## FOR LOOPS

## of the For Loop

As we mentioned earlier, the Python for loop is an iterator based for loop. It steps through the items of lists, tuples, strings, the keys of dictionaries and other iterables. The Python for loop starts with the keyword "for" followed by an arbitrary variable name, which will hold the values of the following sequence object, which is stepped through. The general syntax looks like this:

for <variable> in <sequence>:

<statements>

else:

<statements>

The items of the sequence object are assigned one after the other to the loop variable; to be precise the variable points to the items. For each item the loop body is executed.

Example of a simple for loop in Python:

languages = ["C", "C++", "Perl", "Python"]

**for** language **in** languages:

**print**(language)

### **OUTPUT:**

C

C++

Perl

Python

The else block is special; while Perl programmer are familiar with it, it's an unknown concept to C and C++ programmers. Semantically, it works exactly as the optional else of a while loop. It will be executed only if the loop hasn't been "broken" by a break statement. So it will only be executed, after all the items of the sequence in the header have been used.

If a break statement has to be executed in the program flow of the for loop, the loop will be exited and the program flow will continue with the first statement following the for loop, if there is any at all. Usually break statements are wrapped into conditional statements, e.g.

edibles = ["bacon", "spam", "eggs", "nuts"]

**for** food **in** edibles:

**if** food == "spam":

**print**("No more spam please!")

**break**

**print**("Great, delicious " + food)

**else**:

**print**("I am so glad: No spam!")

**print**("Finally, I finished stuffing myself")

### **OUTPUT:**

Great, delicious bacon

No more spam please!

Finally, I finished stuffing myself

Removing "spam" from our list of edibles, we will gain the following output:

$ python for.py

Great, delicious bacon

Great, delicious eggs

Great, delicious nuts

I am so glad: No spam!

Finally, I finished stuffing myself

$

Maybe, our disgust with spam is not so high that we want to stop consuming the other food. Now, this calls the continue statement into play . In the following little script, we use the continue statement to go on with our list of edibles, when we have encountered a spam item. So continue prevents us from eating spam!

edibles = ["bacon", "spam", "eggs","nuts"]

**for** food **in** edibles:

**if** food == "spam":

**print**("No more spam please!")

**continue**

**print**("Great, delicious " + food)

**print**("Finally, I finished stuffing myself")

### **OUTPUT:**

Great, delicious bacon

No more spam please!

Great, delicious eggs

Great, delicious nuts

Finally, I finished stuffing myself

## The range() Function

The built-in function range() is the right function to iterate over a sequence of numbers. It generates an iterator of arithmetic progressions: Example:

**range**(5)

### **OUTPUT:**

range(0, 5)

This result is not self-explanatory. It is an object which is capable of producing the numbers from 0 to 4. We can use it in a for loop and you will see what is meant by this:

**for** i **in** **range**(5):

**print**(i)

### **OUTPUT:**

0

1

2

3

4

range(n) generates an iterator to progress the integer numbers starting with 0 and ending with (n -1). To produce the list with these numbers, we have to cast range() with the list(), as we do in the following example.

**list**(**range**(10))

### **OUTPUT:**

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

range() can also be called with two arguments:

range(begin, end)

The above call produces the list iterator of numbers starting with begin (inclusive) and ending with one less than the number end.

Example:

**range**(4, 10)

### **OUTPUT:**

range(4, 10)

**list**(**range**(4, 10))

### **OUTPUT:**

[4, 5, 6, 7, 8, 9]

So far the increment of range() has been 1. We can specify a different increment with a third argument. The increment is called the step. It can be both negative and positive, but not zero:

range(begin,end, step)

Example with step:

**list**(**range**(4, 50, 5))

### **OUTPUT:**

[4, 9, 14, 19, 24, 29, 34, 39, 44, 49]

It can be done backwards as well:

**list**(**range**(42, -12, -7))

### **OUTPUT:**

[42, 35, 28, 21, 14, 7, 0, -7]

The range() function is especially useful in combination with the for loop, as we can see in the following example. The range() function supplies the numbers from 1 to 100 for the for loop to calculate the sum of these numbers:

n = 100

**sum** = 0

**for** counter **in** **range**(1, n+1):

**sum** = **sum** + counter

**print**("Sum of 1 until %d: %d" % (n, **sum**))

### **OUTPUT:**

Sum of 1 until 100: 5050Calculation of the Pythagorean Numbers

Generally, it is assumed that the Pythagorean theorem was discovered by Pythagoras that is why it has his name. However, there is a debate whether the Pythagorean theorem might have been discovered earlier or by others independently. For the Pythagoreans, - a mystical movement, based on mathematics, religion and philosophy, - the integer numbers satisfying the theorem were special numbers, which had been sacred to them.

These days Pythagorean numbers are not mystical anymore. Though to some pupils at school or other people, who are not on good terms with mathematics, they may still appear so.

So the definition is very simple: Three integers satisfying a2+b2=c2 are called Pythagorean numbers.

The following program calculates all pythagorean numbers less than a maximal number. Remark: We have to import the math module to be able to calculate the square root of a number.

**from** **math** **import** sqrt

n = **int**(**input**("Maximal Number? "))

**for** a **in** **range**(1, n+1):

**for** b **in** **range**(a, n):

c\_square = a\*\*2 + b\*\*2

c = **int**(sqrt(c\_square))

**if** ((c\_square - c\*\*2) == 0):

**print**(a, b, c)

### **OUTPUT:**

3 4 5

5 12 13

6 8 10

7 24 25

8 15 17

9 12 15

9 40 41

10 24 26

11 60 61

12 16 20

12 35 37

13 84 85

14 48 50

15 20 25

15 36 39

16 30 34

16 63 65

18 24 30

18 80 82

20 21 29

20 48 52

20 99 101

21 28 35

21 72 75

24 32 40

24 45 51

24 70 74

25 60 65

27 36 45

28 45 53

28 96 100

30 40 50

30 72 78

32 60 68

33 44 55

33 56 65

35 84 91

36 48 60

36 77 85

39 52 65

39 80 89

40 42 58

40 75 85

40 96 104

42 56 70

45 60 75

48 55 73

48 64 80

48 90 102

51 68 85

54 72 90

56 90 106

57 76 95

60 63 87

60 80 100

60 91 109

63 84 105

65 72 97

66 88 110

69 92 115

72 96 120

80 84 116

## Iterating over Lists with range()

If you have to access the indices of a list, it doesn't seem to be a good idea to use the for loop to iterate over the lists. We can access all the elements, but the index of an element is not available. However, there is a way to access both the index of an element and the element itself. The solution lies in using range() in combination with the length function len():

fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21]

**for** i **in** **range**(**len**(fibonacci)):

**print**(i,fibonacci[i])

**print**()

### **OUTPUT:**

0 0

1 1

2 1

3 2

4 3

5 5

6 8

7 13

8 21

Remark: If you apply len() to a list or a tuple, you get the number of elements of this sequence.

If you loop over a list, it's best to avoid changing the list in the loop body. Take a look at the following example:

colours = ["red"]

**for** i **in** colours:

**if** i == "red":

colours += ["black"]

**if** i == "black":

colours += ["white"]

**print**(colours)

### **OUTPUT:**

['red', 'black', 'white']

To avoid these side effects, it's best to work on a copy by using the slicing operator, as can be seen in the next example:

colours = ["red"]

**for** i **in** colours[:]:

**if** i == "red":

colours += ["black"]

**if** i == "black":

colours += ["white"]

**print**(colours)

### **OUTPUT:**

['red', 'black']

We still might have done something, we shouldn't have done. We changed the list "colours", but our change didn't have any effect on the loop. The elements to be looped remained the same during the iterations.

## Exercises with for Loops

### Exercise 1

This exercise is about the Ramanujan-Hardy number. There is a little anecdote of the Mathematician G.H. Hardy when he visited Indian mathematician Srinivasa Ramanujan in hospital. It goes like this:

I remember once going to see him when he was ill at Putney. I had ridden in taxi cab number 1729 and remarked that the number seemed to me rather a dull one, and that I hoped it was not an unfavourable omen. "No," he replied, "it is a very interesting number; it is the smallest number expressible as the sum of two cubes in two different ways."

For this reason 1732 is known as the Ramanujan-Hardy number.

Can you verify this with a Python program?

### Exercise 2

1729 is the lowest number which can be represented by a Loeschian quadratic form  in four different ways, with positive integers a and b .

## Solutions

### Solution to Exercise 1

**import** **math**

number = 1729

n = **int**(number \*\* (1/3))

cubes = {}

**for** i **in** **range**(n+1):

**for** j **in** **range**(i):

result = i \*\* 3 + j \*\* 3

**if** result **in** cubes:

cubes[result].append((i, j))

**else**:

cubes[result] = [(i,j)]

**if** result > number:

**break**

**for** x **in** cubes:

**if** **len**(cubes[x]) > 1:

**print**(x, cubes[x])

### **OUTPUT:**

1729 [(10, 9), (12, 1)]

### Solution to Exercise 2

**import** **math**

number = 1729

n = **int**(number \*\* (1/2))

results = {}

**for** a **in** **range**(n+1):

**for** b **in** **range**(a):

result = a\*\*2 + a\*b + b\*\*2

**if** result **in** results:

results[result].append((a, b))

**else**:

results[result] = [(a,b)]

**if** result > number:

**break**

**for** x **in** results:

**if** **len**(results[x]) > 3:

**print**(x, results[x])

### **OUTPUT:**

1729 [(25, 23), (32, 15), (37, 8), (40, 3)]