chap03

July 23, 2020

1 Capítulo 3

1.1 Interacción con Python

Hasta el momento, todos nuestros ejemplos han sido programas que no requieren o solicitan información del usuario. Esto tiene la limitación que el programa es **estático** y siempre se va a comportar de la misma manera. Si, por ejemplo, queremos hacer una multiplicación diferente, cada vez toca escribir un nuevo programa.

Para ampliar la capacidad de los programas de Python, el programa puede solicitar información del usuario a través del teclado.

[42]:
```
# usuario_info.py
# Código que solicita información del usuario
# y reporta los resultados
#

name   = input("Cuál es su nombre? ")

txt    = input('Cuál es su altura (m)? ')
alt    = float(txt)

txt   = input('Cuanto pesa Ud. (kg)? ')
weigh = float(txt)

print ('%s, Ud. mide %4.2f m y pesa %5.1f kg'
       % (name,alt,weigh))
```

```
Cuál es su nombre? Germán
Cuál es su altura (m)? 1.82
Cuanto pesa Ud. (kg)? 87
Germán, Ud. mide 1.82 m y pesa  87.0 kg
```

El comando `input` es el encargado de solicitar al usuario que debe digitar una respuesta con el teclado. El programa no continúa a menos que el usuario oprime `Enter`.

Para los valores solicitados de altura o peso, el comando

`txt    = input('Cuál es su altura (m)? ')`

genera una variable `txt` con caracteres (una variable tipo `string`). La función `float(txt)` convierte

la variable `txt` en una variable real o `float`. Note que si el usuario responde con letras o símbolos, el programa generará un error, es decir **se espera** un número como respuesta.

1.2 Entrada con el teclado

Como se ve en el programa anterior, solicitar información del usuario es bastante sencillo en Python.

1.2.1 Hazlo tu mismo

El siguiente programa muestra una variación al programa `multint.py` de tal forma que los números a ser multiplicados sean definidos por el usuario.

[9]:
```
# usuariomult.py
# Código para multiplicar dos números enteros,
# definidos por el usuario.

a = int(input("Digite un primer número entero:  "))
b = int(input("Digite un sefundo número entero: "))

c = a*b

print(a,"x",b," = ",c)
```

```
Digite un primer número entero:  3
Digite un sefundo número entero: 2
3 x 2  =  6
```

Este código tiene dos llamadas para pedirle al usuario que digite los números a multiplicar. También es importante notar que al poner

```
a = int(input("Enter an integer: "))
```

se le pide al usuario un número entero, por lo que si el usuario digita un numero real (2.2) o caracteres diferentes a números, el programa mostrará un error. Generalmente Python es bastante claro explicando el error y el problema que se generó.

1.2.2 Hazlo tu mismo

El programa anterior tiene una desventaja, el usuario se le solicita la pareja de números uno a uno. Genere una versión más corta, donde se le solicita al usuario los dos números con una sola llamada. El usuario debe digitar los números con un espacio entre ellos, no con coma (,), ni otro separador. Tampoco puede digitar `Enter` entre los números, pues esto generará un error.

[14]:
```
# usuariomult2.py
# Código para multiplicar dos números enteros,
# definidos por el usuario.
# Una sola solicitud para los dos números.
#

intxt = input("Digite dos números enteros: ")
```

```
a,b = intxt.split()
a = int(a)
b = int(b)

c = a*b

print(a,"x",b," = ",c)
```

```
Digite dos números enteros: 2 3
2 x 3  =  6
```

El comando `input` permite digitar una serie de caracteres. Si éstos caracteres están separados por espacios, se pueden asignar a multiples variables con el comando `intxt.split()`. Sin embargo, si el usuario digita tres números (en vez de dos) se genera un error. Puede pensar en alguna alternativa para eviar este error?

1.3 *for* y *while* loops, condicionales *if*

El programa `usuariomult2.py` sigue mostrando limitaciones. Aunque el usuario puede hacer la multiplicación, tiene que volver a correr el programa si quiere cambiar los números. ¿Qué hacer si queremos hacer más de una operación?

1.3.1 Hazlo tu mismo

Genere un programa que solicite continuamente al usuario por 2 números hasta que el usuario quiera detenerse.

[15]:
```
# usuariomult3.py
# Código para multiplicar dos números enteros,
# definidos por el usuario.
# Operación se repite hasta que el usuario lo determine.
#

for i in range(1000):
    intxt = input('Digite dos números enteros (ceros para parar) ')
    a,b   = intxt.split()
    a     = int(a)
    b     = int(b)
    if (a==0 and b==0):
        break
    c = a*b
    print(a,"x",b," = ",c)
```

```
Digite dos números enteros (ceros para parar) 2 5
2 x 5  =  10
Digite dos números enteros (ceros para parar) 7 9
7 x 9  =  63
Digite dos números enteros (ceros para parar) 1 2
```

```
1 x 2  =  2
Digite dos números enteros (ceros para parar) 0 0
```

Python no permite hacer *loops* de manera indefinida (un infinito número de veces), por lo que hay que decirle que lo haga en un rango definido (hasta 1000 veces en nuestro ejemplo).

Pero sería muy ilógico que el usuario tenga que hacer la operación mil veces para que el programa termine. Por eso, tenemos *condicionales*, donde al cumplirse cierta condición (que ambos números sean 0), el programa hace un `break` que le ordena salir del *loop*.

Note que el código que pertenece al *loop* está indentado.

1.4 Operadores de relación en varios lenguajes

F77	F90	C	MATLAB	Python	meaning
.eq.	==	==	==	==	equals
.ne.	/=	!=	~=	!=	does not equal
.lt.	<	<	<	<	less than
.le.	<=	<=	<=	<=	less than or equal to
.gt.	>	>	>	>	greater than
.ge.	>=	>=	>=	>=	greater than or equal to
.and.	.and.	&&	&	and	and
.or.	.or.	||	|	or	or

En Python, el `for` *loop* procesa cada *item* dentro de una secuencia, así que puede ser usado en cualquier secuencia de datos de diferente tipo (arreglos, strings, listas, tuples, etc.). La variable del *loop* (`i` en nuestro ejemplo se le asigna en cada iteración el valor correspondiente de la variable, y el cuerpo (indentado) dentro del *loop* es ejecutado (donde `i` tiene unsólo valor).

La forma general de un `for` *loop* en Python es:

```
for LOOP_VARIABLE in SEQUENCE:
   STATEMENTS
```

Note que los comandos tienen un encabezado (header) que termina con dos puntos (:) y un cuerpo con una serie de comandos que se encuentran **indentados**.

La indentación puede ser 1, 2, o cualquier número de espacios (o un *tab*), pero siempre del mismo tipo o cantidad a la derecha del encabezado.

1.4.1 El comando `while`

En lenguajes modernos se tiene la opción de utilizar el comando `while`, en lugar de un `for` *loop*. El `while` es un comando compuesto, que tiene un header y un cuerpo, y tiene el siguiente formato

general

```
while BOOLEAN_EXPRESSION:
   STATEMENTS
```

El `while` se ejecuta de manera continua siempre y cuando la expresión `BOOLEAN_EXPRESSION` sea cierta. *Boolean* en este caso se puede pensar como una función lógica o condicional. El programa anterior se puede escribir:

1.4.2 Hazlo tu mismo

Genere un programa que solicite continuamente al usuario por 2 números hasta que el usuario quiera detenerse, pero ahora con un `while` loop.

[16]:
```
# usuariomult4.py
# Código para multiplicar dos números enteros,
# definidos por el usuario.
# Operación se repite con while loop.
#

a = 1
b = 1

while (a!=0 or b!=0):
    intxt = input('Digite dos números enteros (ceros para parar) ')
    a,b   = intxt.split()
    a     = int(a)
    b     = int(b)
    c     = a*b
    print(a,"x",b," = ",c)
```

```
Digite dos números enteros (ceros para parar) 2 5
2 x 5  =  10
Digite dos números enteros (ceros para parar) 7 9
7 x 9  =  63
Digite dos números enteros (ceros para parar) 1 2
1 x 2  =  2
Digite dos números enteros (ceros para parar) 0 0
0 x 0  =  0
```

Nota que para iniciar el `while` loop, es necesario tener definidos `a` y `b`, para que el programa pueda realizar la verificación del condicional la primera vez.

1.4.3 `for` vs `while` loops

Existen entonces dos tipos de *loop*. Para el programador clásico (me incluye acá), el `for` *loop* parece más sencillo y lógico. Pero cuál escoger?

- Si Ud. sabe de antemano el número de iteraciones que debe hacer, o va a hacer un *loop* sobre una lista o arreglo donde el número total de elementos es conocido, el **`for` loop** es su mejor opción.

- Si debe realizar una iteración de un cálculo hasta que una condición se cumple, y no se puede saber cuando esa condición se va a cumplir, el **`while` loop** es su mejor opción.

1.5 Múltiples condicionales `if, elif, else`

En los ejemplos anteriores, una operación (`a*b`) se lleva a cabo si una condición se cumple. Una versión más versátil permite múltiples condiciones con la siguiente estructura

```
if (expresión_lógica):
   (bloque de código)
elif (expresión_lógica):
   (bloque de código)
elif (expresión_lógica):
   (bloque de código)
...
else:
   (bloque de código)
```

Cada bloque de código puede tener tantas lineas como se requiera. Y tantos `elif` (else if) bloques como se requiera, y como máximo un `else`.

Cuando una condición se cumple, ese bloque de código se ejecuta, sin importar los bloques siguientes. Es decir el órden importa en este caso (Python no revisa los bloques siguientes). El último `else` será ejecutado si ninguna condición anterior es verdadera.

1.5.1 Hazlo tu mismo

Un juego de adivine el número

Haga un programa donde el usuario debe adivinar un número entre 1 y 1000 y el programa le va informando en cada intento si el número introducido por el usuario es mayor o menor que el número que se busca. **Cuente el número de intentos**.

Para que el número que se debe adivinar no sea siempre el mismo, podemos que el programa genere el número de manera aleatoria. Cómo se genera un número aleatorio?

[30]:
```
# random_ej.py
# Genere números aleatorios, float y enteros
#

import numpy as np

for i in range(5):
    a = np.random.random() # dist uniforme
    b = np.random.randint(1,10)
    print ("%8.7f %2i"%(a,b))
```

```
0.4877788  2
0.6833578  6
0.3136540  9
```

```
0.8072603  2
0.0949658  8
```

El código anterior muestra dos ejemplos de generación de números aleatorios.

`a = np.random.random()`

genera un número aleatorio entre `[0.0 1.0)`, con distribución uniforme. De manera similar

`b = np.random.randint(1,10)`

genera un número entero entre los límites definidos `[1,10)`. Es decir, no incluye el 10.

[34]:
```
# adivine_entero.py
# Juego donde el usuario adivina un número
#

import numpy as np

# Genera un número aleatorio entre [1 y 1000)
number = np.random.randint(1,1000)

# Empieze el juego y cuente los intentos.
guesses  = 1
guess    = int(input('Adivine un número del 1 al 1000: '))

while guess!=number:
    guesses = guesses + 1
    if (guess>number):
        print(guess," es muy alto")
    elif (guess<number):
        print(guess," es muy bajo")

    guess = int(input("Adivine otra vez: "))

print("\nExcelente, adivinaste en ",guesses, ' intentos')
```

```
Adivine un número del 1 al 1000: 501
501  es muy alto
Adivine otra vez: 250
250  es muy alto
Adivine otra vez: 125
125  es muy alto
Adivine otra vez: 62
62  es muy bajo
Adivine otra vez: 90
90  es muy alto
Adivine otra vez: 75
75  es muy bajo
Adivine otra vez: 85
```

```
85  es muy bajo
Adivine otra vez: 90
90  es muy alto
Adivine otra vez: 88
88  es muy alto
Adivine otra vez: 87

Excelente, adivinaste en  10  intentos
```

1.5.2 Hazlo tu mismo

Genere un programa que le solicite al usuario un número **real, positivo** repetidas veces. Si el número es negativo, digale al usuario y pidale nuevamente el número real y positivo. Si el número es positivo, calcule la raiz cuadrada con la función `np.sqrt`. Si el usuario pone 0, el programa se termina.

[36]:
```
# usuarioraiz.py
# Solicite al usuario un número real y toma la raíz cuadrada.
#

import numpy as np

x = 2.
while (x!=0.0):
    x = float(input("Digite un número real y positivo: "))
    if (x<0.0):
        print('Número es negativo')
        continue
    else:
        y = np.sqrt(x)
    print ('sqrt(',x,') = ',y)
```

```
Digite un número real y positivo: 2.0
sqrt( 2.0 ) =  1.4142135623730951
Digite un número real y positivo: -1
Número es negativo
Digite un número real y positivo: 3.4
sqrt( 3.4 ) =  1.8439088914585775
Digite un número real y positivo: 0
sqrt( 0.0 ) =  0.0
```

Note el uso del nuevo comando `continue`, que le ordena a Python a que vaya directamente a la siguiente iteración (sin mirar lo que está por debajo dentro del *loop*), es decir no va a imprimir el resultado.

Otro comando `break` hace que el programa termine o salga de un `for` loop por completo. Ojo, salir del `for` loop, no termina el programa.

1.5.3 Hazlo tu mismo

Re-escriba el código usando `for` loop, con un máximo de 5 loops.

[38]:
```
# usuarioraiz2.py
# Solicite al usuario un número real y toma la raíz cuadrada.
#

import numpy as np

for i in range(5):
    x = float(input("Digite un número real y positivo: "))
    if (x<0.0):
        print('Número es negativo')
        continue
    elif (x==0.0):
        break
    else:
        y = np.sqrt(x)
    print ('sqrt(',x,') = ',y)
```

```
Digite un número real y positivo: 2.0
sqrt( 2.0 ) =  1.4142135623730951
Digite un número real y positivo: -1
Número es negativo
Digite un número real y positivo: 3.4
sqrt( 3.4 ) =  1.8439088914585775
Digite un número real y positivo: 0
```

1.6 El máximo común divisor

El máximo común divisor (o GCF greatest common factor en inglés) de dos (o más) números enteros es el mayor número entero que los divide sin dejar residuo.

1.6.1 Hazlo tu mismo

Cómo se hace?

- Tenemos dos números `a` y `b`.
- El rango de posibles números será del 1, hasta el menor de los dos números. (1 hasta min(a,b))
- Mirar para todo el rango de números posibles si la división tiene un residuo con `a` y con `b`
- Empieze con el `1`, que siempre será un común divisor.
- Siga con el 2, 3, etc. Si alguno de estos números no genera residuo, guardelo como el GCF.
- Continue hasta `min(a,b)`.

[41]:
```
# gcf.py
# encuentre el máximo común divisor (gcf)
# de dos números enteros
#
```

```
a = 1
b = 1
while (a!=0 or b!=0):
    intxt = input('Digite 2 enteros (ceros para parar) ')
    a,b   = intxt.split()
    a     = int(a)
    b     = int(b)

    amin = min(a,b)
    if (amin<1):
        break

    for j in range(1,amin+1):
        if (a%j==0 and b%j==0):
            jmax = j
    print ("Máximo común divisor = ", jmax)
```

```
Digite 2 enteros (ceros para parar) 3 2
Máximo común divisor =  1
Digite 2 enteros (ceros para parar) 7 21
Máximo común divisor =  7
Digite 2 enteros (ceros para parar) 9 24
Máximo común divisor =  3
Digite 2 enteros (ceros para parar) 923 1248
Máximo común divisor =  13
Digite 2 enteros (ceros para parar) 0 0
```

Ojo, este programa no es muy eficiente para números muy grandes

Algunas cosas nuevas `min(a,b)` calcula el mínimo entre `a` y `b`, usando funciones internas de Python. Existe también `max`. Puede tener más de dos argumentos de entrada.

`a%i` calcula el residuo de la división de `a/i`, llamado el *módulo*. En nuestro caso `a%i = 0` si `a` es divisible por `i`. Si el valor es diferente, entonces hay un residuo. Note que el órden **SI** importa.

1.6.2 Otras funciones de Python disponibles

```
abs(a)       absolute value
float(a)     conversion to real
int(a)       conversion to integer
round(a)     nearest integer
pow (x,y)    Return x to the power y
range(a,b,c) Secuencia desde a hasta b. c es el salto.
```