Capítulo 3

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. Por ejemplo, el programa `multint.py` siempre dará como resultado `6`, y si queremos hacer una multiplicación diferente, nos 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. Para esto, se le debe indicar al usuario que información dar, como en el siguiente ejemplo:

```
# user_info.py
# Simple code to ask the user some quick questions
# and report the results
#

name   = input("What is your name? ")

txt    = input('How tall are you? ')
alt = float(txt)

txt = input('How much do you weigh? ')
weigh = float(txt)

print ('%s, you are %4.2f tall and weigh %5.1f kg'
       % (name,alt,weigh))
```

que en la terminal da como resultado

```
gprieto > python user_info.py
What is your name? Germán
How tall are you? 1.80
```

```
How much do you weigh? 88
Germán, you are 1.80 tall and weigh  88.0 kg
```

3.1. Entrada con el teclado

Como se ve en el programa anterior, solicitar información del usuario es bastante sencillo en Python. El siguiente programa muestra una variación al programa `multint.py` de tal forma que los números a ser multiplicados sean definidos por el usuario.

```
# usermult.py
# Program to multiply 2 integers, user defined

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

c = a*b

print ('Product is = ', c)
```

que tiene como resultado

```
gprieto > python usermult.py
Enter an integer: 3
Enter an integer: 5
Product is =  15
```

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:

```
gprieto > python usermult.py
Enter an integer: 3.2
Traceback (most recent call last):
  File "usermult.py", line 4, in <module>
    a = int(input("Enter an integer: "))
ValueError: invalid literal for int() with base 10: '3.2'
```

donde la respuesta y explicación del error es bastante clara y muestra el problema que se generó.

El programa anterior tiene una desventaja, el usuario se le solicita la pareja de números uno a uno. Esta es 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.

```
# usermult2.py
# Program to multiply 2 numbers, user defined

intxt = input("Enter two numbers here: ")

a,b = intxt.split()
a = int(a)
b = int(b)
c = a*b

print ('Product is = ', c)
```

con el programa generando el siguiente resultado

```
gprieto > python usermult2.py
Enter two numbers here: 3 5
Product is =  15
```

3.2. for y while loops, condicionales if

Ahora queremos modificar el programa, de tal forma que se le pregunte continuamente al usuario por 2 números hasta que el usuario quiera detenerse.

```
# usermult3.py
# Program to multiply 2 numbers, user defined
# until user wants to stop

for i in range(1000):
   intxt = input("Enter two integers (zeros to stop) ")
   a,b = intxt.split()
   a = int(a)
   b = int(b)
   if (a==0 & b==0):
      break
   c = a*b
   print ('Product is = ', c)
```

Este código puede no ser el más óptimo, ya que en Python no se permite hacer `for` loops de manera indefinida, por lo que hay que decirle que lo haga en un rango definido (en este caso hasta 1000 veces). Esta es una manera muy clásica

de Fortran de hacer loops indefinidos, hasta que cierta condición se cumpla, y el programa hace un `break` que ordena a salir del `for` loop. Note que el código dentro del loop está indentado. Esto es una exigencia de Python.

El programa seguirá pidiendo al usuario los 2 números y imprimiendo el resultado, hasta que el usuario lo detenga. Una forma poco elegante de parar un programa es con CTRL+C, pero el programa solicita que si se quiere detener, los dos valores deben ser ceros (0) y se detiene de manera limpia con el condicional:

```
if (a==0 & b==0):
```

El resultado de correr el programa como ejemplo:

```
gprieto > python usermult3.py
Enter two integers (zeros to stop) 3 2
Product is =  6
Enter two integers (zeros to stop) 5 3
Product is =  15
Enter two integers (zeros to stop) 0 0
```

Acá se muestra una serie de operadores de relación en varios languajes

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, asi 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. 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 uno, dos, o cualquier número de espacios, siempre de igual cantidad a la derecha del encabezado.

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` loop se ejecuta de manera continua siempre y cuando la expresión `BOOLEAN_EXPRESSION` sea cierta. El programa anterior, se puede escribir utilizando un `while`

```
# usermult4.py
# Program to multiply 2 numbers, user defined
# until user wants to stop

a = 1
b = 1
while (a!=0 and b!=0):
   intxt = input("Enter two integers (zeros to stop) ")
   a,b = intxt.split()
   a = int(a)
   b = int(b)
   c = a*b
   print ('Product is = ', c)
```

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

3.2.1. Escoger entre `for` y `while` loops

Hay entonces dos tipos de *loops*, y para el programar clásico, el `for` loop parece más sencillo y lógico. Una buena guia para decidir que usar, es si Ud. sabe de antemano el número de iteraciones que debe hacer. O si por ejemplo va a hacer un loop sobre una lista o arreglo, donde el número total de elementos es conocido. Por ejemplo, si quiere hacer un modelo y correlo 100 veces, el `for` loop es la mejor opción.

Pero si, por ejemplo, debe realizar una iteración de un cálculo hasta que una condición se de, y no puede saber cuando esa condición se va a cumplir, una mejor opción es el `while` loop.

El primer caso se conoce como una *iteración definida*, mientras que el segundo caso es conocida como *iteración indefinida* – no sabemos de antemano cuantas iteraciones son requeridas.

3.3. Multiples condicionales `if`, `elif`, `else`

En el ejemplo anterior, una operación es llevada a cabo si una condición se cumple. Una versión más versátil permite multiples 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 líneas como sea requiera. 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 seguir mirando las otras condiciones posteriores. Es decir el órden importa en este caso (Python no revisa lo que sigue). El último `else` será ejecutado si ninguna condición anterior es verdadera.

El siguiente programa muestra un ejemplo 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.

```
# guess_number.py
# Short game for user to guess a number

import random      # Import the random module

# Get random number between [1 and 1000)
number = random.randrange(1, 1000)
guesses = 0
guess = int(input("Guess my number between 1 and 1000: "))

while guess != number:
    guesses = guesses + 1
    if guess > number:
        print(guess, "is too high.")
    elif guess < number:
        print(guess, " is too low.")
    guess = int(input("Guess again: "))

print("\nGreat, you got it in", guesses,  "guesses!")
```

Un ejemplo al correr el programa

```
gprieto > python guess_number.py
Guess my number between 1 and 1000: 500
500  is too low.
Guess again: 750
750 is too high.
Guess again: 600
600  is too low.
Guess again: 700
700  is too low.
Guess again: 725
725  is too low.
Guess again: 745
745  is too low.
Guess again: 748
748  is too low.
Guess again: 749

Great, you got it in 7 guesses!
```

Acá otra demostración de un programa que le pide un número real y positivo al usuario repetidamente. Si el número es negativo, repite la solicitud (ya que sólo queremos números positivos). Si el número es positivo, calcula la raíz cuadrada con la función `sqrt`. Si el usuario pone 0, el programa se termina. Esto se repite un máximo de 5 veces.

```
# usersqrt.py
# Ask user for number and take sqrt.

from math import *   # import math functions

for i in range(5):
   x = float(input("Enter positive real number: "))
   if (x==0.0):
      break
   elif (x<0.0):
      print("Number is negative")
      continue
   else:
      y = sqrt(x)
   print ('sqrt = ',y)
```

El programa muestra el uso de un nuevo comando, `continue`, que le dice a Python que vaya directamente a la siguiente iteración, es decir no va a imprimir el resultado `y`. El comando `break` hace que el programa salga del `for` loop por completo.

3.4. Ejemplo: Máximo común divisor

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

```
# gcf.py
# Calculate the greatest common factor (gcf) of two integers

for i in range(10):

   intxt = input("Enter two integers (zeros to stop) ")
   a,b = intxt.split()
   a = int(a)
   b = int(b)

   if (a==0 and b==0):
      break
   else:
      amin = min(a,b)
      for j in range(1,amin+1):
         if ( (a%j)==0 and (b%j)==0):
            jmax = j
   print ("Greatest common factor = ",jmax)
```

con resultados para varios números

```
gprieto > python gcf.py
Enter two integers (zeros to stop) 3 2
Greatest common factor =  1
Enter two integers (zeros to stop) 7 21
Greatest common factor =  7
Enter two integers (zeros to stop) 9 24
Greatest common factor =  3
Enter two integers (zeros to stop) 923 1248
Greatest common factor =  13
Enter two integers (zeros to stop) 0 0
```

Este programa no es muy eficiente si los números investigados son muy grandes, pero funciona muy bien para números pequeños. Algunas cosas nuevas en este programa:

`min(a,b)` calcula el mínimo entre a y b, usando funciones internas de Python. También existe la función `max`. Pueden tener más de dos argumentos de entrada.

`a%i` calcula el residuo de la división de los dós números (llamado el módulo). Para nuestro caso, $a\,\%i = 0$ si a is divisible por i. Si el valor es diferente

de cero, el valor obtenido es el residuo de dividir los dos números. Note que el orden importa.

Abajo se muestran otras funciones internas de Python disponibles (lista parcial)

```
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.
```

Problemas

3.1. Escriba un programa que imprima una lista de números del 2 al 20, y que muestra si el número es divisible por 2, por 3, por ambos o por ninguno. La salida en pantalla se puede ver algo así

```
Num   Div by 2 and/or 3?
---   ------------------
2            by 2
3            by 3
4            by 2
5            neither
6            both
7            neither
```

3.2. Escriba un programa que repetidamente le pida al usuario tres valores (reales) a, b, c, para la ecuación cuadrática

$$a * x^2 + b * x + c = 0,$$

Usando la famosa formula para las raíces de este problema, el programa debe reconocer si hay raíces reales y calcular estos valores. Como resultado, muestre el número de raíces reales y sus valores. El programa debe parar si todos los valores son 0.

3.3. Modifique el programa gfc.py para calcular el mínimo común múltiplo (least common multiple en inglés) de dos números enteros.

3.4. Python incluye un modulo para la generación de números aleatorios (llamado random). Abajo se muestra un ejemplo que imprime en pantalla 20 valores aleatorios entre 0 y 1.

```
# random_example.py
# Generate a short list of random number between 0 and 1

import random

for i in range(20):
   a = random.random()
   print (a)
```

Escriba un programa simple de Python para generar 10000 números aleatorios entre 0 y 1. Realice un test de que tan aleatorio es el generador de números, contando el número de veces que el número cae dentro de 10 rangos distintos entre 0 y 1 (por ejemplo, 0 a 0.1, 0.1 a 0.2, etc.). Imprima el número total de veces para cada rango en la pantalla. Este es un ejemplo de cómo se puede ver la salida del programa:

```
0 1009
1 1048
2 1001
3 1038
4 1008
5 959
6 993
7 925
8 1017
9 1002
```

AYUDA: Cree un arreglo de números enteros (mire el modulo `numpy`) con 10 elementos e inicie con ceros. Para cada número aleatorio, sume una unidad al rango apropiado y así poder adicionar todos los números.

NOTA: En C o Fortran cada vez que corra el programa, Ud. obtendrá siempre la misma serie de números aleatorios. Esto no es muy buena idea, pero el código para obtener series de números aleatorios distintas cada vez que se corre el programa es complicado. Python fija la semilla del número aleatorio (*seed*) con el reloj del computador, así que no debe ser un problema.