Control Structures

Fabrice Rossi

Lecture notes

A standard execution of a Python program is different from a shell interaction for two main reasons: the program executes completely (rather than in a sentence by sentence way) and it generally does not display any result, unless told explicitly to do so.

To have a Python program print a result, use the `print` function. It prints it arguments separated by spaces.

To have a Python program stops for the user to input something, use the `input` function. It prints it argument (at most one argument) and then waits for the user to type something in. It then resumes executing the program by returning the text typed by the user.

Exercise 1 (*Basic input and output*)

Question 1 Type and run the following program (comments can be ignored):

```
# -*- coding: utf-8 -*-
name = input("What's your name? ")
print('Hello', name)
```

Question 2 Determine the type of the object return by the `input` function. Does it differ based on the text typed by the user?

Solution

The type is always `<class 'str'>` and does not depend on the text typed by the user.

Question 3 Write a program that asks to the user 2 numbers and prints the sum of those numbers.

Solution

```
# -*- coding: utf-8 -*-
first_number = int(input('first number= '))
second_number = int(input('second number= '))
print('the sum of', first_number, 'and', second_number, 'is',
      first_number + second_number)
```

Lecture notes

In general the processing done by a program depends on its inputs and on other conditions

undetermined when the program is written. A program must therefore react and adapt to the actual conditions it faces. This is done by testing for some properties and by executing different commands depending on the results of the tests. The main way to achieve this behavior in Python is the following construct

```
if boolean expression:
   something to do
   when the expression evals to True
else:
   something to do
   when the expression evals to False
```

Notice that the lines associated to the two parts of the construct are shifted and aligned to the right. This is mandatory in Python. The second part of the construct (i.e., the `else:` part) is optional.

Exercise 2 (*Selections*)

Question 1 Type and run the following program (comments can be ignored):

```
# -*- coding: utf-8 -*-
command = input("Command? " )
if len(command) > 3 :
    print("Your command:", command[0:4])
else:
    print("Your command:", command)
```

Question 2 Modify the program in such a way that it prints the number of characters that are removed from the text of the command when it is too long. A typical interaction could be

```
Command? abcdefgh
Your command: abcd
4 characters were ignored.
```

Solution

```
# -*- coding: utf-8 -*-
command = input("Command? " )
if len(command) > 4 :
    print("Your command:", command[0:4])
    print(len(command) - 4, 'characters were ignored')
else:
    print("Your command:", command)
```

Question 3 Make sure that the modified program use proper English by handling in a different way the situation were only one character was ignored than when more than one were ignored.

Solution

```
# -*- coding: utf-8 -*-
```

```
command = input("Command? " )
if len(command) > 5:
    print("Your command:", command[0:4])
    print(len(command) - 4, 'characters were ignored')
elif len(command)==5:
    print("Your command:", command[0:4])
    print('one character was ignored')
else:
    print("Your command:", command)
```

Question 4 Write a new program that asks for a number and prints its absolute value without using the `abs` function.

Solution

```
# -*- coding: utf-8 -*-
x = float(input('Please enter a number: '))
absx = x
if x<0:
    absx = -x
print('The absolute value of',x , 'is', absx)
```

Lecture notes

In numerous programs, some instructions need to be performed several times. This is done via constructs called *loops*. Python proposes two such constructs. The first one is the `for` loop:

```
for variable in something:
    do something
    repeatedly
```

In order to execute a for loop, Python needs the `something` object to by *iterable*. Intuitively, this means that the object can be decomposed into several objects which comes in a certain order. Then the execution consists in executing the content of the loop for each of the "internal" objects in their order. Each execution is done with `variable` referring to the corresponding "internal" object.

Exercise 3 (*For Loops*)

Question 1 Type and run the following program (comments can be ignored):

```
# -*- coding: utf-8 -*-
for x in "abcdef":
    print(x)
```

Question 2 Modify the program to ask the user a text to iterate over.

Solution

```
# -*- coding: utf-8 -*-
my_text = input('Enter a (short) text: ')
```

```
for x in my_text:
    print(x)
```

Question 3 Modify again the program to ask in addition a *step* value (an integer) such that rather than printing all the characters, the program prints only characters separated by $step - 1$ other characters (starting with the first characters). Hint: use a slice. A typical interaction could be

```
Text? abcdefgh
Step? 2
a
c
e
g
```

Solution

```
# -*- coding: utf-8 -*-
my_text = input('Enter a (short) text: ')
my_step = int(input('Enter a step:' ))
for x in my_text[::my_step]:
    print(x)
```

Lecture notes

The practical usefulness of *for* loops depends on the existence of interesting *iterable* objects. A basic and very practical type of such objects is the `range` one. A range object "contains" integers specified in a simple way, using the `range` function. For instance the following program

```
for i in range(1, 4):
    print(i)
```

prints

```
1
2
3
```

More generally `range` works as follows:

- `range(end)`: all integers from 0 to `end-1`;
- `range(begin, end)`: all integers from `begin` to `end-1`;
- `range(begin, end, step)`: integers from `begin` to `end-1` with an increment of `step`.

Exercise 4 (*Ranges*)

Question 1 Write a program that asks to the user an integer n and prints k^2 for all k between 1 and n (both included).

Solution

```
# -*- coding: utf-8 -*-
n = int(input('n= '))
for k in range(1, n + 1):
    print(k**2)
```

Question 2 Write a program that asks to the user an integer n and prints the following ascii art

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

where the height of the ascii art in lines is n (here $n = 4$). Hint: use string multiplication.

Solution

```
# -*- coding: utf-8 -*-
n = int(input('Height of the pyramid= '))
for line in range(n):
    print((' ' * (n - line - 1)) + ('*' * (2 * line + 1)))
```

Exercise 5 (*Accumulating values*)

We consider the following program:

```
# -*- coding: utf-8 -*-
n = int(input('n='))
result = 0
for k in range(1, n + 1):
    result = result + k
    print(k, result)
```

Question 1 Type and run the program for different input values.

Question 2 What is the integer value in `result` at the end of the program as a function of the input value? (give a formula.)

Solution

The value is $0 + 1 + 2 + \ldots + n$, that is $\sum_{k=0}^{n} k$ (or $\sum_{k=1}^{n} k$)

Question 3 Modify the program in such a way that `result` contains $n!$ at the of the program ($n! = 1 \times 2 \times \ldots \times n$). Do not use the `factorial` function from the `math` module.

Solution

```
# -*- coding: utf-8 -*-
n = int(input('n='))
```

```
result = 1
for k in range(1, n + 1):
    result = result * k
    print(k, result)
```

Lecture notes

The second loop construct in Python is the `while` loop:

```
while boolean expression:
  do something
  repeatedly
  until the expression evaluates to False
```

Contrarily to the `for` loop, the number of iterations of the `while` loop is not known at the beginning of the loop. It might not execute at all because the expression evaluates to `False` or in the extreme inverse case it might execute forever if the expression evaluates always to `True`.

Exercise 6 (*While Loops*)

Question 1 Type and run the following program (comments can be ignored):

```
# -*- coding: utf-8 -*-
n = int(input('n='))
m = n
p = 0
while m > 0:
    p = p + 1
    m = m // 2
    print(p, m)
print(2**(p-1), n, 2**p)
```

Explains the behavior of the program when a positive integer is entered by the user.

Solution

The body of the loop divides by 2 the value designated by `m` each times it executes, taking the integer part of the result when `m` is odd. This reduces strictly `m` each time. As `m` corresponds always to an integer value, the loop terminates eventually. The content of `p` at the end of the loop is the number of times the body was executed. This is also the number of times the initial value `n` was divided by before reaching 0. This explains why `n` falls in between 2^{p-1} and 2^p.

Question 2 What happens if the user enters a negative integer?

Solution

The body of the loop does not execute and the program prints

```
0.5 n 1
```

where `n` is replaced by the value chosen by the user.

Question 3 Write a program that asks to the user an integer n and prints k^2 for all k between 1 and n (both included) without using a `for` loop.

Solution

```
# -*- coding: utf-8 -*-
n = int(input('n= '))
k = 1
while k <= n:
    print(k ** 2)
    k = k + 1
```

The `random` module provides ways to generate random numbers. In particular the `randint(a, b)` function generates a random integer between the specified bounds `a` and `b` (included).

Question 4 Write a program that chooses random integer between 1 and 200 (inclusive) and then asks repeatedly to the user to guess the number. To help the user guess quickly the number, the program will tell the user whether the propose number if larger or smaller than the number to guess.

Solution

```
# -*- coding: utf-8 -*-
import random as rd
toguess = rd.randint(1, 200)
my = 0
nbguess = 0
while my != toguess:
    my = int(input('Your guess= '))
    if my > toguess:
        print('too large')
    elif my < toguess:
        print('too small')
    nbguess = nbguess + 1
print('Congratulations! You used', nbguess, 'guesses')
```