FUNCIONES

Una función es una forma de agrupar expresiones y algoritmos de forma tal que estos queden contenidos dentro de «una cápsula», que solo pueda ejecutarse cuando el programador la invoque. Una vez que una función es definida, puede ser invocada tantas veces como sea necesario.

Funciones incorporadas

Una función puede ser provista por el lenguaje o definida por el usuario. A las funciones provistas por el lenguaje se las denomina «funciones incorporadas» y en inglés se conocen como «*Built-in functions*».

FUNCIONES DEFINIDAS.POR.EL USUARIO

En Python la definición de funciones se realiza mediante la instrucción **`def`** más un nombre de función descriptivo -para el cuál, aplican las mismas reglas que para el nombre de las variables- seguido de paréntesis de apertura y cierre.

Como toda estructura de control en Python, la definición de la función finaliza con dos puntos (:) y el algoritmo que la compone, irá identado con 4 espacios en blanco:

```
def mi_funcion():
    # aquí el algoritmo identado
```

Una función no es ejecutada hasta tanto no sea invocada. Para invocar una función, simplemente se la llama por su nombre:

```
def mi_funcion():
    print (“Hola Mundo”)

funcion()
```

Las funciones pueden **retornar datos**:

```
def funcion():
    return "Hola Mundo"
```

Y el valor de retorno de una función puede **almacenarse** en una variable:

```
frase = funcion()
```

Imprimirse:

```
print (funcion())
```

Ignorarse:

```
funcion()
```

Sobre los Paramétros

Un parámetro es un valor que la función espera recibir cuando sea llamada (invocada). Una función puede esperar uno o más parámetros (que son definidos entre los paréntesis y separados por una coma) o ninguno.

```
def mi_funcion(param1, param2):
     pass
```

> La instrucción **pass** se utiliza para completar una estructura de control que no realiza ninguna acción.

Los parámetros que una función espera, serán utilizados por ésta, dentro de su algoritmo, a modo de **variables de ámbito local**. Es decir, que los parámetros serán variables locales que solo son accesibles dentro de la función:

```
def calcular_neto(bruto, alicuota):
    iva = bruto * float(alicuota) / 100
    neto = bruto + iva
    return neto
```

Si se quisiera acceder a esas variables locales fuera del ámbito de la función, se obtendría un error:

```
def calcular_neto(bruto, alicuota):
    iva = bruto * float(alicuota) / 100
    neto = bruto + iva
    return neto

print (bruto) # Retornará el error: NameError: name 'bruto' is not defined
```

Al llamar a una función, **se le deben pasar sus argumentos en el mismo orden en el que los espera**. Para evitar pasarlos en el mismo orden, pueden utilizarse claves como argumentos (definidos más abajo).

Parametros por omisión

Es posible asignar valores por defecto a los parámetros de las funciones. Esto significa, que la función podrá ser llamada con menos argumentos de los que ha definido:

```
def calcular_neto(bruto, alicuota=21):
    iva = bruto * float(alicuota) / 100
    neto = bruto + iva
    return neto

calcular_neto(100)        # Retorna 121.0
calcular_neto(100, 10.5)  # Retorna 110.5
```

PEP 8: Funciones

A la definición de una función la deben anteceder dos líneas en blanco.

Al asignar parámetros por omisión, no debe dejarse espacios en blanco ni antes ni después del signo =.

Los parámetros por omisión deben ser definidos a continuación de los parámetros obligatorios.

CLAVES COMO ARGUMENTOS

Las claves como argumentos *(keword arguments)* son una característica de Python que no todos los lenguajes poseen. Python permite llamar a una función pasándole los argumentos esperados como pares de `claves=valor`:

```
def funcion(obligatorio1, opcional='valor opcional', opcional_dos=15):
    pass

funcion('valor obligatorio', opcional_dos=43)
```

PARÁMETROS ARBITRARIOS

Es posible que una función espere recibir un número arbitrario -desconocido- de argumentos. Estos argumentos, llegarán a la función en forma de tupla.

Para definir argumentos arbitrarios en una función, se antecede un asterisco (*) al nombre del parámetro :

```
def funcion(obligatorio, *arbitrarios):
    pass


funcion('fijo', 1, 2, 3, 4, 5)
funcion('fijo', 1, 2)
funcion('fijo', 1, 2, 3, 4, 5, 6, 7)
funcion('fijo')
```

Cuando una función espera recibir parámetros obligatorios y arbitrarios, **los arbitrarios siempre deben suceder a los obligatorios**.

Es posible también, obtener parámetros arbitrarios como pares de clave=valor. En estos casos, al nombre del parámetro deben precederlo dos asteriscos (**):

```
def funcion(obligatorio, **arbitrarios):
    pass


funcion('fijo', a=1, b=2, c=3)
```

El **recorrido de los parámetros arbitrarios** es como el recorrido de cualquier tupla (para parámetros arbitrarios sin clave) o de cualquier diccionario (para los parámetros arbitrarios con clave):

```
def funcion(*arbitrarios):
    for argumento in arbitrarios:
        pass


def funcion(**arbitrarios):
    for argumento in arbitrarios:
        valor = arbitrarios[argumento]
```

DESEMPAQUETADO DE PARÁMETROS

Al invocar a una función se le pueden pasar los parámetros que espera dentro de una lista o de un diccionario (donde los nombres de las claves equivalen al nombre de los argumentos). A este procedimiento se lo conoce como desempaquetado de parámetros:

```
def funcion(uno, dos, tres):
    pass


# DESEMPAQUETADO DE LISTAS
parametros = [1, 2, 3]
calcular(*parametros)

# DESEMPAQUETADO DE DICCIONARIOS
```

```
parametros = dict(uno=1, dos=2, tres=3)
calcular(**parametros)

parametros = {'uno': 1, 'dos': 2, 'tres': 3}
calcular(**parametros)
```

LLAMADAS RECURSIVAS Y DE RETORNO

Una función puede llamar a otra función que retorne un valor. Esto se conoce como llamada de retorno:

```
def retornar(algo):
    return str(algo)


def llamar():
    algo = retornar()
```

La llamada interna a otra función puede almacenarse, retornarse o ignorarse.

```
def almacenar():
    algo = retornar()


def volver_a_retornar():
    return retornar()


def ignorar():
    retornar()
```

Cuando la llama que se hace es a la misma función que se llama, se conoce como **llamada recursiva**.

```
def get_nombre():
    nombre = raw_input("Nombre: ")
    if not nombre:
        get_nombre()
```

SOBRE LA FINALIDAD DE LAS FUNCIONES

Una función puede tener cualquier tipo de algoritmo y cualquier cantidad de instrucciones. Sin embargo, **una buena práctica** indica que la finalidad de una función, debe ser **realizar una única acción**.