



# Programación II

Tecnicatura Universitaria en Inteligencia Artificial

2022

# Recursión

## 1. Breve repaso de funciones

Una función es un fragmento de código que permite efectuar una operación determinada. Por ejemplo:

```
>>> abs(-10)
10
>>> len([1,2,3])
3
>>> max(1,2,4,5)
5
>>> min(23,4,5,3)
3
```

- La función `abs` permite calcular el valor absoluto de un número.
- La función `len` permite obtener la longitud de una cadena de texto.
- Las funciones `min` y `max` permiten obtener el mínimo y el máximo en un conjunto de números.

Una función puede recibir cero o más **parámetros** o **argumentos** (expresados entre paréntesis, y separados por comas), efectúa una operación y pueden o no devolver un **resultado**. Por ejemplo, la función `abs` recibe un único parámetro (un número) y su resultado el el valor absoluto de dicho número.

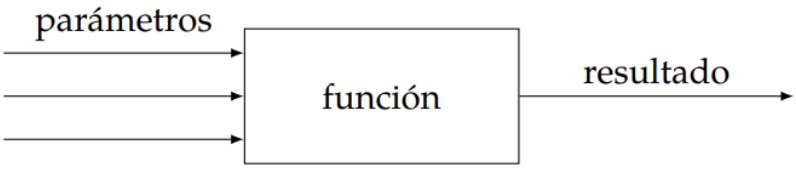



Figura 1: Una función es como una caja negra, o una máquina que, dados los parámetros correctos, produce un resultado.

Además de las funciones incluidas en Python, podemos definir nuestras propias funciones utilizando la siguiente sintaxis:

```
def nombre_funcion(argumentos):
        código
        return retorno
```

Las partes que las componen son:

1. El `nombre_función` puede ser cualquier identificador válido en Python, es decir, debe cumplir:
   - Debe contener solo letras minúsculas, mayúsculas, números y guión bajo.
   - El primer carácter no puede ser un numero.
   - Las palabras reservadas (`False`, `None`, `while`, etc) no pueden ser utilizadas como nombre de función.

   Por convención, escribiremos los nombres de las funciones siempre en minúscula y con guión bajo para separar entre palabras. Conviene utilizar nombres que denoten una acción relacionada a lo que esta ocurriendo en la función.
2. La lista de **parámetros** que recibe la función, acompañados por su tipo.
3. Si la función devuelve un valor, el tipo de reotrno se indica utilizando una flecha (`->`).
4. Luego viene el bloque de código que conforma el cuerpo de la función; debe ir indentado (por convención: 4 espacios).
5. Si la función devuelve un valor, se hace con la instrucción `return`.

Por ejemplo, consideremos la siguiente función:

```
    def hola(alguien: str) -> str:
       return "Hola " + alguien + "!"
```

La función `hola` recibe un único *parámetro* (`alguien`, de tipo `str`). Para llamar a una función debemos asociar cada uno de los parámetros con algún valor determinado (que se denomina argumento). Por ejemplo, podemos invocar a la función hola dos veces, para saludar a Ana y a Juan, haciendo que alguien se asocie al valor Ana en la primera llamada y al valor Juan en la segunda. La función en cada caso devolverá un resultado que se calcula a partir del argumento.

```
    >>> hola("Ana")
    Hola Ana!
    >>> hola("Juan")
    Hola Juan!
```

Otro ejemplo es el siguiente

```
def f(x: int) -> int:
    return 2*x
y = f(3)
print(y) # 6
```

Esta función recibe un único parámetro (`x`, de tipo `int`) y devuelve un entero, que se calcula como `2 * x`. Al invocar la función, podemos asignar el resultado de la misma a una variable.

## 1.1. Imprimir versus devolver

Supongamos que tenemos una medida de tiempo expresada en horas, minutos y segundos, y queremos calcular la cantidad total de segundos. Cuando nos disponemos a escribir una función en Python para resolver este problema nos enfrentamos con dos posibilidades:

1. Devolver el resultado con la instrucción `return`.

```
def devolver_segundos(
    horas: int, minutos: int, segundos: int
) -> int:
    """Transforma en segundos una medida de tiempo expresada en
    horas, minutos y segundos"""
    return (
        3600 * horas
        + 60 * minutos
        + segundos
    )
```

2. Imprimir el resultado llamando a la función `print`.

```
def imprimir_segundos(
    horas: int, minutos: int, segundos: int
) -> None:
    """Transforma en segundos una medida de tiempo expresada en
    horas, minutos y segundos"""
    print (
        3600 * horas
        + 60 * minutos
        + segundos
    )
```

Estas funciones pueden ser invocadas de la siguiente manera:

```
>>> imprimir_segundos(1, 10, 10)
4210
>>> devolver_segundos(1, 10, 10)
4210
```

Ambas funciones parecen comportarse igual, pero hay una gran diferencia. La función `devolver_segundos` nos permite **asignar su resultado a una variable**

```
>>> s1 = devolver_segundos(1, 10, 10)
>>> s2 = devolver_segundos(2, 32, 20)
>>> s1 + s2
13350
```

En cambio, la función `imprimir_segundos` nos impide utilizar el resultado de la llamada para hacer otras operaciones; lo único que podemos hacer es mostrarlo en pantalla

Una función es más reutilizable si devuelve un resultado (utilizando `return`) en lugar de imprimirlo (utilizando `print`). Análogamente, una función es más reutilizable si recibe parámetros en lugar de leer datos mediante la función `input`.

### 1.2. Scope de variables

El scope o alcance es una región del programa, y el scope de variables se refiere al área del programa donde las variables del programa pueden ser accedidas después de haber sido declaradas. De manera general, podemos decir que hay dos lugares donde las variables pueden ser declaradas:

- Dentro de una función o un bloque de código, las cuales se conocen como variables locales.
- Fuera de una función, las cuales se conocen como variables globales.

Una variable solo está disponible dentro de la región en la cual es creada. Las variables locales solo tienen sentido dentro de la definición de la función. Por ejemplo, en el siguiente fragmento:

```
    def gravity():
        x = 9.81
        print(x, id(x))

    def pi():
        x = 3.14159
        print(x, id(x))

    gravity()
    pi()
    print(x)
```

Al llamar a la función `gravity` se imprimirá el valor 9.81, al llamar a la función `pi` se imprimirá el valor 3.14159 y el tercer print fallara puesto que la variable `x` no esta definida como variable global. Otro punto a destacar es que, si bien en ambas funciones hay una variable llamada `x`, se trata en ambos casos de variables distintas, puesto que el scope local es privado de cada función.

**Recordatorio** La función incorporada `id()` recibe como argumento un objeto y retorna otro objeto que sirve como identificador único para el primero.

Además, dentro de una función podemos llamar a otras funciones

```
def f() -> None:
  x = 50
  a = 20
  print("En f, x vale", x)

def g() -> None:
  x = 10
  b = 45
  print("En g, antes de llamar a f, x vale", x)
  f()
  print("En g, después de llamar a f, x vale", x)
```

Notar que en este caso, si llamamos a la función `g`, esta invocará a la función `f`, y habrá dos variables llamadas `x` activas en ese momento, una perteneciente a `f` y otra perteneciente a `g`, sin embargo, ambas variables son totalmente independientes.

## 2. Recursión

Estamos acostumbrados a escribir funciones que llaman a otras funciones. Pero lo cierto es que nada impide que en Python (y en muchos otros lenguajes) **una función se llame a sí misma**. Y lo más interesante es

que esta propiedad, que se llama **recursión**, permite en muchos casos encontrar soluciones muy elegantes para determinados problemas.

**Definición recursión** La recursión es una técnica para programar computadoras que involucra el uso de procedimientos, subrutinas, funciones o algoritmos que se llaman a si mismos hasta que se alcance una condición de finalización.

**Definición función recursiva** Una función recursiva es cualquier función que se llama a si misma.

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. Dicho de otra manera, una función recursiva es aquella que está definida en función de sí misma, por lo que se llama repetidamente a sí misma hasta llegar a un punto de salida.

Cualquier función recursiva tiene dos secciones de código claramente divididas:

- Por un lado, tiene que existir siempre una condición en la que la función retorna sin volver a llamarse. Es muy importante porque de lo contrario, la función se llamaría de manera indefinida. A este se lo llama **caso base** de la recursión.

- Por un lado tenemos la sección en la que la función se llama a sí misma, con alguna variación en sus argumentos. A esto se lo llama **caso recursivo** de la recursión.

## 2.1. Ejemplo: Factorial

Es muy común tener definiciones recursivas de operaciones. Por ejemplo, el factorial[1] se define usualmente de forma recursiva: $0! = 1$
$x! = x(x-1)!$, si $x > 0$
Este tipo de definición se traduce naturalmente en una función en Python:

```
def factorial(n: int) -> int:
    """Precondición: n entero >= 0
    Devuelve: n!"""
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Esta es la ejecución del factorial para $n = 0$ y para $n = 3$.

```
>>> factorial(0)
1
>>> factorial(3)
6
```

El sentido de la instrucción `n * factorial(n - 1)` es exactamente el mismo que el de la definición inductiva: para calcular el factorial de `n` se debe multiplicar `n` por el factorial de `n - 1`.

Dos piezas fundamentales para garantizar el funcionamiento de este programa son:

- Que se defina un caso base (en este caso la indicación no recursiva de cómo calcular `factorial(0)`.

[1] La función factorial se representa con un signo de exclamación "!" detrás de un número. Esta exclamación quiere decir que hay que multiplicar todos los números enteros positivos que hay entre ese número y el 1. A este número, 6! le llamamos generalmente "6 factorial", aunque también es correcto decir "factorial de 6".

- Que el argumento de la función respete la precondición de que `n` debe ser un entero mayor o igual que 0.

### 2.2. Algoritmos recursivos y algoritmos iterativos

Llamaremos **algoritmos recursivos** a aquellos que realizan llamadas recursivas para llegar al resultado, y **algoritmos iterativos** a aquellos que llegan a un resultado a través de una iteración mediante un ciclo definido o indefinido.
**Todo algoritmo recursivo puede expresarse como iterativo y viceversa**. Sin embargo, según las condiciones del problema a resolver podrá ser preferible utilizar la solución recursiva o la iterativa.

Una posible implementación iterativa de la función factorial podría ser la siguiente:

```
def factorial(n: int) -> int:
    """Precondición: n entero >= 0
    Devuelve: n!"""
    fact = 1
    for num in range(n, 1, -1):
        fact *= num
    return fact
```

A continuación, nos dedicamos a estudiar, mediante dos ejemplos, situaciones donde puede ser conveniente utilizar recursion, y situaciones donde no.

## 3. Un ejemplo de recursion eficiente

La función `potencia(b, n)`, que calcula la $n$-esima potencia de $b$, puede escribirse del siguiente modo, a partir de sus propiedades matemáticas:

- $b^0 = 1$
- $b^n = b^{\frac{n}{2}}.b^{\frac{n}{2}}$ si $n$ es par.
- $b^n = b^{\frac{n-1}{2}}.b^{\frac{n-1}{2}}.b$ si $n$ es impar

Antes de programar una función recursiva, es importante distinguir cual es el caso base y cual es el caso recursivo. En este caso, tomaremos 0 como el caso base, y el caso recursivo tendrá dos partes, una para cuando $n$ es par y una para cuando $n$ es impar.

```
def potencia(b: int ,n: int) -> int:
  """Precondición: n >= 0
  Devuelve: b^n."""
  if n == 0:
    # caso base
    return 1
  if n % 2 == 0:
    # caso n par
    p = potencia(b, n // 2)
    return p * p
  else:
    # caso n impar
    p = potencia(b, (n - 1) // 2)
    return p * p * b
```

Comparando con la versión iterativa para calcular la potencia, vemos que la recursión es mas eficiente, pues hay que multiplicar menos veces

```
def potencia(b: int, n: int) -> int:
  """Precondición: n >= 0
  Devuelve: b^n."""
  p = 1
  for _ in range(n):
    p *= b
  return p
```

## 4. Un ejemplo de recursión poco eficiente

Del ejemplo anterior se podría deducir que siempre es mejor utilizar algoritmos recursivos; sin embargo, como ya se dijo, cada situación debe ser analizada por separado. Un ejemplo clásico en el cual la recursión tiene un resultado muy poco eficiente es el de los números de Fibonacci. La sucesión de Fibonacci está definida por la siguiente relación:

$F_n = 0$, si $n = 0$
$F_n = 1$, si $n = 1$
$F_n = F_{n-1} + F_{n-2}$, si $n > 1$

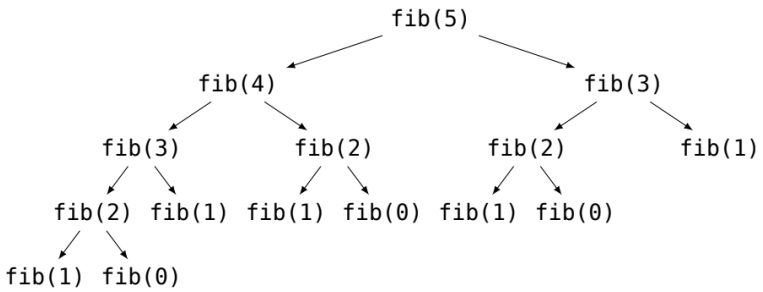
Los primeros números de esta sucesión son:
$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55$.

Dada la definición recursiva de la sucesión, puede resultar muy tentador escribir una función que calcule en valor de `fib(n)` de la siguiente forma:

```
def fib(n: int) -> int:
    """Precondición: n >= 0.
    Devuelve: el número de Fibonacci número n."""
    if n == 0 or n == 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

Si bien esta implementación es muy sencilla y elegante, también es **extremadamente poco eficiente**: para calcular `fib(n - 1)` es necesario calcular `fib(n - 2)`, que luego volverá a ser calculado para obtener el valor `fib(n)`.

Se pueden ubicar las llamadas recursivas en un diagrama de árbol, lo que hace explicito que es poco eficiente.




**Definición Árbol de Llamadas** Un árbol de llamadas (o de recursión) representa gráficamente las sucesivas llamadas recursivas que un programa recursivo puede generar. El nodo superior (nodo raíz) representa a la

primera invocación que se hace a la función y todas las invocaciones recursivas aparecen debajo de él en el orden en el que se realicen las llamadas. El nodo más inferior (nodo hoja) representa a la última invocación que se corresponde con un caso base. Los nodos se etiquetan con el nombre de la función y sus argumentos. Por simplicidad, en algunos casos que no generen confusión, se puede obviar el nombre de la función y etiquetar los nodos solo con los argumentos.

Resulta más conveniente en este caso definir una versión iterativa:

```
def fib(n: int) -> int:
    """Precondición: n >= 0.
    Devuelve: el número de Fibonacci número n."""
    if n == 0 or n == 1:
        return n
    ant2 = 0
    ant1 = 1
    for i in range(2, n + 1):
        fibn = ant1 + ant2
        ant2 = ant1
        ant1 = fibn
    return fibn
```

En el caso iterativo se calcula el número de Fibonacci de forma incremental, de modo que para obtener el valor de `fib(n)` se harán `n - 1` iteraciones.

## 5. Diseño de funciones recursivas

**Ejercicio** Escribir una función recursiva que sume los elementos de una lista en Python.

- Lo primero que debemos hacer es encontrar un caso base. El caso base debe ser sencillo de calcular, muchas veces ni siquiera es necesario calcular nada. Si la lista es vacía, es obvio que la suma de sus elementos será 0.

- Luego, debemos pensar como reducir una lista no vacía para que llegue a tener 0 elementos eventualmente. Hay muchas formas de hacer eso. La forma mas sencilla es quitar un elemento de la lista en cada llamada recursiva

```
def sumar(lista: list[int]) -> int:
  if len(lista) == 0:
    return 0
  return lista[0] + sumar(lista[1:])
```

## 6. Errores comunes

Los errores mas comunes al escribir funciones recursivas son dos:

- No escribir un caso base para la recursión.

- No asegurarse de que el caso recursivo converja a un caso base.

En ambos casos, es usual que el interprete de Python nos muestre un error:

```
RecursionError: maximum recursion depth exceede
```

# 7. Conclusiones

- La **recursión** es el proceso en el cual una función se invoca a sí misma. Este proceso **permite crear un nuevo tipo de ciclos**.

- Siempre que se escribe una función recursiva es importante considerar el **caso base** (el que detendrá la recursión) y el **caso recursivo** (el que realizará la llamada recursiva). **Una función recursiva sin caso base es equivalente a un bucle infinito**.

- **Una función no es mejor ni peor por ser recursiva**. En cada situación a resolver puede ser conveniente utilizar una solución recursiva o una iterativa. Para elegir una o la otra será necesario analizar las características de elegancia y eficiencia.

# Referencias

[1] Apunte de la Facultad de Ingeniería de la UBA, 2da Edición, 2016. *Algoritmos y Programación I: Aprendiendo a programar usando Python como herramienta*. Unidad 18

[2] https://ellibrodepython.com/