Facultad de Ciencias de la Administración

TECNICATURA UNIVERSITARIA EN DESARROLLO WEB

PROGRAMACIÓN I

Unidad IV – Funciones y Módulos

Recursividad y Alcance de 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.

Recursividad (1)

- Sabemos que una función puede llamar a otra función y ésta a su vez a otra, y así sucesivamente; dicho de otro modo, las funciones se pueden anidar.
- Se puede tener:
  - **`A llamar_a B, B llamar_a C, C llamar_a D`**
- Cuando se produce el retorno de los subprogramas a la terminación de cada uno de ellos el proceso resultante será:
  - **`D retornar_a C, C retornar_a B, B retornar_a A`**

  ¿Qué sucedería si dos funciones de una secuencia son los mismos?

  - **`A llamar_a A`**
- o bien
  - **`A llamar_a B, B llamar_a A`**
- En primera instancia, parece incorrecta. Sin embargo, existen muchos lenguajes de programación donde una función puede llamarse a sí misma.

Recursividad (2)

- La **recursividad** o **recursión** es un concepto que proviene de las matemáticas, y que aplicado al mundo de la programación nos permite resolver problemas o tareas donde las mismas pueden ser divididas en subtareas cuya funcionalidad es la misma. Dado que los subproblemas a resolver son de la misma naturaleza, se puede usar la misma función para resolverlos.
- Una ***función recursiva*** es aquella que se llama repetidamente a sí misma hasta llegar a un punto de salida.
- La **recursión** puede ser utilizada como una **alternativa a la repetición o estructura repetitiva**.
- El uso de la **recursión** es particularmente idóneo para la solución de aquellos problemas que pueden definirse de modo natural en términos recursivos.
  - Es una herramienta muy potente en algunas aplicaciones, sobre todo de cálculo.
- Una **función recursiva** es similar a una tradicional solo que tiene dos secciones de código claramente divididas:
  - La sección en la que la **función se llama a sí misma**.
  - Por otro lado, tiene que existir siempre ***una condición de terminación*** en la que la función **retorna** sin volver a llamarse. Es muy importante porque **de lo contrario**, la función **se llamaría de manera indefinida**.

Cuenta regresiva

- Reveamos la solución iterativa de la cuenta regresiva para luego realizar su implementación recursiva.

Implementación Iterativa (usando while)

```
def cuenta_descendente(n):
        while n > 0:
            print (n)
            n = n - 1
        print ("It's the final countdown!!! Tarata taaaaa!!!")
```

Implementación Recursiva

```
def cuenta_descedente_rec(n):
    if n == 0:
        print("It's the final countdown! Tarata taaaa!!!")
    else:
        print(n)
        cuenta_descedente_rec(n - 1)
```

Factorial

- Muchas funciones matemáticas se definen **recursivamente**. Un ejemplo de ello es el factorial de un número entero **n**.

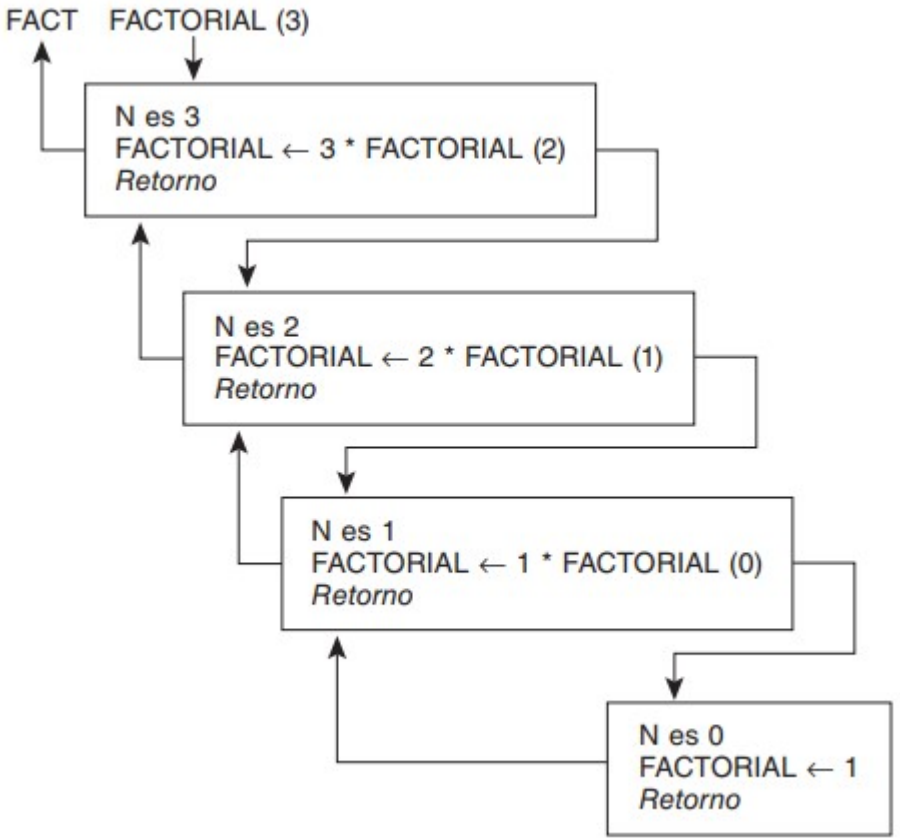
$$n! = \begin{cases} 1 \ si \quad n = 0 \quad 0! = 1 \\ nx(n-1)x(n-2)x \ \ldots \ x3x2x1 \quad si \ n > 0 \ n.\ (n-1)\ .\ (n-2)\ldots.3.2.1 \end{cases}$$

- Si se observa la fórmula anterior cuando **n > 0**, es fácil definir **n!** en función de **(n-1)!** Por ejemplo, **5:**
- **5! = 5x4x3x2x1 = 120**
- **4! = 4x3x2x1 = 24**
- **3! = 3x2x1 = 6**
- **2! = 2x1 = 2**
- **1! = 1x1 = 1**
- **0! = 1 = 1**
- En términos generales sería:

$$n! = \begin{cases} 1 & \text{si} \quad n = 0 \\ n(n-1)! & \text{si} \quad n > 0 \end{cases}$$

Factorial (2)

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



Serie Fibonacci

- Otro ejemplo típico de una función recursiva es la **serie Fibonacci**.
- Esta serie fue concebida originalmente como modelo para el crecimiento de una granja de conejos (multiplicación de conejos) por el matemático italiano del siglo XVI, Fibonacci.
- La serie es la siguiente: **`1, 1, 2, 3, 5, 8, 13, 21, 34 ...`**
- Esta serie crece muy rápidamente; como ejemplo, el término **`15`** es **`610`**.
- La serie de Fibonacci **(fib)** se expresa así:

fib(1) = 1

fib(2) = 1

fib(n) = fib(n – 1) + fib(n – 2) para n > 2

Serie Fibonacci (2)

- En **Python**, una función recursiva que calcula el elemento enésimo de la serie de Fibonacci es:

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Comprobación de tipos

- ¿Qué sucede si llamamos a factorial y le damos 1.5 como argumento?

  » **factorial (1.5)** → **RuntimeError: Maximum recursion depth exceeded**

- Tiene todo el aspecto de una recursión infinita Pero, ¿cómo ha podido ocurrir?
- Hay una condición de salida o caso base: cuando **n == 0**. Pero el valor de n nunca coincide con el caso base.
- Para solucionar esto podemos usar la función **isinstance()** para determinar si el tipo del parámetro es entero y también controlar que el parámetro sea positivo:

```
def factorial(n):
    if not isinstance(n, int):
        print('La función factorial solo se aplica a enteros.')
        return None
    elif n < 0:
        print('La función factorial solo se aplica a enteros positivos.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Espacios de nombres y alcance de variables

- Un nombre de variable puede referirse a diferentes cosas, dependiendo de dónde se use.
- Los programas de **Python** tienen varios ***espacios de nombres***. *Un* ***espacio de nombres*** *es una sección dentro de la cual un nombre en particular es único y no está relacionado con el mismo nombre en otros espacios de nombres.*
- Cada **función** define su propio **espacio de nombres**. Si define una variable con nombre **x** en un programa principal y otra variable llamada **x** en una función, **se refieren a cosas diferentes**. Sin embargo, en caso de ser necesario, esto puede superarse. Se puede acceder a variables en otros espacios de nombres de varias maneras.
- La **parte principal** de un programa define el ***espacio de nombres global***; por lo tanto, las variables en ese espacio de nombres son ***variables globales***.
- Puede obtenerse el valor de una variable global desde dentro de una función:

```
animal = 'carpincho'
def print_en_funcion():
    print('En función:', animal)
...
print('En el nivel superior:', animal) → En el nivel superior: carpincho
print_en_funcion() → En función: carpincho
```

Si intentamos cambiar el valor de animal dentro de **print_en_funcion()** vamos a tener un error.

Espacios de nombres y alcance de variables (2)

- Cambiamos un poco el ejemplo:

```
animal = 'carpincho'
def cambiar_local():
    animal = 'benteveo'
    print('En función cambiar_local():', animal, id(animal))
print('En el nivel superior:', animal, id(animal))
→ En el nivel superior: carpincho 2204778549872
cambiar_local() → En función cambiar_local(): benteveo 2204778550064
```

- ¿Que pasó aquí? La primera línea asignó la cadena `'carpincho'` a una variable **global** llamada **`animal`**.
- La función **`cambiar_local()`** también tiene una variable llamada **`animal`**, pero está en su ***espacio de nombres local***.
- Usamos la función **`id()`** para imprimir el valor único de cada objeto y probar que la variable **`animal`** dentro de **`cambiar_local()`** no es lo mismo que **`animal`** en el nivel principal del programa.

Espacios de nombres y alcance de variables (3)

- Para acceder a la variable **global** en lugar de la **local** dentro de una función, debe ser explícito y usar la palabra clave `global`:

```
animal = 'carpincho'
def cambiar_local():
    global animal
    animal = 'benteveo'
    print('En función cambiar_local():', animal, id(animal))
print('En el nivel superior:', animal, id(animal))
→ En el nivel superior: benteveo 2492781351408
cambiar_local() → En función cambiar_local(): benteveo 2492781351408
```

- Si no anteponemos global al nombre de la variable dentro de una función, **Python** usa el **espacio de nombres local** y **la variable es local**. Desaparece después de que se completa la función.

Bibliografía

- Luis Joyanes Aguilar: ***“Fundamentos de programación, algoritmos, estructura de datos y objetos"***. Ed. Mc Graw Hill. 2008.
- Allen Downey. ***“Think Python”***. 2Da Edición. Green Tea Press. 2015.
- Bill Lubanovic. ***“Introducing Python”***. 2Da Edición. O’ Reilly. 2020.