Facultad de Ciencias de la Administración

TECNICATURA UNIVERSITARIA EN DESARROLLO WEB

PROGRAMACIÓN I

Unidad IV – Funciones y módulos

Funciones. Argumentos. Parámetros y variables

Tecnicatura Universitaria en Desarrollo Web

Facultad de Ciencias de la Administración

Universidad Nacional de Entre Ríos

- **Objetivos**
  - Entender cómo definir funciones.
  - Comprender las distintas técnicas para separar las responsabilidades de una aplicación.
  - Conocer la importancia de la reutilización de código.
  - Definir funciones recursivas.
- **Temas a desarrollar:**
  - Modularización. Definición. Funciones. Definición.
  - Parámetros y argumentos. Técnicas de diseño top-down y bottom-up.
  - Módulos. Concepto. Definición. Reutilización. Concepto.
  - Recursividad. Definición.

Flujo de ejecución

- Para asegurar que una **función** se **defina antes de usarse por primera vez**, tenemos que conocer el **orden de las sentencias que contiene**, lo que se conoce con el nombre de ***flujo de ejecución***.
- La **ejecución** siempre comienza por la primera sentencia del programa. Las sentencias son ejecutadas una a la vez **de arriba hacia abajo**.
- Las definiciones de funciones **no alteran** el flujo de ejecución del programa.
- Una **llamada a una función** es como un **desvío** en el **flujo de ejecución**. En vez de ir hacia la siguiente sentencia, el flujo **salta** al cuerpo de la función, ejecuta las sentencias allí y luego vuelve para retomar donde se había dejado.
- Mientras se encuentra en medio de una función, es posible que el programa deba ejecutar las sentencias de otra función. Luego, mientras ejecuta esa nueva función, ¡el programa podría tener que ejecutar otra función más!
- **Python** se encarga por nosotros de llevar nota de donde es necesario retomar la ejecución luego de una función es invocada y cuando se llega al fin del programa, lo termina.

Parámetros y argumentos

- Algunas funciones que vimos requieren **argumentos**. Por ejemplo, cuando llamamos a la función **`math.sqrt`** pasamos un número como **argumento**.
- Algunas funciones requieren más de un **argumento**. Por ejemplo, **`math.pow`**, necesita la base y el exponente.
- Dentro de la función, los **argumento** son asignados a variables que se llaman **parámetros**. Aquí la definición de una función que acepta **argumentos**:

```
def imprimir_dos_veces(nombre):
    print(nombre)
    print(nombre)
```

- La función asigna el **argumento** a un **parámetro** que se llama **nombre**. Cuando la función es invocada, imprime el valor del parámetro dos veces. Funciona con cualquier valor que pueda imprimirse por pantalla.

```
» imprimir_dos_veces('Martín')
    Martín
    Martín
» imprimir_dos_veces(42)
    42
    42
```

Parámetros y argumentos (2)

```
» imprimir_dos_veces(math.pi)
3.14159265359
3.14159265359
```

- También podemos hacer **composición** utilizando funciones integradas:

```
» imprimir_dos_veces('Nacho' * 4)
Nacho Nacho Nacho Nacho
Nacho Nacho Nacho Nacho
» imprimir_dos_veces(math.cos(math.pi))
-1
-1
```

Parámetros y argumentos (3)

- El **argumento** es evaluado antes que la llamada a la función, así que los ejemplos de las expresiones. `'Nacho' * 4` y `math.cos(math.pi)` se evalúan una sola vez.
- También podemos usar una variable como argumento:
  - `» larita = 'Lari, lari!!!'`
  - `» imprimir_dos_veces(larita)`
- El nombre de la variable que pasamos como argumento (`larita`) no tiene nada que ver con el nombre del parámetro nombre. No importa qué nombre tenga el argumento en desde donde se hizo la invocación.

Alcance de variables y parámetros

- Cuando creamos una **variable** dentro de una función su alcance es **local**, esto quiere decir que solo existe dentro de la función. Por ejemplo:

```
def concat_imprimir(part1, part2):
    cat = part1 + part2
    imprimir_dos_veces(cat)
```

- Esta función toma dos argumentos, los concatena, e imprime los resultados dos veces. Ejemplo:

```
» linea1 = 'Línea 1'
» linea2 = 'Línea 2'
» concat_imprimir(linea1, linea2)
```

- Cuando **concat_imprimir** termina, la variable **cat** es destruida. Si intentamos imprimir su valor obtendremos una excepción:
- » **print(cat)** → `NameError: name 'cat' is not defined`
- Los parámetros son locales a la función. Por ejemplo, fuera de **imprimir_dos_veces**, no hay nada como **cat**.

Funciones que retornan valores

- Algunas **funciones** que hemos usado, tales como las funciones matemáticas, ***retornan valores*** (en algunos contextos se llaman ***funciones que retornan valores*** o **fructíferas**).
- En funciones, como **`imprimir_dos_veces`**, se lleva a cabo una acción pero ***NO se retorna un valor***.
- Estas funciones son llamadas **funciones que no retornan valores**.
- Cuando llamamos a una función que retorna valores, casi siempre tenemos que hacer algo con el resultado. Por ejemplo, puede ser que queramos asignarlo a una variable o utilizarlo como parte de una expresión.

```
» calculo = (math.sqrt(25) + 1) / 2
```

- Cuando llamamos a cualquier función en modo interactivo, **Python** muestra un resultado.
- Pero en un script si **llamamos a una función** que **devuelve resultados** el **valor de retorno se pierde**.
- Las funciones que retornan vacío puede ser que muestren algo en la pantalla o tengan algún efecto pero no tienen valor de retorno. Si asignamos el resultado a una variable, obtendremos un valor especial que es **`None`**.
  - **`None`** no es lo mismo que **`'None'`**. **`None`** es un valor que tiene su tipo especial
  - **`» type(None) → <class 'NoneType'>`**

Funciones que retornan valores (2)

- Cuando invocamos a una ***función que retorna valores*** generalmente utilizamos el **resultado** para asignárselo a una **variable** o como parte de una **expresión**.

```
e = math.exp(1.0)

height = radius * math.sin(radians)
```

- A continuación vamos a programar la función **`area()`** que devuelve el área de un círculo dado su radio:

```
def area(radio):
    calculo = math.pi * radio**2
    return calculo
```

La **semántica** de la sentencia **`return`** es: ***"Devolver inmediatamente a quien invocó la expresión que sigue"***.

- La expresión puede ser tan complicada como queramos. Por ejemplo, podemos omitir la variable **`calculo`** del ejemplo anterior.

```
def area(radio):
    return math.pi * radio**2
```

¿Por qué funciones?

- Hay varias razones por las cuales vale la pena dividir un programa en **funciones**:
  - Crear una nueva función nos da la oportunidad de nombrar un grupo de sentencias lo que hace ***más fácil de leer*** y ***depurar nuestros programas***.
  - Una función hace el programa más pequeño eliminado código repetitivo. Más tarde si tenemos que hacer un cambio ***solo tenemos que hacerlo en un único lugar***.
  - Dividir un programa largo en funciones nos permite depurar las partes que lo componen una a la vez y luego ensamblarlo en un todo que funcione.
  - Las funciones bien diseñadas, generalmente son utilizadas por muchos programas. Una vez que escribimos y depuramos podemos reutilizar.

Bibliografía

- Óscar Ramírez Jiménez: ***"Python a fondo"*** 1era Edición. Ed. Marcombo S.L.. 2021.
- Allen Downey. ***"Think Python"***. 2Da Edición. Green Tea Press. 2015.
- Bill Lubanovic. ***"Introducing Python"***. 2Da Edición. O' Reilly. 2020.
- Eirc Matthes: ***"Python Crash Course"***. 1era Edición. Ed. No Starch Press. 2016.
- Zed A. Shaw: ***"Learn Python 3 the Hard Way"***. 1era Edición. Ed. Addison-Wesley. 2017.