Tema 0 - Introducción a `python`- 3

Semestre 2018-2

M. en C. Gustavo Contreras Mayén

M. en C. Abraham Lima Buendía

Facultad de Ciencias - UNAM

9 de febrero de 2018

Funciones

Con lo que hemos revisado sobre `python`, tenemos elementos para iniciar la solución de problemas, con una manera particular de agrupar un conjunto de instrucciones, a través de funciones.

Las funciones intrínsecas de cualquier lenguaje son pocas, pero podemos extenderlas con funciones definidas por el usuario.

Estructura de una función

La palabra reservada `def` se usa para definir funciones.

Debe seguirle el nombre de la función y la lista de parámetros formales entre paréntesis. Las sentencias que forman el cuerpo de la función empiezan en la línea siguiente, y deben estar con sangría.

Estructura de una función

La estructura de una función en Python es la siguiente:

```
def nombre_funcion(parametro1, ...):
    conjunto de instrucciones
    return valores_devueltos
```

Un parámetro puede ser cualquier objeto de `python`, incluyendo una función.

Estructura de una función

Los parámetros pueden darse por defecto, por lo que en la función son opcionales.

Si no se utiliza la instrucción **`return`**, la función devuelve un objeto de tipo **`None`**

Ejemplo

Código 1: Función sencilla

```
1  def cuadrados(a):
2    for i in range(len(a)):
3      a[i] = a[i]**2
4
5
6  a = [1, 2, 3, 4]
7
8  solucion = cuadrados(a)
9
10 print(solucion)
```

Cálculo de la serie de Fibonacci

La sucesión fue descrita por Fibonacci como la solución a un problema de la cría de conejos:

“Cierto hombre tenía una pareja de conejos juntos en un lugar cerrado y uno desea saber cuántos son creados a partir de este par en un año, cuando es su naturaleza parir otro par en un simple mes, y en el segundo mes los nacidos parir también”

Cálculo de la serie de Fibonacci

La sucesión fue descrita por Fibonacci como la solución a un problema de la cría de conejos:

“Cierto hombre tenía una pareja de conejos juntos en un lugar cerrado y uno desea saber cuántos son creados a partir de este par en un año, cuando es su naturaleza parir otro par en un simple mes, y en el segundo mes los nacidos parir también”

¿Cómo le hacemos?

Propuesta de código

Código 2: Primer intento para la serie de Fibonacci

```
1 def fib(n):
2     a, b = 0, 1
3     while b < n:
4         print (b)
5         a, b = b, a + b
6
7
8 f = fib(2000)
9 print(f)
```

Propuesta de código

Código 3: Primer intento para la serie de Fibonacci

```
1 def fib(n):
2     a, b = 0, 1
3     while b < n:
4         print (b)
5         a, b = b, a + b
6
7
8 f = fib(2000)
9 print(f)
```

Segunda propuesta de código

Código 4: Otra propuesta para la serie de Fibonacci

```
1  def fib2(n):
2    resultado = []
3      a, b = 0, 1
4      while a < n:
5          resultado.append(a)
6          a, b = b, a + b
7      return resultado
8
9  f100 = fib2(100)
10 print(f100)
```

Paso de argumentos

Para que una función sea en verdad útil (y reutilizable), es necesario que podamos pasarle entradas.

Los nombres de las entradas (o argumentos) que requiere una función se declaran a continuación del nombre en `def` (siempre entre paréntesis)

Paso de argumentos

Código 5: Paso de argumentos en una función

```
1 def FuncionSuma(x, y):
2     return x + y
3
4 print FuncionSuma(5, 3)
5 print FuncionSuma(7, 42.0)
6 print FuncionSuma(" hola ", " mundo ")
```

Paso de argumentos

Nota:

- Nunca se mencionan los tipos de datos de x e y, ni el tipo de datos que devuelve `FuncionSuma`.

Paso de argumentos

Nota:

- Nunca se mencionan los tipos de datos de x e y, ni el tipo de datos que devuelve `FuncionSuma`.
- Los argumentos y el valor devuelto son, tal como las variables, simples etiquetas a zonas de memoria.

Paso de argumentos con nombre

Si la función que definimos tiene muchos argumentos, es fácil olvidar el orden en que fueron declarados.

Como un argumento no lleva asociado un tipo, `python` no tiene manera de saber que los argumentos están cambiados.

Paso de argumentos con nombre

Para evitar este tipo de errores, hay una manera de llamar a una función pasando los argumentos en cualquier orden arbitrario: los argumentos se pasan usando el nombre usado en la declaración.

Ejemplo de paso de argumentos con nombre

Código 6: Paso de argumentos con nombre

```
1  def Prueba(a, b, c):
2  #  %r formatea automaticamente cualquier valor
3      print("a = %r, b = %r, c = %r" %(a, b, c))
4
5  Prueba(1, 2, 3)
6  Prueba(b=3, a=2, c=1)
7
8  #Salida en la terminal
9  a = 1, b = 2, c = 3
10 a = 2, b = 3, c = 1
```

Argumentos con valores por omisión

Para hacer que algunos argumentos sean opcionales, se les asignan valores por omisión al declararlos:

Código 7: Ejemplo cuando se omiten argumentos declarados

```
1 from math import sqrt
2 # argumento v es requerido , c es opcional
3 # c toma el valor 3.0e8 por omision
4 def Gamma(v, c = 3.0e+8):
5     return sqrt(1.0 -(v/c)**2)
6
7 print(Gamma(0.1 , 1.0))
8 print(Gamma(1.e+7)) # usa c = 3.0e+8
```

Regresando varios valores en una función

Para hacer que una función devuelva más de un valor, en lenguajes como Fortran, C o C++, lo que se hace es definir argumentos de entrada y argumentos de salida.

Regresando varios valores en una función

Para devolver múltiples valores en `python`, lo usual es devolver los valores “empaquetados” en una tupla:

Regresando varios valores en una función

Código 8: Devolviendo varios valores

```
1  from math import atan , sqrt
2
3  def ModuloArgumento(x, y):
4      norm = sqrt(x**2 + y**2)
5      arg = atan2(y, x)
6      return (norm, arg)
7
8  n, a = ModuloArgumento(3.0,4.0)
9  print(" El modulo es:", n)
10 print(" El argumento es:", a)
```

Número variable de argumentos

¿Cómo le hacemos para que una función acepte un número no prefijado de argumentos?

Es posible pasar una lista o tupla, pero `python` ofrece una mejor solución:

Número variable de argumentos

Código 9: Uso de *args

```
1  import math
2
3  def miatan(*args ):
4  # args es una tupla de argumentos
5      if len(args) == 1:
6          return math.atan(args[0])
7      else :
8          return math.atan2(args[0],args[1])
9
10 print(miatan (0.2)) # 0.19739
11 print(miatan (2.0,10.0)) # 0.19739
```

Funciones lambda

`python` admite una interesante sintaxis que permite definir funciones mínimas, de una línea, sobre la marcha.

Se trata de las denominadas funciones **lambda**, que pueden utilizarse en cualquier lugar donde se necesite una función.

Funciones lambda I

Código 10: Ejemplo de función lambda

```
1  def f(x):
2    return x*2
3
4  print(f(3))
5  #Devuelve 6
6
7  g = lambda x: x*2
8  print(g(3))
9  #Devuelve 6
10
```

Funciones lambda II

```
11 print((lambda x: x*2)(3))
12 #se usa lambda dentro del print
```

Funciones lambda

La primera función **lambda** consigue el mismo efecto que la función **f(x)**.

Nótese que: la lista de argumentos no está entre paréntesis, y falta la palabra reservada **return** (está implícita, ya que la función entera debe ser una única expresión).

Igualmente, la función no tiene nombre, pero puede ser llamada mediante la variable a que se ha asignado.

Módulos

Es una buena práctica almacenar las funciones en módulos.

Un módulo es un archivo en donde se dejan las funciones, el nombre del módulo es el nombre del archivo.

Uso de los módulos

Un módulo se carga al programa con la instrucción

```
from nombre_modulo import *
```

`python` incluye un número grande de módulos que contienen funciones y métodos para varias tareas.

Uso de los módulos

La gran ventaja de los módulos es que están disponibles en internet y se pueden descargar, dependiendo de la tarea que se requiera atender.

Usando el módulo `math`

Muchas funciones matemáticas no se pueden llamar directo del intérprete, pero para ello existe el módulo **`math`**.

Hay tres diferentes maneras en las que se puede llamar y utilizar las funciones de un módulo.

```
from math import *
```

Usando el módulo `math`

De esta manera, se importan todas las funciones definidas en el módulo **`math`**, siendo quizá un gasto innecesario de recursos, pero también generar conflictos con definiciones cargadas de otros módulos.

```
from math import func1, func2,...
```

```
from math import func1, func2,...
```

```
>>> from math import log,sin
>>> print (log(sin(0.5)))
-0.735166686385
```

Usando el módulo `math`

El tercer método que es el más usado en programación, es tener disponible el módulo:

```
import math
```

Las funciones en el módulo se pueden usar con el nombre del módulo como prefijo:

```
>>> import math
>>> print (math.log(math.sin(0.5)))
-0.735166686385
```

Contenido del módulo `math`

Podemos ver el contenido de un módulo con la instrucción:

```
>>> import math
>>> dir(math)
```

```
['__doc__', '__name__', '__package__', 'acos',
'acosh',  'asin', 'asinh', 'atan', 'atan2',
'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'gamma', 'hypot', 'isinf', 'isnan',
'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh', 'trunc']
```