Chapter 1

Iterations

In programming, iterating means repeating some part of your program. This lesson presents basic programming constructions that allow iterations to be performed: "for" and "while" loops.

1.1. For loops

If you want to repeat some operations a given number of times, or repeat them for each element in some collection, a "for" loop is the right tool to use. Its syntax is as follows:

1.1: For loop syntax

```
for some_variable in range_of_values:
    loop_body
```

The for loop repeats `loop_body` for each value in turn from the `range_of_values`, with the current value assigned to `some_variable`. In its simplest form, the range of values can be a range of integers, denoted by: `range(lowest, highest + 1)`. For example, the following loop prints every integer from 0 to 99:

```
for i in range(0, 100):
    print i
```

Looping over a range of integers starting from 0 is a very common operation. (This is mainly because arrays and Python lists are indexed by integers starting from 0; see Chapter 2 Arrays for more details.) When specifying the range of integers, if the starting value equals zero then you can simply skip it. For example, the following loop produces exactly the same result as the previous one:

```
for i in range(100):
    print i
```

Example: We are given some positive integer n. Let's compute the factorial of n and assign it to the variable `factorial`. The factorial of n is $n! = 1 \cdot 2 \cdot \ldots \cdot n$. We can obtain it by starting with 1 and multiplying it by all the integers from 1 to n.

```
factorial = 1
for i in range (1, n + 1):
    factorial *= i
```

Example: Let's print a triangle made of asterisks ('$\star$') separated by spaces. The triangle should consist of n rows, where n is a given positive integer, and consecutive rows should contain 1, 2, ..., n asterisks. For example, for $n = 4$ the triangle should appear as follows:

```
*
* *
* * *
* * * *
```

We need to use two loops, one inside the other: the outer loop should print one row in each step and the inner loop should print one asterisk in each step[2].

```python
for i in range(1, n + 1):
    for j in range(i):
        print '*',
    print
```

The `range` function can also accept one more argument specifying the step with which the iterated values progress. More formally, `range(start, stop, step)` is a sequence of values beginning with `start`, whose every consecutive value is increased by `step`, and that contains only values smaller than `stop` (for positive `step`; or greater than `stop` for negative `step`). For example, `range(10, 0, -1)` represents sequence 10, 9, 8, ..., 1. Note that we cannot omit `start` when we specify `step`.

Example: Let's print a triangle made of asterisks ('$\star$') separated by spaces and consisting of n rows again, but this time upside down, and make it symmetrical. Consecutive rows should contain $2n - 1$, $2n - 3$, ..., 3, 1 asterisks and should be indented by 0, 2, 4, ..., $2(n - 1)$ spaces. For example, for $n = 4$ the triangle should appear as follows:

```
* * * * * * *
  * * * * *
    * * *
      *
```

The triangle should have n rows, where n is some given positive integer.

This time we will use three loops: one outer and two inner loops. The outer loop in each step prints one row of the triangle. The first inner loop is responsible for printing the indentations, and the second for printing the asterisks.

```python
for i in range(n, 0, -1):
    for j in range(n - i):
        print ' ',
    for j in range(2 * i - 1):
        print '*',
    print
```

1.2. While loops

The number of steps in a for loop, and the values over which we loop, are fixed before the loop starts. What if the number of steps is not known in advance, or the values over which

[2]There is a clever idiom in Python denoting a string repeated a number of times. For example, `'*' * n` denotes a string comprising n asterisks. But to make our examples more instructive, we will not use this idiom here. Instead we will use the `print` statement ended with a comma. It doesn't print the newline character, but follows the output with a single space.

we loop are generated one by one, and are thus not known in advance either? In such a case, we have to use a different kind of loop, called a "while" loop. The syntax of the while loop is as follows:

1.2: While loop syntax

```
while some_condition:
    loop_body
```

Before each step of the loop, `some_condition` is computed. As long as its value is true[3], the body of the loop is executed. Once it becomes false, we exit the loop without executing `loop_body`.

Example: Given a positive integer n, how can we count the number of digits in its decimal representation? One way to do it is convert the integer into a string and count the characters. Here, though, we will use only arithmetical operations instead. We can simply keep dividing the number by ten and count how many steps are needed to obtain 0.

```
result = 0
while n > 0:
    n = n // 10
    result += 1
```

Example: The Fibonacci numbers[4] form a sequence of integers defined recursively in the following way. The first two numbers in the Fibonacci sequence are 0 and 1, and each subsequent number is the sum of the previous two. The first few elements in this sequence are: 0, 1, 1, 2, 3, 5, 8, 13. Let's write a program that prints all the Fibonacci numbers, not exceeding a given integer n.

We can keep generating and printing consecutive Fibonacci numbers until we exceed n. In each step it's enough to store only two consecutive Fibonacci numbers.

```
a = 0
b = 1
while a <= n:
    print a
    c = a + b
    a = b
    b = c
```

1.3. Looping over collections of values[5]

We have seen how to loop over integers. Is it possible to loop over values of other types? Yes: using a "for" loop, we can loop over values stored in virtually any kind of container. The `range` function constructs a list containing all the values over which we should loop. However, we can pass a list constructed in any other way.

[3]Note that the condition can yield any value, not only `True` or `False`. A number of values other than `False` are interpreted as false, e.g. `None`, `0`, `[]` (empty list) and `''` (empty string).

[4]You can read more about the Fibonacci numbers in Chapter 13.

[5]If you are not familiar with various built-in data structures such as lists, sets and dictionaries, you can safely skip this section. Looping over lists of values will be explained in Chapter 2.

Example: The following program:

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
        'Friday', 'Saturday', 'Sunday']
for day in days:
    print day
```

prints all the days of the week, one per line.

When we use the `range` function, we first build a list of all the integers over which we will loop; then we start looping. This is memory-consuming when looping over a long sequence. In such cases, it's advisable to use – instead of `range` – an equivalent function called `xrange`. This returns exactly the same sequence of integers, but instead of storing them in a list, it returns an object that generates them on the fly.

If you loop over a set of values, the body of the loop is executed exactly once for every value in the set; however, the order in which the values are processed is arbitrary.

Example: If we modify the above program slightly, as follows:

```
days = set(['Monday', 'Tuesday', 'Wednesday', 'Thursday',
            'Friday', 'Saturday', 'Sunday'])
for day in days:
    print day
```

we might get the days output in some strange order, e.g.:

```
Monday
Tuesday
Friday
Wednesday
Thursday
Sunday
Saturday
```

Looping over a dictionary means looping over its set of keys. Again, the order in which the keys are processed is arbitrary.

Example: The following program:

```
days = {'mon': 'Monday', 'tue': 'Tuesday', 'wed': 'Wednesday',
        'thu': 'Thursday', 'fri': 'Friday', 'sat': 'Saturday',
        'sun': 'Sunday'}
for day in days:
    print day, 'stands for', days[day]
```

might output e.g.:

```
wed stands for Wednesday
sun stands for Sunday
fri stands for Friday
tue stands for Tuesday
mon stands for Monday
thu stands for Thursday
sat stands for Saturday
```

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.

Chapter 2

Arrays

Array is a data-structure that can be used to store many items in one place. Imagine that we have a list of items; for example, a shopping list. We don't keep all the products on separate pages; we simply list them all together on a single page. Such a page is conceptually similar to an array. Similarly, if we plan to record air temperatures over the next 365 days, we would not create lots of individual variables, but would instead store all the data in just one array.

2.1. Creating an array

We want to create a shopping list containing three products. Such a list might be created as follows:

```
shopping = ['bread', 'butter', 'cheese']
```

(that is, `shopping` is the name of the array and every product within it is separated by a comma). Each item in the array is called an element. Arrays can store any number of elements (assuming that there is enough memory). Note that a list can be also empty:

```
shopping = []
```

If planning to record air temperatures over the next 365 days, we can create in advance a place to store the data. The array can be created in the following way:

```
temperatures = [0] * 365
```

(that is, we are creating an array containing 365 zeros).

2.2. Accessing array values

Arrays provide easy access to all elements. Within the array, every element is assigned a number called an index. Index numbers are consecutive integers starting from 0. For example, in the array `shopping = ['bread', 'butter', 'cheese']`, 'bread' is at index 0, 'butter' is at index 1 and 'cheese' is at index 2. If we want to check what value is located at some index (for example, at index 1), we can access it by specifing the index in square brackets, e.g. `shopping[1]`.

2.3. Modifying array values

We can change array elements as if they were separate variables, that is each array element can be assigned a new value independently. For example, let's say we want to record that on the 42nd day of measurement, the air temperature was 25 degrees. This can be done with a single assignment:

```
temperatures[42] = 25
```

If there was one more product to add to our shopping list, it could be appended as follows:

```
shopping += ['eggs']
```

The index for that element will be the next integer after the last (in this case, 3).

2.4. Iterating over an array

Often we need to iterate over all the elements of an array; perhaps to count the number of specified items, for example. Knowing that the array contains of N elements, we can iterate over consecutive integers from index 0 to index $N-1$ and check every such index. The length of an array can be found using the `len()` function. For example, counting the number of items in shopping list can be done quickly as follows:

```
N = len(shopping)
```

Let's write a function that counts the number of days with negative air temperature.

2.1: Negative air temperature.

```
def negative(temperatures):
    N = len(temperatures)
    days = 0
    for i in xrange(N):
        if temperatures[i] < 0:
            days += 1
    return days
```

Instead of iterating over indexes, we can iterate over the elements of the array. To do this, we can simply write:

```
for item in array:
    ...
```

For example, the above solution can be simplified as follows:

2.2: Negative air temperature — simplified.

```
def negative(temperatures):
    days = 0
    for t in temperatures:
        if t < 0:
            days += 1
    return days
```

In the above solution, for every temperature, we increase the number of days with a negative temperature if the number is lower than zero.

2.5. Basic array operations

There are a few basic operations on arrays that are very useful. Apart from the length operation:

```
len([1, 2, 3]) == 3
```

and the repetition:

```
['Hello'] * 3 == ['Hello', 'Hello', 'Hello']
```

which we have already seen, there is also concatenation:

```
[1, 2, 3] + [4, 5, 6] == [1, 2, 3, 4, 5, 6]
```

which merges two lists, and the membership operation:

```
'butter' in ['bread', 'butter', 'cheese'] == True
```

which checks for the presence of a particular item in the array.

2.6. Exercise

Problem: Given array A consisting of N integers, return the reversed array.

Solution: We can iterate over the first half of the array and exchange the elements with those in the second part of the array.

2.3: Reversing an array.

```python
def reverse(A):
    N = len(A)
    for i in xrange(N // 2):
        k = N - i - 1
        A[i], A[k] = A[k], A[i]
    return A
```

Python is a very rich language and provides many built-in functions and methods. It turns out, that there is already a built-in method `reverse`, that solves this exercise. Using such a method, array A can be reversed simply by:

```python
A.reverse()
```

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.