Python
Programación Modular
Versión Preliminar

Carlos E. González C.*

21 de julio de 2019

1. Introducción

2. Programación Modular

La programación modular es un paradigma cuyo énfasis está en la creación de programas divididos en **módulos de funcionalidad** (ó subprogramas), que por definición deben ser intercambiables e independientes.

La modularización facilita la construcción de programas en general y reduce considerablemente la dificultad detrás del diseño de programas extensos. Además, surge como consecuencia directa el concepto de **reutilización de código**, que de forma poco rigurosa, se puede describir como la acción de utilizar un módulo de funcionalidad en diversos programas.

En términos generales, un subprograma es indistinguible de un programa, ya que ambos aceptan datos, realizan cálculos y generan un resultado (directa o indirectamente), el prefijo sub solo sugiere que dicho componente de software es usado como bloque constructor de otro programa.

El método de diseñar la solución de un problema separandolo en subproblemas se conoce como **diseño desendente** (*top-dow design*) [1]. Tradicionalmente los subprogramas se suelen clasificar como procedimientos o funciones.

*Dpto. Investigación de Operaciones y Computación, Facultad de Ingeniería, Universidad Central de Venezuela

2.1. Funciones y Procedimientos

Los subprogramas una vez implementados suelen clasificarse como funciones o como procedimientos, en ambos casos dichos extractos de código reciben *datos de entrada*, que en el contexto de la programación modular suelen ser llamados **argumentos**.

2.1.1. Funciones

En lineas generales, una **función** es la implementación de un algoritmo mediante un lenguaje de programación. Históricamente, los subprogramas que generan un **único resultado** son llamados funciones. Las funciones toman un número finito de argumentos.

2.1.2. Procedimientos

Al igual que las funciones, los **procedimientos** reciben un número finito de argumentos, la diferencia fundamental radica en que los procedimientos pueden arrojar **cero, uno o múltiples resultados**. Por lo tanto, los procedimientos podrían pensarse como una generalización de las funciones, aunque dependiendo del autor existen más o menos diferencias entre estos conceptos.

En algunos lenguajes (e.g., Pascal) existen estructuras bien diferenciadas y palabras claves asociadas a los conceptos de funciones y procedimientos; en contraste, existen lenguajes que no hacen referencia directa dentro de sus palabras claves a ninguno de los dos términos (e.g., Python).

3. Funciones en Python

A pesar de la terminología tradicional mostrada en la sección 2.1 donde se diferencian los conceptos de funciones y procedimientos, la documentación de Python 3 solo hace referencia a funciones, como sinónimo de una implementación de un subprograma.

3.1. Definición de funciones en Python

Los subprogramas en general deben ser definidos, es necesario indicar al interprete explicitamente que se desea definir una función. A continuación se muestra un ejemplo de la implementación de una función:

```
#Definicion, funcion que retorna
#un termino de la serie de fibonacci
#inmediatamente menor a n.
def fib(n):
        """Return a Fibonacci's term immediately lower
        than n."""
        a,b = 0,1
        while b < n:
                a,b = b, a+b
        return a
```

En este código, se muestra la definición de una función con identificador `fib(n)`, la cual genera el término de la serie de fibonacci inmediatamente menor a `n`.

La palabra clave **`def`** introduce la *definición* de la función. Seguidamente debe estar el identificador; luego entre paréntesis, la lista de los parámetros de entrada y posteriormente el símbolo "`:`" para indicar que sigue el cuerpo de la función. Las sentencias que forman el cuerpo empiezan en la línea siguiente y deben estar indentadas [2].

La primera sentencia del cuerpo de la función puede ser opcionalmente una cadena de texto; esta cadena es el texto de documentación de la función, también llamado **docstring**. Existen diversas herramientas que usan el docstring para producir documentación automática online. Es por esto que se considera una buena práctica incluirlo en la definición de las funciones [2].

Finalmente, la sentencia **`return`** se encarga de regresar el valor desde la función. En el ejemplo anterior el contenido de la variable `a` será el resultado de la función `fib`.

3.2. Llamada de una función

El término **llamada** (ó invocación) de una función se refiere a la utilización del subprograma dentro de una línea de interpretación. Tomemos como ejemplo el código mostrado a continuación:

```
#Definicion, funcion que retorna la suma
#de dos argumentos, a y b.
def suma(a,b):
        """Return a the sum of two elements a,b."""
        c = a + b
        return c
#Ahora se llama a la funcion que se acaba de definir:
var_1 = suma(20,5)
print(var_1)
print(suma(15,var_1))
```

```
[1]: 25
[2]: 40
```

En este ejemplo, se define la función `suma(a,b)` con dos argumentos `a,b` y retorna la suma de estos. Seguido a la definición, se realizan dos llamadas de dicha función.

En la primera invocación `var_1 = suma(20,5)`, el interprete ejecuta los siguientes pasos:

1. El interprete establece una correspondencia entre los argumentos genéricos de la función `a,b` y los datos `20,5`, introducidos por el usuario en su llamada[1]. Una vez establecida dicha correspondencia, el identificador `a` toma localmente[2] el valor de 20 y el identificador `b` toma localmente el valor de 5.

2. El interprete ejecuta el cuerpo de la función. En este caso formado únicamente por el comando `c = a + b`. Como consecuencia, se le asigna localmente el valor de 25 a la variable `c`.

3. Finalmente, cuando el interprete encuentra la palabra clave **`return`** detiene la ejecución del cuerpo del subprograma y sustituye el valor de la variable `c` donde se realizó la llamada de la función. Concretamente, `suma(20,5)` equivale al valor `25`.

Al finalizar la llamada `var_1 = suma(20,5)`, queda asignado el valor `25` a la varaible `var_1`. En el caso de la segunda llamada, **`print`**`(suma(15,var_1))` el proceso es similar al descrito anteriormente, con la diferencia que a las variable `a,b` se les asignará localmente los valores `15` y `25` respectivamente.

[1]El orden como se introducen dichos argumentos resulta fundamental, tema que se estudiará detalladamente en al sección 3.4.

[2]En la sección 3.3 se ampliará a que hace referencia el término *localmente*.

3.3. Ámbito de las variables

Las variables utilizadas en los programas principales y subprogramas se clasifican en dos tipos [1]

1. Variables locales.
2. Variables Globales.

Las **variables locales** son aquellas que están definidas dentro de un subprograma, lo que implica que son distintas a otras variables con el mismo identificador declaradas en cualquier otra parte del código, ya sea en el programa principal (el que llama a la función) o en otro subprograma. Todas las variables definidas dentro del cuerpo de una función o en su argumento, son por definición variables locales y su **ámbito** es el cuerpo de la función donde fueron definidas [1].

Las **variables globales** son aquellas que pueden ser modificadas en cualquier región del código y almacenan un valor inequívoco en cualquier ámbito del código. Una variable global puede ser modificada desde cualquier lugar de un programa, incluso fuera de su ámbito, por ejemplo, en el interior de una función. Para que el interprete trate a dicha variable como global dentro de un ámbito diferente al de la variable, se utiliza la palabra clave **global**. A continuación se muestra un ejemplo :

```
def change():
        """Global variable test."""
        global x #Indica que x es global
        x = 100 #Asigna el valor 100 a x

def nchange():
        """Local variable test."""
        x = 250 #Asigna el valor 250 a x
#Se llama a las funciones change() y nchange()
#dentro del programa principal
x = 20
print(x)
change()
print(x)
nchange()
print(x)
```

```
[1]: 20
[2]: 100
[3]: 100
```

En el ejemplo anterior se definen dos funciones `change()` y `nchange()` (procedimientos, estrictamente hablando), en ninguno de los dos casos se esperan argumentos de entrada ni salidas. En el cuerpo de la función `change()`, se asigna a la variable `x` el valor de 100. En el caso de `nchange()` se asigna a la variable x el valor de 250.

En el programa principal del último ejemplo, el intérprete ejecuta los siguientes pasos:

1. `x = 20` : El intérprete asigna a la variable `x` el valor de `20`.
2. `print(x)`: El intérprete imprime el valor de `x`, en la consola se muestra el valor `20`.
3. `change()`: El intérprete realiza el llamado a la función `change`, la cual en su cuerpo indica que la variable `x` debe ser tratada como global, (`global x`) asigna el valor de `x = 100` y vuelve a la línea de interpretación del programa principal.
4. `print(x)`: El intérprete imprime el valor de `x`, en la consola se muestra el valor `100`.
5. `nchange()`: El intérprete realiza el llamado a la función `nchange`, la cual en su cuerpo asigna el valor de `x = 250` y vuelve a la línea de interpretación del programa principal.
6. `print(x)`: Finalmente, el intérprete imprime el valor de `x`, en la consola se muestra el valor `100` ya que la función `nchange()` modificó localmente la variable `x`, modificación que no tiene validez fuera del cuerpo de la función.

Como se muestra en los pasos anteriores, una variable puede ser modificada en un ámbito diferente al propio, siempre y cuando se indique explícitamente al intérprete que dicha variable debe ser considerada global.

3.4. Argumentos de una función

Hasta ahora se han utilizado una cantidad predefinida de argumentos **posicionales**. A continuación se mostraran algunos mecanismos que ofrece Python para pasar argumentos a una función.

3.4.1. Argumentos por defecto.

Bajo ciertas condiciones es deseable darle valores por defecto a los argumentos de las funciones. A continuación se muestra un ejemplo:

```
#Definicion, funcion que retorna el valor
# x elevado a la raiz n-esima de x
def root(x,n = 2):
        """Calculate the n-th root of x"""
        c = x ** (1/n)
        return c

#Se llama a las funciones implementadas
numero = 16
r_cuadrada = root(numero)
print(raiz_cuadrada)
r_cuarta = root(numero, n = 4)
print(raiz_cuarta)
```

```
[1]: 4.0
[2]: 2.0
```

Los argumentos por defecto se inicializan en la definición de la función, se les da un valor el cual tomarán en el cuerpo del subprograma en caso de que no se indique un valor diferente. Para modificar el valor por defecto de un argumento, es necesario conocer el identificador del mismo e igualarlo al valor deseado al momento de llamar la función. En este caso particular, el valor de argumento `n` se modifica con la instrucción `r_cuarta = root(numero, n = 4)`).

Como regla general, los argumentos por defecto siempre deben definirse después de los argumentos posicionales.

3.4.2. Cantidad arbitraria de argumentos

Existe la posibilidad de que no se conozca a priori la cantidad de argumentos que debe recibir una función, en estos casos se utiliza el argumento genérico `*args`, el cual recoge en una tupla con identificador `args` todos los argumentos pasados por el usuario. A continuación se muestra un ejemplo que ilustra el funcionamiento de dicho concepto:

```
#Definicion, funcion que retorna el promedio
#de una cantidad indeterminada de argumentos
#dados por el usuario
def mean(*args):
        """Calculate the mean of a data"""
        p = sum(args)/len(args)
        return p

#Se llama a la funcion implementada
print(mean(5,5))
print(mean(3,4,5,6,7,7))
```

```
[1]: 5.0
[2]: 5.333333333333
```

El identificador `args` no es una palabra reservada del sistema ni obligatoria, por lo tanto puede sustituirse por cualquier identificador pero es considerada una buena práctica utilizar `args`. Los argumentos posicionales, los argumentos por defecto y los argumentos de tamaño genérico `*args` pueden ser combinados en una función, siempre y cuando se definan en el siguiente orden:

1. Argumentos posicionales.
2. Argumentos enviados por una tupla dinámica `*args`.
3. Argumentos keyword.

En la sección 3.5 se muestra un ejemplo de una función que combina todas estas formas de pasar argumentos.

3.5. Salida de funciones

Hasta el momento se han considerado subrutinas que arrojan una variable de salida. La funciones de Python permiten devolver más de una variable en la salida. A continuación se muestra un ejemplo de una función que recibe múltiples argumentos (con los mecanismos mostrados en la sección 3.4) y devuelve una tupla de variables en la salida del programa :

```
#Definicion, funcion que recibe argumentos posicionales
#por una tupla dinamica y de tipo keyword.
def genericf(x,*args,n = 2):
        """Generic function to demonstrate the
        performance with multiple arguments."""
        c = x ** (1/n)
        lista_1 = list(args)

        return c, lista_1

#Zona programa principal
print(genericf(16))
a,b = genericf(16)
print("a: {}\t b: {}".format(a,b))
a,b = genericf(16, n = 4)
print("a: {}\t b: {}".format(a,b))
a,b = genericf(16,20,5,6,15,4)
print("a: {}\t b: {}".format(a,b))
a,b = genericf(16,20,5,6,15,n=4)
print("a: {}\t b: {}".format(a,b))
```

```
[1]: (4.0, [])
[2]: a: 4.0             b: []
[3]: a: 2.0             b: []
[4]: a: 4.0             b: [20,5,6,15,4]
[5]: a: 2.0             b: [20,5,6,15]
```

4. Módulos en Python

Un **módulo** es un archivo que contiene la definición de clases, funciones y variables. Los módulos surgen por la necesidad de reutilizar código, generalmente se necesita realizar llamados a funciones útiles en diferentes programas, por esta razón dichas funciones son empaquetadas dentro de un módulo. Las definiciones almacenadas en un módulo pueden ser importadas en otros módulos o en un script [2].

4.1. Creación de módulos en Python

El nombre de un módulo está definido por el nombre del archivo donde se almacena. Dentro de dicho archivo, el nombre es almacenado en una variable global con el identificador `__name__`. A continuación se muestra un ejemplo de un módulo, fibo.py [2]:

```
# Fibonacci numbers module
def fib(n):
        """write Fibonacci series up to n"""
        a, b = 0, 1
        while a < n:
                print(a, end=' ')
                a, b = b, a+b
        print()
def fib2(n):
        """return Fibonacci series up to n"""
        result = []
        a, b = 0, 1
        while a < n:
                result.append(a)
                a, b = b, a+b
        return result
```

4.2. Llamada o importación de módulos en Python

Es posible entrar al intérprete (o en otro archivo en el mismo directorio) e importar este módulo con el siguiente comando.

```
>>>import fibo
```

Este método de importación no introduce directamente los nombres de las funciones definidas en el módulo fibo.py en la tabla de símbolos desde donde fue importado; esto solo introduce el nombre del módulo fibo. Por lo tanto, si intentamos llamar a la función fib2(), dicha función no será encontrada. Lo anterior es resuelto utilizando el nombre del módulo para aclarar al interprete donde estamos buscando dicha función. A continuación se muestra un ejemplo [2]:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

En pocas palabras, si un módulo se importa de la forma **import** modulename, entonces para acceder a sus definiciones (llamar funciones o clases), es necesario escribir el nombre del módulo seguido de un punto y el identificador de la definición deseada (e.g., modulename.fname()).

Existen algunas variantes de la sentencia **import**. Una forma de importar directamente los nombres de las definiciones se muestra a continuación [2]:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

En este caso se importan directamente fib y fib2 desde el módulo fibo.py. Por lo tanto, si realiza una llamada a alguna de estas funciones, no es necesario indicar explícitamente el nombre del módulo para dicha llamada. Es posible además importar *todos* los nombres de las definiciones de un módulo mediante la instrucción **from** modulename **import** $*$. A continuación se muestra un ejemplo:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Como se observa en el código anterior, si se sustituyen los nombres de las definiciones por el caracter $*$, entonces se importa todo el contenido del módulo. De esta manera, es posible realizar llamadas a *cualquier* definición del módulo sin indicar explícitamente el nombre del mismo.

Finalmente, es posible importar módulos o definiciones internas de módulos con nombres personalizados, la validez de estos nombres (alternativos) solo se limita al entorno donde realizó la importación. A continuación se muestran dos ejemplos:

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

En el primer ejemplo, es *necesario* referirse al módulo fibo con el identificador `fib`. Equivalentemente, en el segundo ejemplo la función `fib`, debe ser referida con el identificador `fibonacci`.

5. Paquetes

Los **paquetes** son definidos como una colección de módulos, una vía para almacenar y desplegar módulos (con fines similares). Tal vez, el ejemplo más representativo sea el paquete NumPy. Como regla general los paquetes utilizan la técnica **dotted module names**, con la cual es posible acceder a módulos dentro de dicho paquete. A continuación un ejemplo:

```
>>>import numpy
>>>numpy.random.random()
```

En el ejemplo anterior es importado el paquete NumPy con la instrucción `import numpy`. La instrucción `numpy.random.random()` realiza una llamada a la función `random()` dentro del módulo random.py, que a su vez se encuentra en el paquete NumPy. La creación de paquetes no será abordada en este documento ya que sale del alcance del mismo.

Referencias

[1] Luis Joyanes. *Fundamentos de Programación*. McGraw-Hill, 2008.

[2] Guido Van Rossum. *Python Tutorial*. Python Software Fundation, 2019.