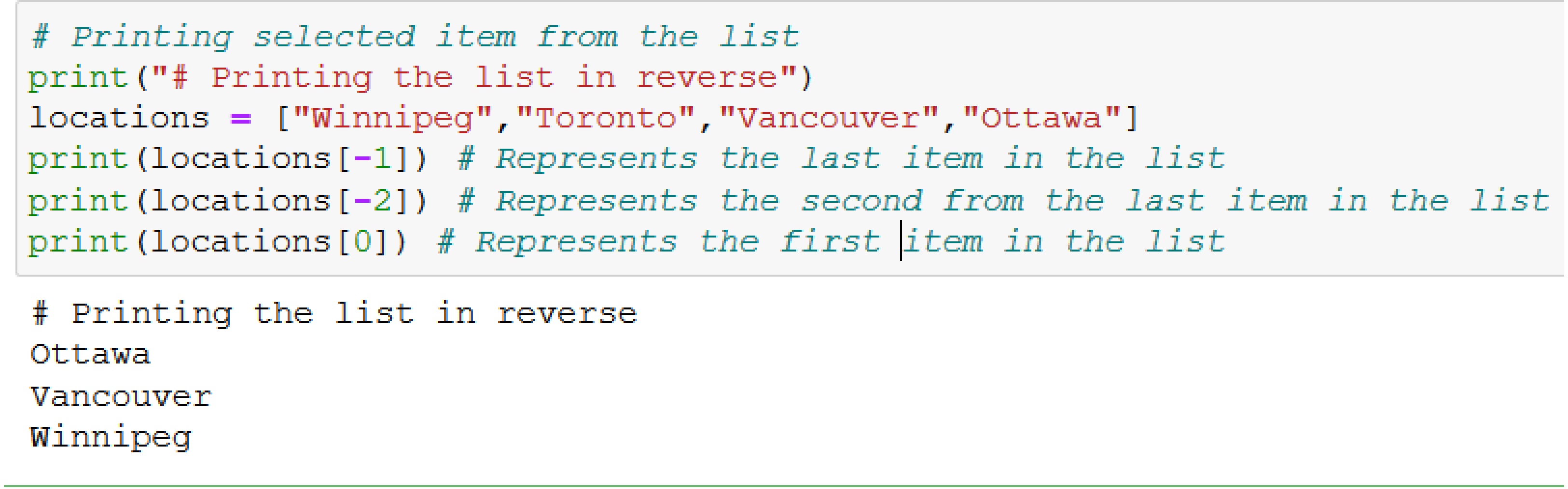
```
# Deleting elements in the list
locations = ["Winnipeg","Toronto","Vancouver","Ottawa"]
del (locations[0])
print(locations)
```

```
['Toronto', 'Vancouver', 'Ottawa']
```

Lists – Collections of Data

Operations on lists

More examples:



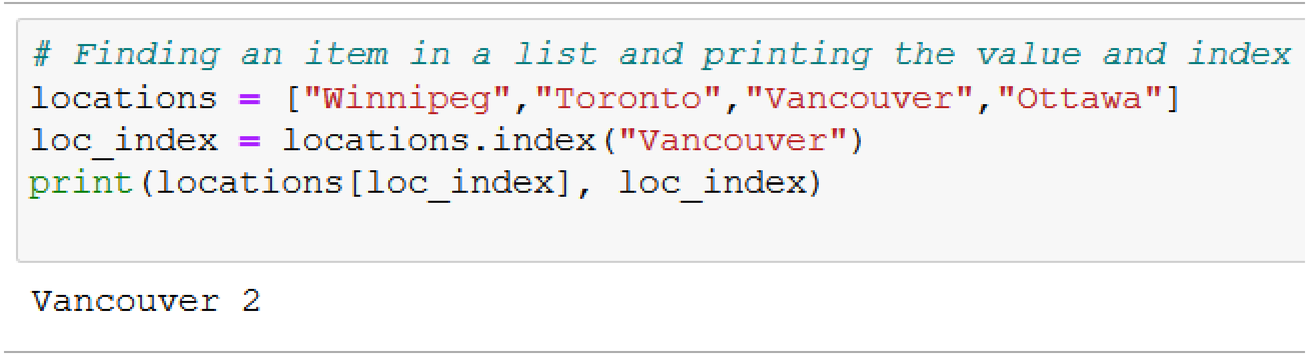
```
# Printing selected item from the list
print("# Printing the list in reverse")
locations = ["Winnipeg","Toronto","Vancouver","Ottawa"]
print(locations[-1]) # Represents the last item in the list
print(locations[-2]) # Represents the second from the last item in the list
print(locations[0]) # Represents the first item in the list
```

```
# Printing the list in reverse
Ottawa
Vancouver
Winnipeg
```

Lists – Collections of Data

Operations on list

More examples:

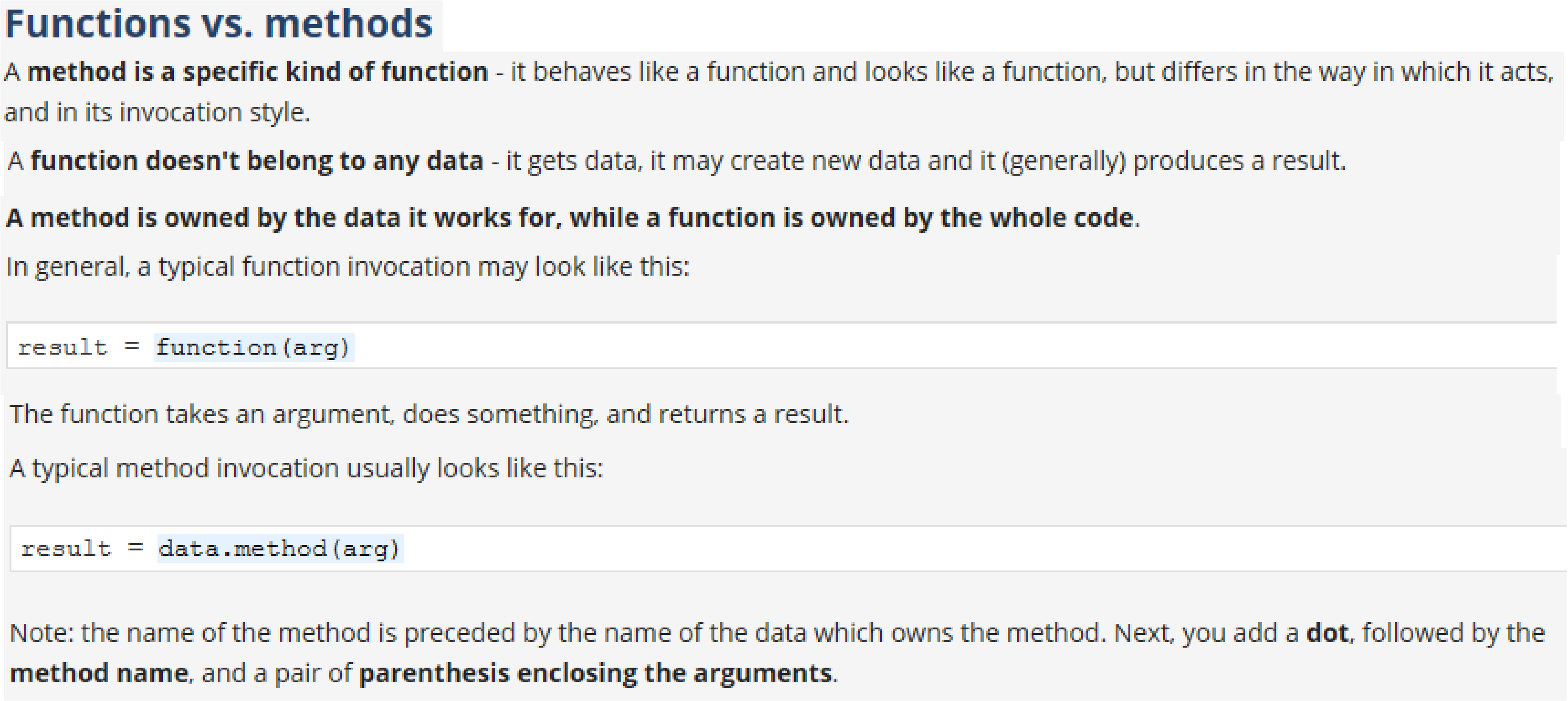


```
# Finding an item in a list and printing the value and index
locations = ["Winnipeg","Toronto","Vancouver","Ottawa"]
loc_index = locations.index("Vancouver")
print(locations[loc_index], loc_index)
```

```
Vancouver 2
```

Lists – Collections of Data

Functions and methods



Functions vs. methods

A **method is a specific kind of function** - it behaves like a function and looks like a function, but differs in the way in which it acts, and in its invocation style.

A **function doesn't belong to any data** - it gets data, it may create new data and it (generally) produces a result.

A method is owned by the data it works for, while a function is owned by the whole code.

In general, a typical function invocation may look like this:

```
result = function(arg)
```

The function takes an argument, does something, and returns a result.

A typical method invocation usually looks like this:

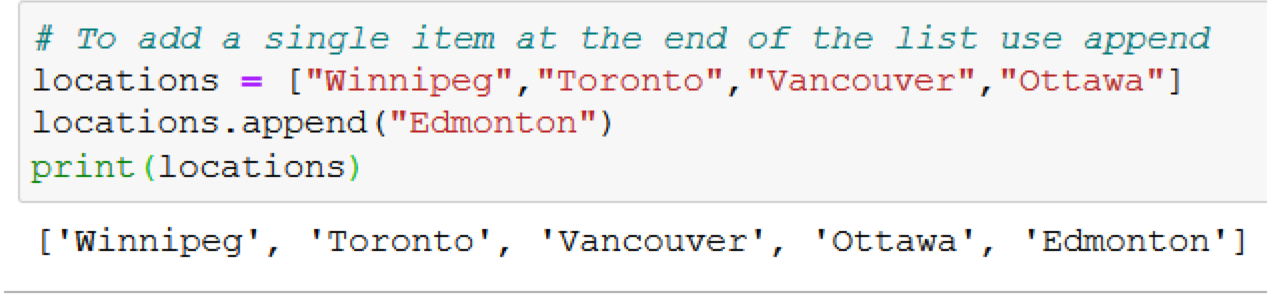
```
result = data.method(arg)
```

Note: the name of the method is preceded by the name of the data which owns the method. Next, you add a **dot**, followed by the **method name**, and a pair of **parenthesis enclosing the arguments**.

Lists – Collections of Data

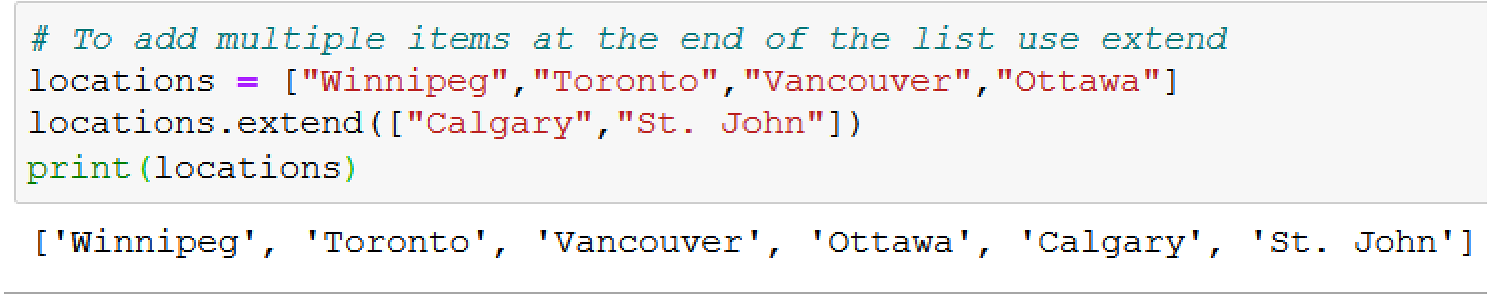
List methods

Examples:



```
# To add a single item at the end of the list use append
locations = ["Winnipeg","Toronto","Vancouver","Ottawa"]
locations.append("Edmonton")
print(locations)
```

```
['Winnipeg', 'Toronto', 'Vancouver', 'Ottawa', 'Edmonton']
```



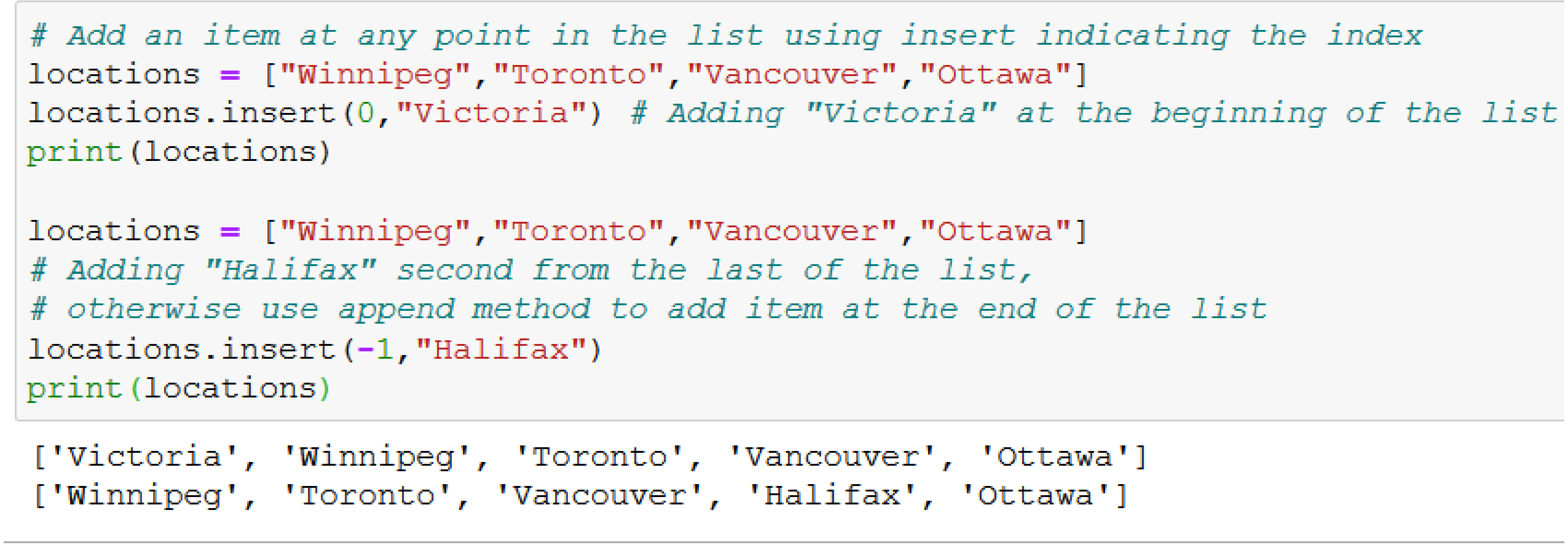
```
# To add multiple items at the end of the list use extend
locations = ["Winnipeg","Toronto","Vancouver","Ottawa"]
locations.extend(["Calgary","St. John"])
print(locations)
```

```
['Winnipeg', 'Toronto', 'Vancouver', 'Ottawa', 'Calgary', 'St. John']
```

Lists – Collections of Data

List methods

More examples:



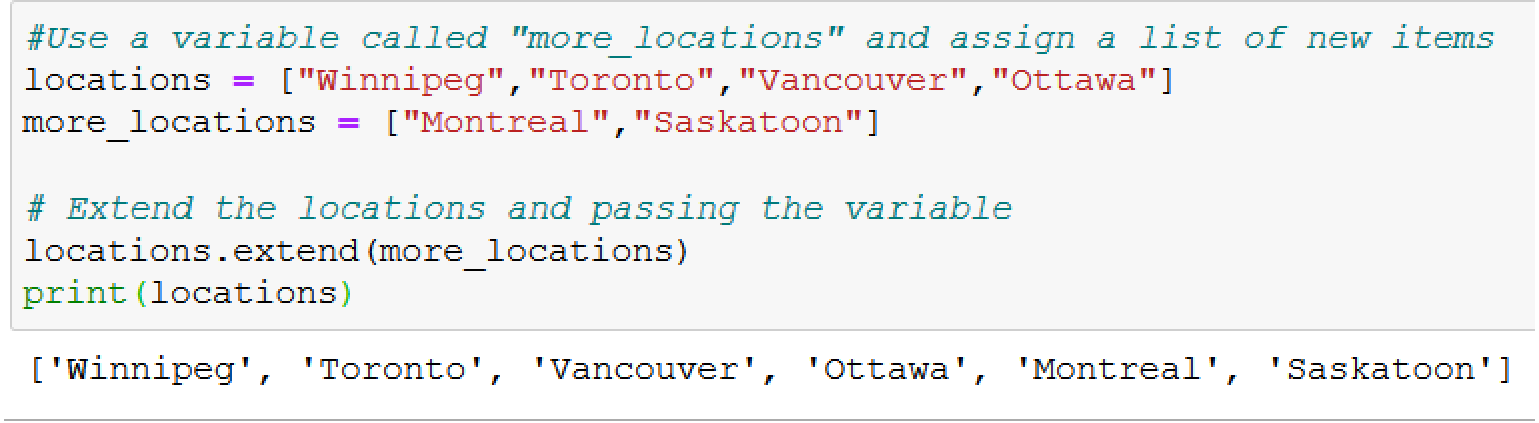
```
# Add an item at any point in the list using insert indicating the index
locations = ["Winnipeg","Toronto","Vancouver","Ottawa"]
locations.insert(0,"Victoria") # Adding "Victoria" at the beginning of the list
print(locations)

locations = ["Winnipeg","Toronto","Vancouver","Ottawa"]
# Adding "Halifax" second from the last of the list,
# otherwise use append method to add item at the end of the list
locations.insert(-1,"Halifax")
print(locations)
```

```
['Victoria', 'Winnipeg', 'Toronto', 'Vancouver', 'Ottawa']
['Winnipeg', 'Toronto', 'Vancouver', 'Halifax', 'Ottawa']
```

Lists – Collections of Data

List methods

More examples:



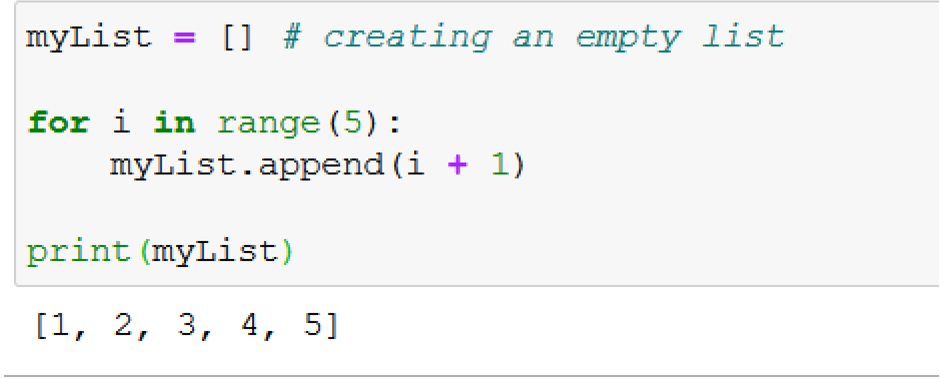
```
#Use a variable called "more_locations" and assign a list of new items
locations = ["Winnipeg","Toronto","Vancouver","Ottawa"]
more_locations = ["Montreal","Saskatoon"]

# Extend the locations and passing the variable
locations.extend(more_locations)
print(locations)
```

```
['Winnipeg', 'Toronto', 'Vancouver', 'Ottawa', 'Montreal', 'Saskatoon']
```

Lists – Collections of Data

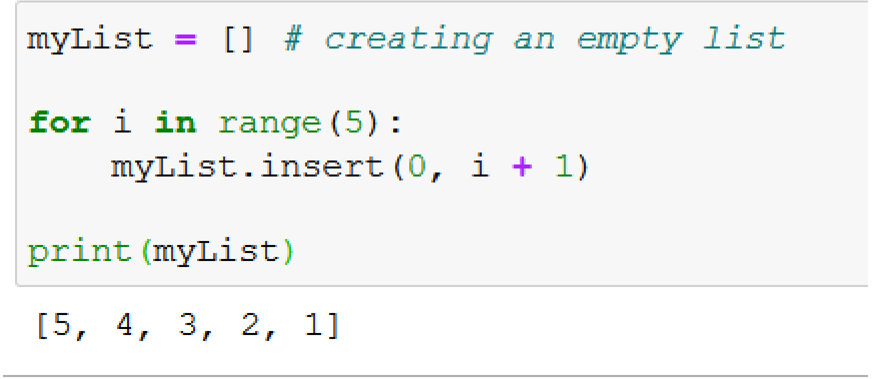
List methods

More examples:



```
myList = [] # creating an empty list

for i in range(5):
    myList.append(i + 1)

print(myList)
```

```
[1, 2, 3, 4, 5]
```



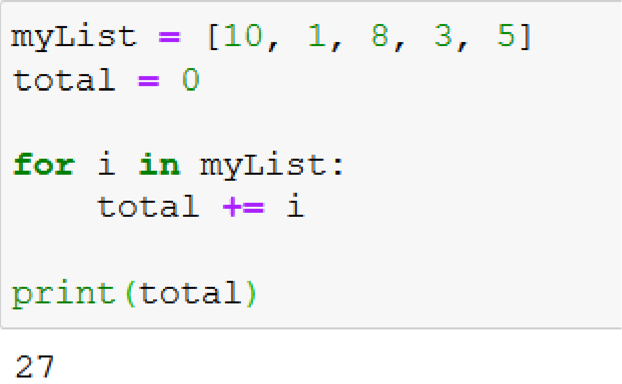
```
myList = [] # creating an empty list

for i in range(5):
    myList.insert(0, i + 1)

print(myList)
```

```
[5, 4, 3, 2, 1]
```

Lists – Collections of Data

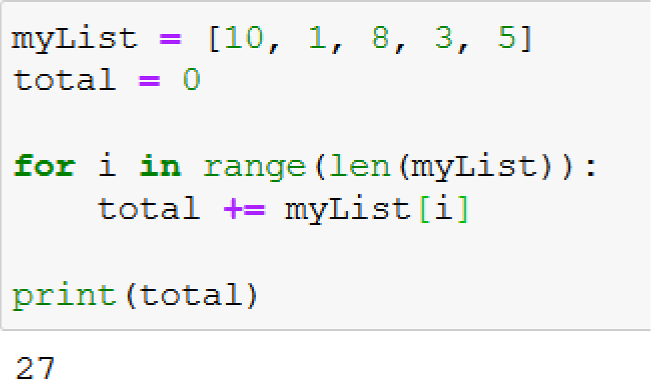
Lists and loops | making use of lists

Examples:



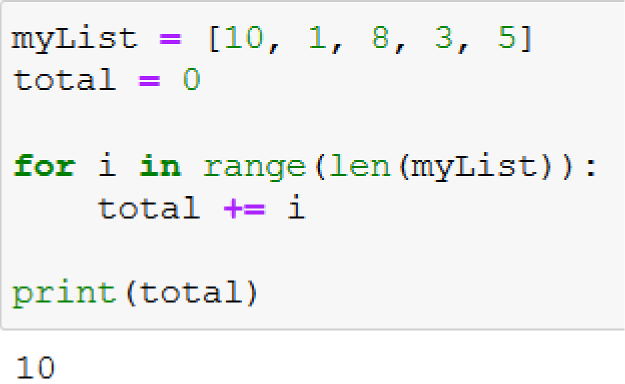
```
myList = [10, 1, 8, 3, 5]
total = 0

for i in myList:
    total += i

print(total)
```

```
27
```



```
myList = [10, 1, 8, 3, 5]
total = 0

for i in range(len(myList)):
    total += myList[i]

print(total)
```

```
27
```



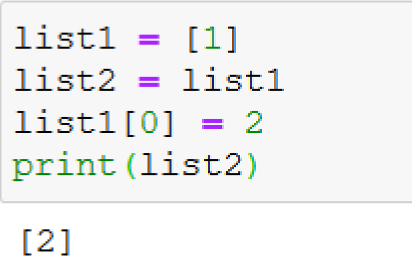
```
myList = [10, 1, 8, 3, 5]
total = 0

for i in range(len(myList)):
    total += i

print(total)
```

```
10
```

Lists – Operations of lists

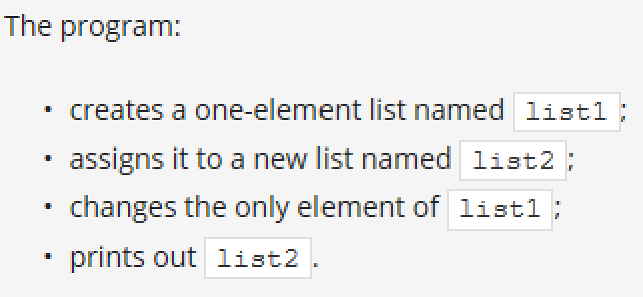
The inner life of lists

Example:



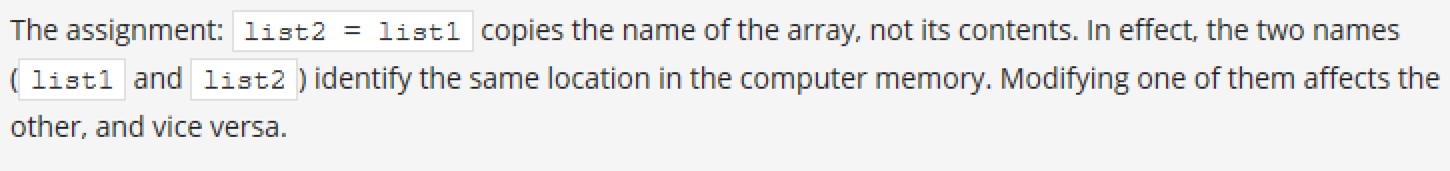
```
list1 = [1]
list2 = list1
list1[0] = 2
print(list2)
```

```
[2]
```



The program:

- creates a one-element list named `list1`;
- assigns it to a new list named `list2`;
- changes the only element of `list1`;
- prints out `list2`.

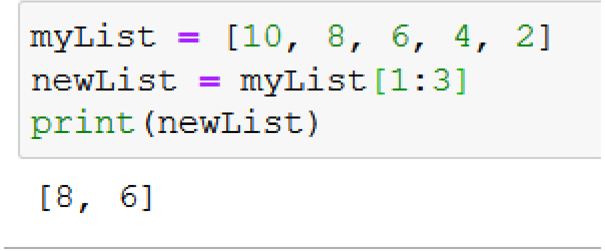


The assignment: `list2 = list1` copies the name of the array, not its contents. In effect, the two names (`list1` and `list2`) identify the same location in the computer memory. Modifying one of them affects the other, and vice versa.

Lists – Operations of lists

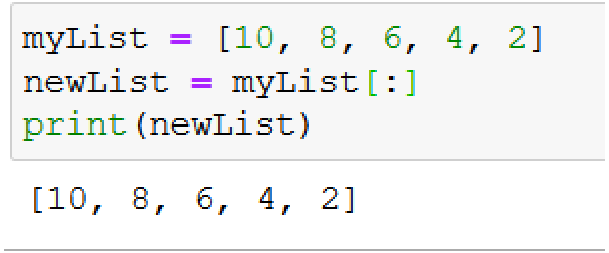
Slices

More examples:



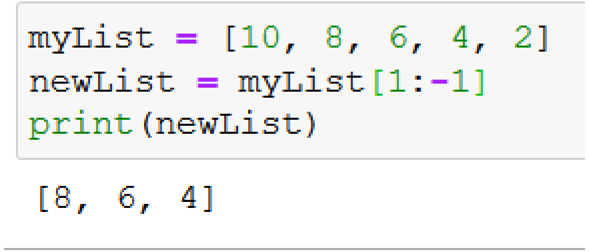
```
myList = [10, 8, 6, 4, 2]
newList = myList[1:3]
print(newList)
```

```
[8, 6]
```



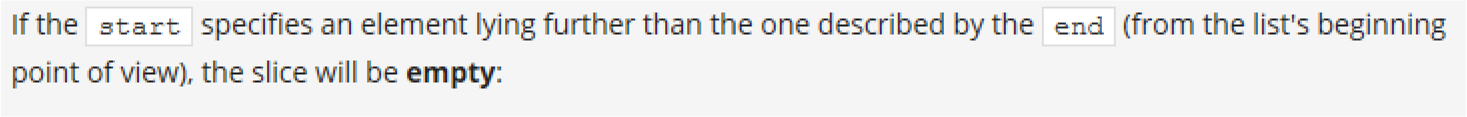
```
myList = [10, 8, 6, 4, 2]
newList = myList[:]
print(newList)
```

```
[10, 8, 6, 4, 2]
```

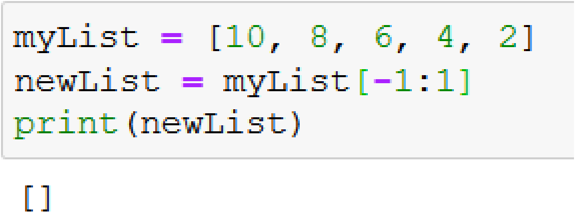


```
myList = [10, 8, 6, 4, 2]
newList = myList[1:-1]
print(newList)
```

```
[8, 6, 4]
```



If the `start` specifies an element lying further than the one described by the `end` (from the list's beginning point of view), the slice will be **empty**:



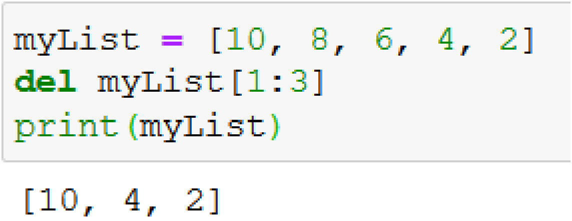
```
myList = [10, 8, 6, 4, 2]
newList = myList[-1:1]
print(newList)
```

```
[]
```

Lists – Operations of lists

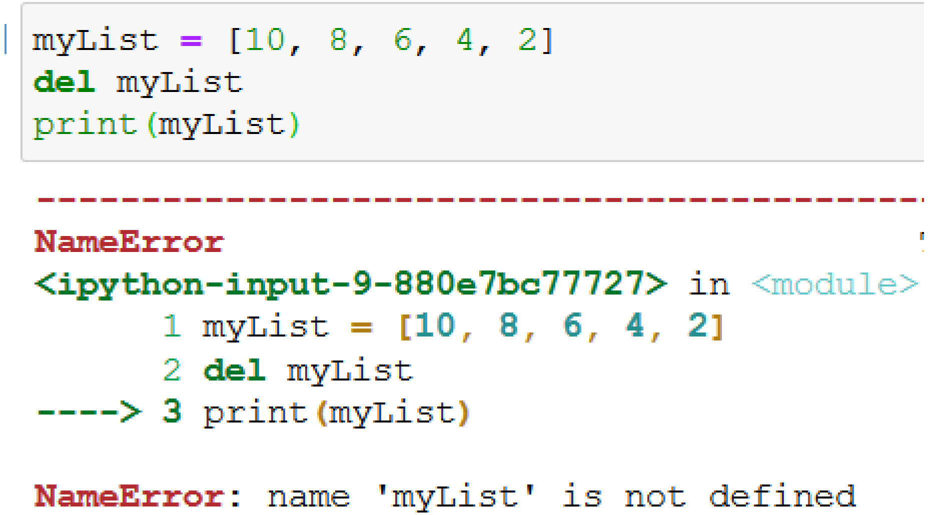
Slices

More examples:



```
myList = [10, 8, 6, 4, 2]
del myList[1:3]
print(myList)
```

```
[10, 4, 2]
```



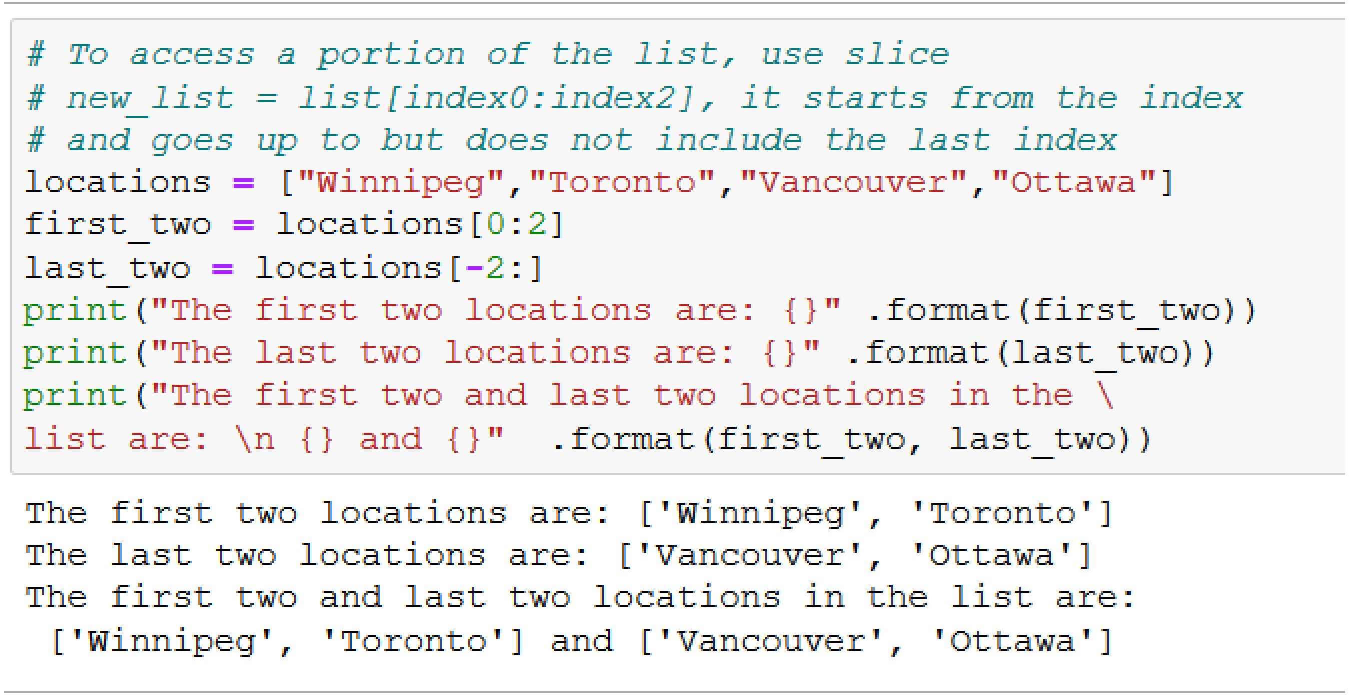
```
myList = [10, 8, 6, 4, 2]
del myList
print(myList)
```

```
---------------------------------------------------------------------------
NameError
<ipython-input-9-880e7bc77727> in <module>
      1 myList = [10, 8, 6, 4, 2]
      2 del myList
----> 3 print(myList)

NameError: name 'myList' is not defined
```

Lists – Operations of lists

Slices

Examples:

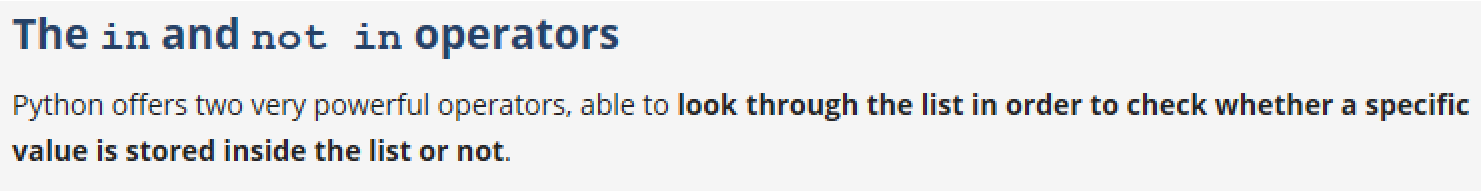


```
# To access a portion of the list, use slice
# new_list = list[index0:index2], it starts from the index
# and goes up to but does not include the last index
locations = ["Winnipeg","Toronto","Vancouver","Ottawa"]
first_two = locations[0:2]
last_two = locations[-2:]
print("The first two locations are: {}" .format(first_two))
print("The last two locations are: {}" .format(last_two))
print("The first two and last two locations in the \
list are: \n {} and {}"  .format(first_two, last_two))
```

```
The first two locations are: ['Winnipeg', 'Toronto']
The last two locations are: ['Vancouver', 'Ottawa']
The first two and last two locations in the list are:
 ['Winnipeg', 'Toronto'] and ['Vancouver', 'Ottawa']
```

Lists – Operations of lists

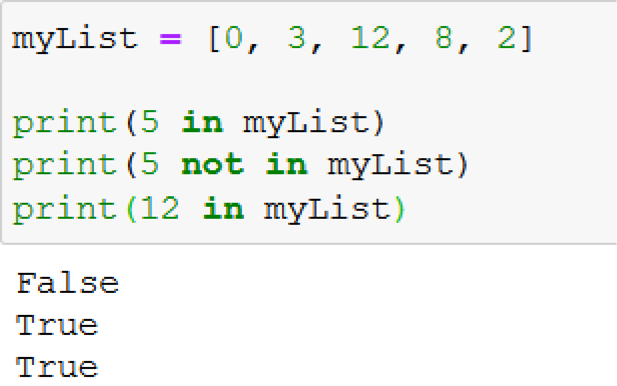
in, not in



The `in` and `not in` operators

Python offers two very powerful operators, able to **look through the list in order to check whether a specific value is stored inside the list or not**.

Example:



```
myList = [0, 3, 12, 8, 2]

print(5 in myList)
print(5 not in myList)
print(12 in myList)
```

```
False
True
True
```

cisco