Introduction à la programmation, lesson 3: cycles and functions

While loop

Using the while loop, we can repeat an instruction many times (while the value of a condition is True.)

```
while <condition>:
    instructions
```

The instructions are repeated while the value of the condition is True. When the value of the condition becomes False, the program exits the while loop.

Example: a program asks for an integer A, then for a non-zero integer B, and computes A/B.

```
A = input("Enter an integer A: ")
B = input("Enter an integer B: ")
while B == 0:
   B = input("B is zero. Give me another value: ")
print "A / B = ", float(A) / B
```

Be careful with infinite loops! If the value of the condition of the while loop never becomes False, the algorithm never stops. Here are two examples:

```
n = 5
while n < 10:
          print "n equals: ", n
print "The end."
```

```
while True:
          print "I continue…"
print "The end."
```

Sometimes, it is convenient to introduce a counter.

```
i = 0     #counter
while i < 10:
   print 2 ** I
   i = i + 1
print "The end."
```

Important: you must always initialize the counter. The **step** is the change of the counter at each loop. In the example above, the step is equal to 1, but we can choose the step differently:

```
i = 0 #counter
while i < 20:
    print i
    i = i + 2 #step equals 2
print "The end."
```

```
i = 10 #counter, here initialized with 10
while i > 0:
    print i
    i = i – 1 #step equals -1
print "The end."
```

Exercise 1. Write a program that asks to enter an integer n and then prints out the sum of integers from 1 to n (use the while loop).

The keyword **break** allows to exit the while loop immediately.

```
i = 1
while i < 100:
    if i % 2 == 0: #if i is even
        print "%d " %i
        break
    i = i + 1
print "The end."
```

The keyword **continue** allows to restart the while loop immediately.

```
i = 1
while i < 100:
    if i % 2 == 0:
        print "%d " %i
        continue
    i = i + 1
print "The end."
```

Try to avoid using **break** and **continue** as they make your code less readable.

Exercise 2. Define a list L of integers, and write a program that creates and prints out a list L' containing all the integers of L starting from the first one and until the first integer larger than 100. For example, if L = [1, 2, 1, 100, 5, 4], the program must create L' = [1, 2, 1, 100].

Functions

Exercise 3. Print out the table of multiplication by 3, 5, and 8.

We will now learn how to solve Exercise 3 in much more elegant way using functions. To create a function, we use the following syntax:

```
def name_of_function(parameter1, parameter2, parameter3, parameterN):
   instructions
```

The first line contains:

- The keyword **def**, that stands for “define”
- The name of the function. It must be unique.
- A list of parameters separated with commas.

Here is a function that prints out the table of multiplication by an integer n.

```
def multiplication_table(n):
   i = 0 #a counter
   while i < 10:
      print “%d * %d = % d” % (i + 1, n, (i + 1) * n)
      i = i + 1
```

We can now use this function to solve Exercise 3 by adding the following three lines:

```
multiplication_table(3)
multiplication_table(5)
multiplication_table(7)
```

Some functions must compute a value that will be later used by the program, and in this case they need to communicate the computed value. To this end, we can use the keyword **return**. For example,

```
def linear(a, b, n):
    return a * n + b

a, b = 1, 1
i = 0
while i < 10:
      print “x = %d, y = %d” % (i, linear(a, b, i))
```

Attention: The **return** instruction must be the last one in the definition of the function.

Exercise 4. Write a function add_prefix that receives two strings, a word and a prefix, and returns the concatenation of the prefix and of the word. For example, if you give it “construction” and “re”, it must return “reconstruction”.

```
def add_prefix(word, prefix):
    #add your code here

print add_prefix(“construction”, “re”)
```

Exercise 5. Write a function that receives the voltage V in volts and the current I in amps, and returns the resistance R in ohms. As a reminder, V = I R.

```
def resistance(V, I):
    #add your code here
    return R

print resistance(5.0, 2.0)
```

Exercise 6. Write a function string_power that receives a string S and a number n, and returns a string obtained by concatenating S n times. For example, if S = “bla” and n = 3, your function must return “blablabla”.

```
def string_power(S, n):
    #add your code here

print string_power(“bla”, 3)
```

Cycle for

The for cycle allows iterating over lists, letters in a string, etc. Let us start with an example, and note the usage of the keyword **in**:

```
list = [2,4,6,8,10, "toto"]
for element in list:
    print element
```

Take a look at how your text editor highlights the code, it will help understanding. Note also that the code below produces exactly the same result:

```
list = [2,4,6,8,10, "toto"]
for x in list:
    print x
```

The **for** cycle can also be used to iterate over the characters of a string.

```
name = "Felipe Juan Pablo Alfonso de Todos los Santos"
print “My name is spelled as: ”
for letter in name:
   print letter
```

We will now consider another way of using the cycle **for**. This time, we will use one more keyword **range**.

```
for x in range(1, 10, 3):
    print x
```

Execute this code and try to understand what it does. Formally:
range(start, end, step):

- is a function that takes three integer parameters
- and generates a list of integers between start and (end - 1) with the given step.

Here are two more examples:

```
for x in range(12, 24, 5):
    print x
```

```
for x in range(20, 10, -2):
    print x
```

Attention: The list generated by the function range can be empty.

```
for x in range (100, 110, -2):
    print x
```

Attention: parameters start and step are optional.

- **range(a)** generates a list of integers between 0 and (a - 1) with a step 1.
- **range(b, c)** generates a list of integers between b and (c - 1) with a step 1.

Exercise 7. In this exercise, you must write a function and then test it on an example.

- sum_even: receives a list of integers and returns the sum of all even integers in the list.
- number_even: receives a list of integers and returns the number of even integers in the list.
- first_odd: receives a list of integers and returns the first odd number in the list.
- min_odd: receives a list of integers and returns the smallest odd number in the list.
- index_of_element: receives a list of integers and an integer and returns the position of this integer in the list. If the integer does not appear in the list, return None.
- extract_even: receives a list of integers L and returns a list L' containing all even integers in L.
- add_to_list: receives a list of integers L and an integer d and returns a list L' such that L'[i] = L[i] + d.
- cut: receives a list of integers L and an integer d, and returns two lists, the first containing all elements of L smaller than d, and the second all elements of L equal or larger than d.