

# Programación II

## Recursión

Universidad Nacional de Rosario.  
Facultad de Ciencias Exactas, Ingeniería y Agrimensura.



# Repaso rápido de Python

Python es un lenguaje muy simple...

En Python, una **variable** es un **nombre** asignado a un **valor** que se almacena en la memoria. Este nombre actúa como una referencia que puedes utilizar para acceder o manipular el valor asociado. (según ChatGPT)

Importante no confundir: El nombre es solo un nombre, la variable es el par **nombre** - **valor**.

La **asignación** es el proceso mediante el cuál hacemos que un nombre se refiera a un valor.

```
# Hacemos que el nombre "x" se refiera al valor 23
x = 23
print(x)
```



## Muchos nombres pueden referirse al mismo valor

```
x = 23  
y = x
```

Cuando decimos que `y = x`, `y` no hace referencia a `x`, si no al **valor al que hace referencia** `x`. .

Hay que evitar caer en la tentación de decir que "la variable se llama `x` e `y` hace referencia a `x`". En realidad hay dos variables distintas, pero que referencian el mismo valor.



**Cuando reasignamos un nombre, lo hacemos independientemente**

```
x = 23  
y = x  
x = 12
```



**Un valor existe hasta que ya no existen nombres que lo referencien**

```
x = "hello"  
x = "world"
```



## Una asignación nunca copia los datos

```
nums = [1, 2, 3]
others = nums
```

Existe una sola lista en memoria, y dos nombres que hacen referencia a la misma.

```
nums.append(4)
print(nums) # ?
```

**Los cambios son visibles para todos los nombres que hagan referencia al mismo valor ... ¿verdad?**



**recordatorio** Cuando decimos que un tipo de dato es mutable en Python, significa que el valor del objeto puede ser cambiado o modificado después de que ha sido creado, sin necesidad de crear un nuevo objeto en la memoria. En otras palabras, puedes alterar el contenido del objeto directamente.(según chatGPT).

Algunos tipos de datos mutables son las listas y los diccionarios.  
Cuando:

- tenemos un valor mutable (como una lista).
- ese valor tiene varios nombres referenciándolo.
- el valor cambia.
- todos los nombres "ven" el cambio.



**recordatorio** Un tipo inmutable en Python es un tipo de dato cuyo valor no puede ser cambiado después de haber sido creado. Si intentas modificar un valor inmutable, Python no alterará el objeto original en la memoria; en su lugar, creará un nuevo objeto con el nuevo valor, y el nombre (o variable) apuntará a este nuevo objeto. (según chatGPT)

Algunos tipos de datos inmutable son los enteros, las tuplas, las cadenas de texto, los flotantes y los booleanos.

**Los valores inmutables no soportan aliasing**

```
x = "hello"
y = x
x = x + "world"
print(x) #?
print(y) #?
```





# Lecciones importantes

Las palabras "cambio", "cambiar" tienen un significado ambiguo.

"Cambiar un entero": En realidad lo que estamos haciendo es **reasignar** el mismo nombre a un nuevo valor

```
x = x + 1
```

"Cambiar una lista": En realidad lo que estamos haciendo es **mutar** el valor

```
nums.append(7)
```

Tambien podemos reasignar una lista

```
nums = nums + [7]
```



¿Son lo mismo?

No exactamente, todos los nombres son unas referencias, pero hay algunas referencias que no son nombres. Por ejemplo:

```
nums = [1, 2, 3]
x = nums[1]
```

"nums[1]" es una referencia al valor 2, aunque estrictamente no es un nombre.

Lo mismo ocurre con tuplas y diccionarios.





# Breve repaso de funciones

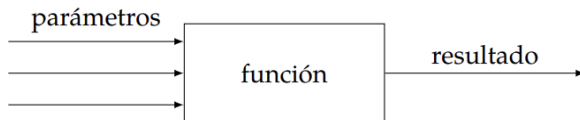
Una función es un fragmento de código que permite efectuar una operación determinada. `abs`, `max`, `min` y `len` son ejemplos de funciones *built-in* o incluidas en Python.

- 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 0 o más *argumentos* (expresados entre paréntesis, y separados por comas), efectúa una operación y devuelve un *resultado*. Por ejemplo, la función `abs` recibe un único argumento (un número) y su resultado es el valor absoluto del número.



# Breve repaso de funciones



# Breve repaso de funciones

Obviamente, además de las funciones incluidas en Python podemos definir nuestras propias funciones utilizando la siguiente sintaxis.

```
def nombre_función(parámetros):  
    código  
    return retorno
```

Por ejemplo:

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

Pero... ¿porqué escribir funciones?



- El **principio de reusabilidad**, que nos dice que si por ejemplo tenemos un fragmento de código usado en muchos sitios, la mejor solución sería pasarlo a una función. Esto nos evitaría tener código repetido, y que modificarlo fuera más fácil, ya que bastaría con cambiar la función una vez.
- El **principio de modularidad**, que defiende que en vez de escribir largos trozos de código, es mejor crear módulos o funciones que agrupen ciertos fragmentos de código en funcionalidades específicas, haciendo que el código resultante sea más fácil de leer



## Argumentos por posición

```
def resta(a: float, b: float) -> float:  
    return a - b  
  
resta(5, 3) # 2
```





## Argumentos por nombre

```
def resta(a: float, b: float) -> float:  
    return a - b
```

```
resta(a=5, b=3) # 2
```

```
resta(a=5, c=3) # Error!
```



## Argumentos por defecto

```
def suma(a: float, b: float, c: float =0) -> float:  
    return a + b + c
```

```
suma(a=5, b=3) # 8
```

```
suma(a=5, b=2, c=3) # 10
```



## La sentencia `return`

El uso de la sentencia `return` permite realizar dos cosas:

- Salir de la función y transferir la ejecución de vuelta a donde se realizó la llamada. Cuando se ejecuta la sentencia `return`, se detiene la ejecución de la función y se "vuelve" o retorna al punto donde fue llamada. Por ello, sólo llamamos a `return` una vez hemos acabado de hacer lo que teníamos que hacer en la función.
- Devolver uno o varios resultados, fruto de la ejecución de la función. Es posible devolver mas de un valor, separados por comas.



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



## Opción 1: Imprimir

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



## Opción 2: Devolver

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



# Imprimir versus devolver

```
>>> 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 hacer esto:

```
>>> 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



# Imprimir versus devolver

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`.





# Funciones que llaman a otras funciones

Naturalmente, 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)
```



Las variables y los parámetros que se declaran dentro de una función no existen fuera de ella, y por eso se las denomina **variables locales**. Fuera de la función se puede acceder únicamente al valor que devuelve mediante `return`.



# La recursión y cómo puede ser que funcione

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.





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 esto se lo llama **caso base** de la recursión.
- Por otro 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.



# Una función recursiva matemática

Es muy común tener definiciones inductivas de operaciones, como por ejemplo la operación factorial.

El factorial de un número entero no negativo  $n$ , denotado como  $n!$ , es el producto de todos los enteros positivos desde 1 hasta  $n$ .

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

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)
```

CE ROSA, P.

# Factorial

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
```



# Factorial

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

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





# 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.



# Factorial iterativo

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
```



# Un ejemplo de recursion eficiente

La función potencia(b, n) puede escribirse del siguiente modo:

$$b^n = \begin{cases} 1 & \text{si } n = 0 \\ b^{\frac{n}{2}} \cdot b^{\frac{n}{2}} & \text{si } n \text{ es par} \\ b^{\frac{n}{2}} \cdot b^{\frac{n}{2}} \cdot b & \text{si } n \text{ es impar} \end{cases}$$

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 tendra dos partes, una para cuando  $n$  es par y una para cuando  $n$  es impar.



# Un ejemplo de recursion eficiente

```
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
```



# Un ejemplo de recursion eficiente

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
```



# 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:

$$F(0) = 0, \quad F(1) = 1$$

$$F(n) = F(n-1) + F(n-2) \quad \text{para } n > 1$$

Los primeros números de esta sucesión son:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55.



# Definición Recursiva de Fibonacci

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

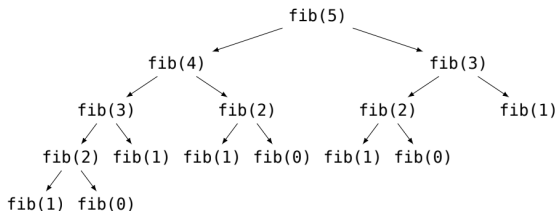
```
def fib(n: int) -> int:
    """Precondición: n >= 0.
    Devuelve: el número de Fibonacci número n."""
    if n == 0 or n == 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

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)`.



# Árbol de llamadas

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





# Definición Iterativa de Fibonacci

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.



En definitiva, vemos que un algoritmo recursivo **no** es mejor que uno iterativo, ni viceversa. En cada situación será conveniente analizar cuál algoritmo provee la solución al problema de forma más clara y eficiente.



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 exceeded



- 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.



¿PREGUNTAS?



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

Unidad 17



<https://ellibrodepython.com/>

