

# **FUNCIONES PYTHON**

---

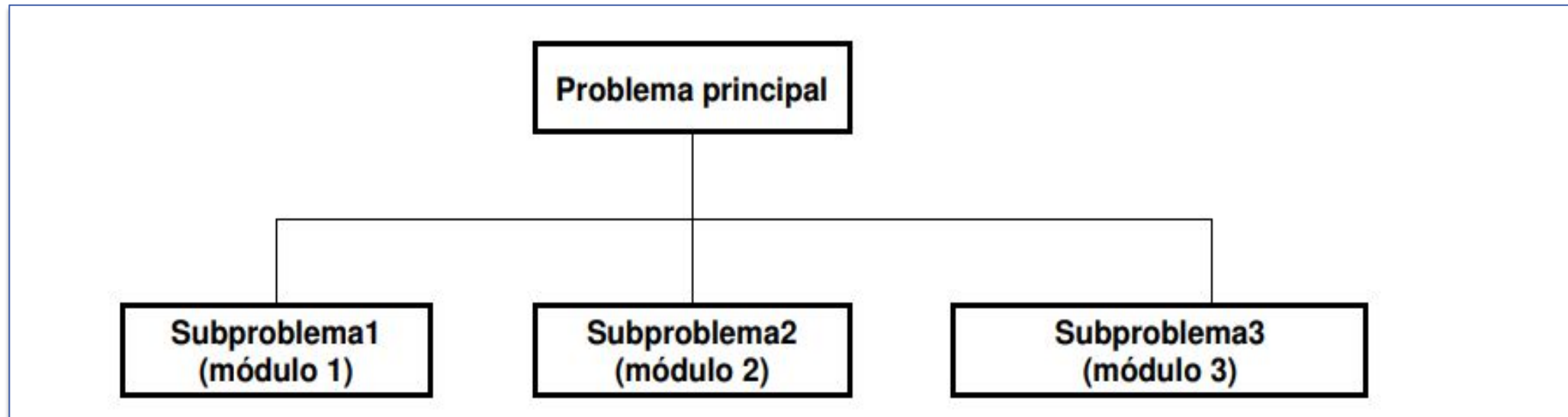
**UNPAZ**

Introducción a la Programación

# Funciones y SubProcesos

## Introducción

Se usa la estrategia de “Dividir y Conquistar”, en otras palabras, es realizar la partición de la tarea en subtareas más fáciles de abordar. Todo problema se puede resolver con funciones o sin ellas, pero su uso adecuado hace que el programa sea más eficiente, fácil de leer y probar, por ello para escribir un programa se divide éste en varios módulos, en lugar de uno solo largo. El programa se divide en muchos módulos (rutinas pequeñas denominadas funciones), que producen muchos beneficios: aislar mejor los problemas, escribir



# Funciones y SubProcesos

---

## Función

Una función es un objeto del ambiente, con nombre, tipo y valor único. El tipo se asocia al valor que retorna la función cuando es evaluada para un conjunto dado de valores de sus argumentos

<b>Funcion</b> nombre (lista de parámetros formales): Tipo de resultado Declaración de variables <b>Inicio</b> Acciones Devolver (constante, variable o expresión) <b>Fin función</b>
--

**Lista de parámetros formales:** contiene las variables que pasan alguna información necesaria para que la función ejecute el conjunto de acciones.

**Tipo de resultado:** señala el tipo de dato que devuelve la función. Declaración de variables: en este lugar se deben declarar los parámetros formales y también aquellas variables que se usarán en la función.

**Cuerpo de la función:** constituye el conjunto de acciones a realizar por la función.

**Retorno resultado:** el único resultado que devuelve la función puede ser un valor constante, o una variable o una expresión válida, la cual debe colocarse entre paréntesis al lado de la acción Devolver. Cuando se ejecuta esta acción se devuelve el control del programa al lugar donde se ha llamado a la función.

# Funciones y SubProcesos

## Subprograma o Procedimiento

Un procedimiento o subrutina es un subalgoritmo que recibiendo o no datos permite devolver varios resultados, un resultado o ninguno. Un procedimiento está compuesto por un grupo de sentencias a las que asigna un nombre (identificador o simplemente nombre del procedimiento) y constituye una unidad de programa. La tarea asignada al procedimiento se ejecutará siempre que se encuentre el identificador (nombre del procedimiento) en el conjunto de sentencias que definen el programa. Cómo trabajar con procedimiento

### ▪ Formato 1

<b>Subrutina</b> nombre() Declaración de variables <b>Inicio</b> Acciones <b>Fin subrutina</b>
--

### ▪ Formato 2

<b>Subrutina</b> nombre (lista de parámetros formales) Declaración de variables <b>Inicio</b> Acciones <b>Fin subrutina</b>
---

**Nombre:** identificador válido

**Lista de parámetros formales:** parámetros formales del procedimiento; sirven para pasar información al procedimiento y/o devolver información del procedimiento a la unidad de programa que le invoca. Están separados por comas, y precedidos por las letras E (entrada), S (Salida) o E/S (Entrada/Salida)

# Funciones y SubProcesos

---

## Parámetros actuales o formales

Las acciones que contienen el llamado al procedimiento constan de dos partes: un identificador (nombre del procedimiento) y una lista de **parámetros actuales**

Nombre (par1, par2, par3, ....)

Los parámetros actuales par1, par2, etc. Contienen los valores que se transferirán al procedimiento. En la declaración de un procedimiento, cuando se incluyen los parámetros, éstos se denominan **parámetros formales** parf1, parf2, parf3, etc. Ellos sirven para contener los valores de los parámetros actuales cuando se invoca al procedimiento.

**Procedimiento** Nombre (parf1, parf2, parf3, ...)

***Importante:** Los parámetros actuales y formales deben coincidir en números, tipo y orden. Es decir debe existir correspondencia entre los parámetros actuales y formales*

# Funciones y SubProcesos

## Argumentos o Parámetros por Valor o Referencia

- Los parámetros formales reciben sus valores iniciales desde los parámetros actuales, es decir desde los valores que se les pasan a través de la llamada.
- Los mismos podrán ser utilizados **por valor** o **por referencia**

**Paso por VALOR:** Los valores iniciales se proporcionan copiando los valores correspondientes en la lista de parámetros actuales.

**Paso por REFERENCIA:** Se produce el paso de la dirección del parámetro actual. En realidad se pasa la posición de memoria. Es decir que una variable pasada por referencia puede ser modificada dentro del subprograma y producir un efecto en el programa de llamada. Utilizaremos la palabra reservada var para indicar este tipo de paso.

<b>Algoritmo EJEMPLO</b> <b>variables</b> entero : A,B,C <b>Inicio</b> A ← 3 B ← 5 C ← 17 SUMAR ( A, A, A+B, C) <b>Escribir</b> ('el valor en C es :', C) <b>fin</b>	<ul style="list-style-type: none"> <li>• <b>Por valor</b>  <b>Subrutina SUMAR (E: x, y, z, v : entero)</b>  <b>inicio</b>              x ← x + 1              v ← y + z  <b>fin subrutina</b></li> <li>• <b>Por referencia</b>  <b>Subrutina SUMAR (E/S: x, y : entero, E: z: entero, E/S: v: entero)</b>  <b>inicio</b>              x ← x + 1              v ← y + z  <b>fin subrutina</b></li> </ul>
---	---

# Funciones y SubProcesos

## Ámbito de las variables

- **Variables Locales:**

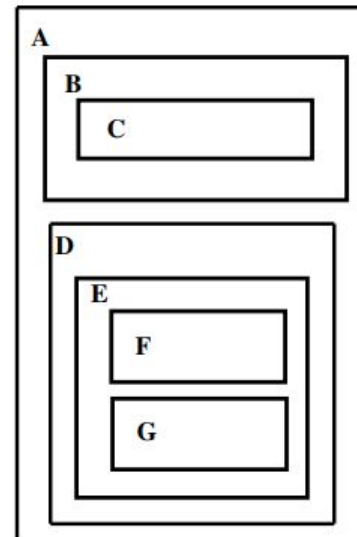
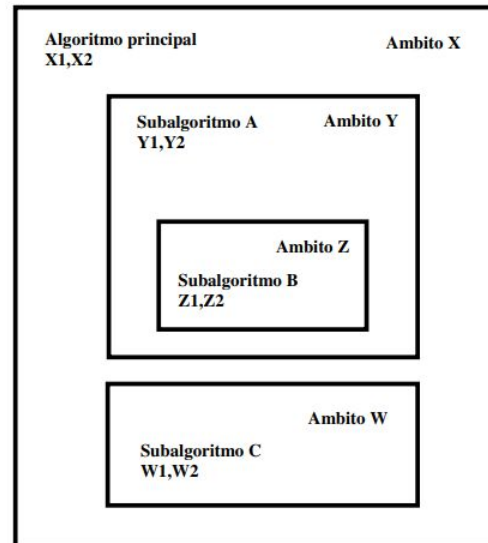
Se declaran dentro de la función y sólo están disponibles durante su ejecución.

Se crean cuando se entra en ejecución una función y se destruyen cuando se termina.

- **Variables globales:**

Se declaran fuera de las funciones. Pueden ser utilizadas por todas las funciones.

Existen durante toda la vida del programa.



Variables definidas en	Accesibles desde
<b>A</b>	<b>A,B,C,D,E,F,G</b>
<b>B</b>	<b>B,C</b>
<b>C</b>	<b>C</b>
<b>D</b>	<b>D,E,F,G</b>
<b>E</b>	<b>E,F,G</b>
<b>F</b>	<b>F</b>
<b>G</b>	<b>G</b>

# Funciones y SubProcesos

1

## Algoritmo Funciones

udpInicio()

**Escribir** "Ingrese un Numero"

**Leer** miNumero

**Escribir** "Duplica (D) o Triplica (T)"

**Leer** miTrioDup

**Escribir** udfDuplicaoTriplica(miNumero,miTrioDup)

**FinAlgoritmo**

**Funcion** resultado ← udfDuplicaoTriplica (pnumero,pTrioDup)

**si** pTrioDup = 'D'

..... resultado ← pnumero\*pnumero

**SiNo**

..... resultado ← pnumero\*pnumero\*pnumero

**FinSi**

**Fin Funcion**

**SubProceso** udpInicio

**Escribir** "\*\*\*\*\*"

**Escribir** "Este es el inicio"

**Escribir** "\*\*\*\*\*"

**FinSubProceso**

1

## Algoritmo Funciones

udpInicio()

'Ingrese un Numero'

miNumero

'Duplica (D) o Triplic...

miTrioDup

udfDupoTrip(miNumero,m...

FinAlgoritmo



# Funciones y SubProcesos

## Algoritmo Funciones

udpInicio()

**Escribir** "Ingrese un Numero"

**Leer** miNumero

**Escribir** "Duplica (D) o Triplica (T)"

**Leer** miTrioDup

**Escribir** udfDuplicaoTriplica(miNumero,miTrioDup)

**FinAlgoritmo**

**Funcion** resultado ← udfDuplicaoTriplica (pnumero,pTrioDup)

**si** pTrioDup = 'D'

..... resultado ← pnumero\*pnumero

**SiNo**

..... resultado ← pnumero\*pnumero\*pnumero

**FinSi**

**Fin Funcion**

**SubProceso** udpInicio

**Escribir** "\*\*\*\*\*"

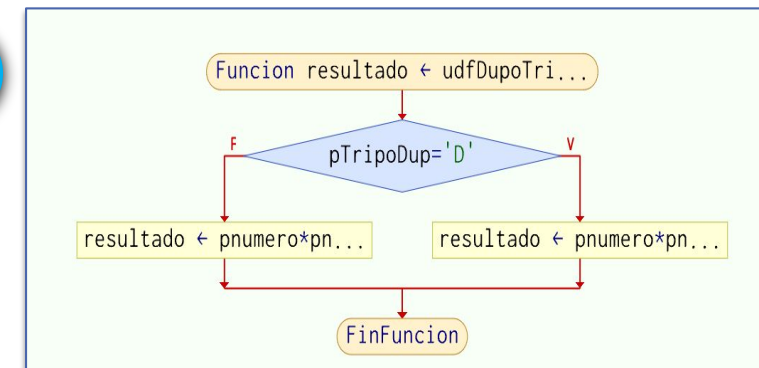
**Escribir** "Este es el inicio"

**Escribir** "\*\*\*\*\*"

**FinSubProceso**

2

2



# Funciones y SubProcesos

## Algoritmo Funciones

```

udpInicio()
  Escribir "Ingrese un Numero"
  Leer miNumero
  Escribir "Duplica (D) o Triplica (T)"
  Leer miTrioDup
  Escribir udfDuplicaoTriplica(miNumero,miTrioDup)

```

## FinAlgoritmo

**Funcion** resultado ← udfDuplicaoTriplica (pnumero,pTrioDup)

```

  si pTrioDup = 'D'
  ..... resultado ← pnumero*pnumero
  SiNo
  ..... resultado ← pnumero*pnumero*pnumero

```

**FinSi**

**Fin Funcion**

**SubProceso** udpInicio

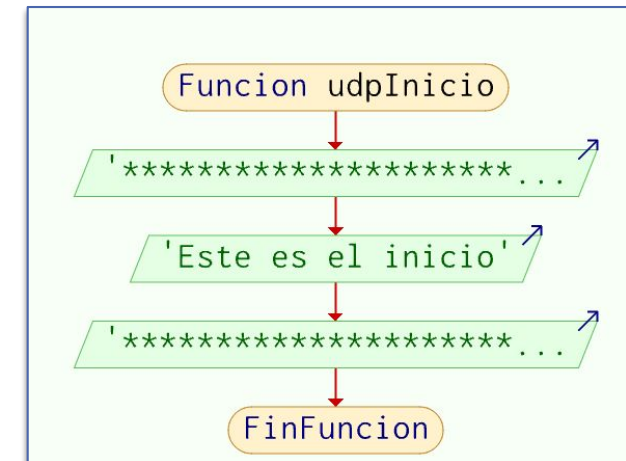
```

  Escribir "*****"
  Escribir "Este es el inicio"
  Escribir "*****"

```

**FinSubProceso**

3



3

# Funciones

---

## ¿Qué es una función en Python?

En Python, una función es un grupo de declaraciones relacionadas que realizan una tarea específica.

Las funciones ayudan a dividir nuestro programa en partes más pequeñas y modulares. A medida que nuestro programa crece más y más, las funciones lo hacen más organizado y manejable.

Además, evita la repetición y hace que el código sea reutilizable.

## Sintaxis de la función

```
def nombre_función(parámetros):  
    """cadena de documentación"""  
    declaraciones
```

# Funciones

---

## Sintaxis de la Función

- Palabra clave <def> que marca el inicio del encabezado de la función.
- Un nombre de función para identificar de forma única la función. La nomenclatura de funciones sigue las mismas reglas de escritura de identificadores en Python .
- Parámetros (argumentos) a través de los cuales pasamos valores a una función son opcionales.
- Dos puntos (:) para marcar el final del encabezado de la función.
- Cadena de documentación opcional (docstring) para describir lo que hace la función.
- Una o más declaraciones de Python válidas que componen el cuerpo de la función. Las declaraciones deben tener el mismo nivel de sangría (generalmente 4 espacios).
- La declaración opcional <return> es utilizada para devolver un valor de la función.

# Funciones

## Sintaxis de la Función

### Ejemplo de una función

```
def greet(name):  
    """  
    This function greets to  
    the person passed in as  
    a parameter  
    """  
    print("Hello, " + name + ". Good morning!")
```

### ¿Cómo llamar a una función en Python?

Una vez que hemos definido una función, podemos llamarla desde otra función, programa o incluso desde el indicador de Python. Para llamar a una función, simplemente escribimos el nombre de la función con los parámetros apropiados.

```
>>> greet('Paul')  
Hello, Paul. Good morning!
```

# Funciones

## The return statement

Esta instrucción puede contener una expresión que se evalúa y se devuelve el valor. Si no hay una expresión en la declaración o la <return> declaración en sí no está presente dentro de una función, entonces la función devolverá el <None> objeto.

### Syntax of return

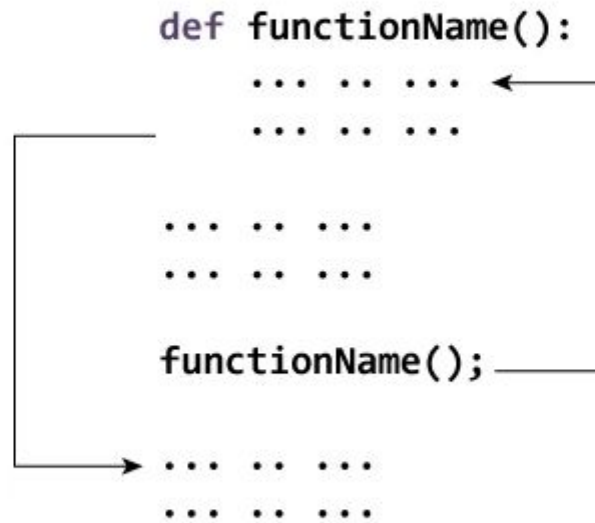
```
return [expression_list]
```

### Example of return

```
def absolute_value(num):  
    """This function returns the absolute  
    value of the entered number"""  
  
    if num >= 0:  
        return num  
    else:  
        return -num  
  
print(absolute_value(2))  
print(absolute_value(-4))
```

# Funciones

## How Function works in Python?



```
def functionName():  
    ... ..  
    ... ..  
  
    ... ..  
    ... ..  
  
functionName();  
  
    ... ..  
    ... ..
```

The diagram illustrates the execution flow of a function call in Python. It shows a function definition `def functionName():` followed by its body. Below the definition, the function is called `functionName();`. An arrow points from the function call to the function definition, indicating the call. Another arrow points from the end of the function body back to the function call, representing the return value being passed back to the caller.

Working of functions in Python

# Funciones

## Alcance y vida útil de las variables

El alcance de una variable es la parte de un programa donde se reconoce la variable. Los parámetros y variables definidos dentro de una función no son visibles desde fuera de la función. Por lo tanto, tienen un **alcance local**.

El tiempo de vida de una variable es el período durante el cual la variable existe en la memoria. La vida útil de las variables dentro de una función es mientras se ejecuta la función.

Se destruyen una vez que regresamos de la función. Por lo tanto, una función no recuerda el valor de una variable de sus llamadas anteriores.

Aquí hay un ejemplo para ilustrar el alcance de una variable dentro de una función.

```
def my_func():  
    x = 10  
    print("Value inside function:",x)  
  
x = 20  
my_func()  
print("Value outside function:",x)
```

Producción

```
Valor dentro de la función: 10  
Valor fuera de función: 20
```





# Funciones

---

## Tipos de funciones

Básicamente pueden dividirse en dos tipos:

- User-defined functions - Funciones creadas por los usuarios desarrolladores. Son las abordadas recientemente.
- Built-in functions - Funciones integradas en Python.

# Funciones

## Built-in Functions

Built-in Functions			
<b>A</b> abs() all() any() ascii()	<b>E</b> enumerate() eval() exec()	<b>L</b> len() list() locals()	<b>R</b> range() repr() reversed() round()
<b>B</b> bin() bool() breakpoint() bytearray() bytes()	<b>F</b> filter() float() format() frozenset()	<b>M</b> map() max() memoryview() min()	<b>S</b> set() setattr() slice() sorted() staticmethod() str() sum() super()
<b>C</b> callable() chr() classmethod() compile() complex()	<b>G</b> getattr() globals()	<b>N</b> next()	<b>T</b> tuple() type()
<b>D</b> delattr() dict() dir() divmod()	<b>H</b> hasattr() hash() help() hex()	<b>O</b> object() oct() open() ord()	<b>V</b> vars()
	<b>I</b> id() input() int() isinstance() issubclass() iter()	<b>P</b> pow() print() property()	<b>Z</b> zip()  __import__()

# Funciones - Argumentos

---

## Argumentos de funciones variables

Hasta ahora, las funciones tenían un número fijo de argumentos. En Python, hay otras formas de definir una función que puede tomar un número variable de argumentos.

A continuación se describen tres formas diferentes de este tipo.

- Argumentos predeterminados (con Default)
- Argumentos de palabras clave
- Argumentos arbitrarios

# Funciones - Argumentos

## Argumentos predeterminados (Default)

Los argumentos de función pueden tener valores predeterminados en Python. Podemos proporcionar un valor predeterminado a un argumento utilizando el operador de asignación (=). Aquí hay un ejemplo.

```
def greet(name, msg="Good morning!"):
    """
    This function greets to
    the person with the
    provided message.

    If the message is not provided,
    it defaults to "Good
    morning!"
    """

    print("Hello", name + ', ' + msg)

greet("Kate")
greet("Bruce", "How do you do?")
```

### Output

```
Hello Kate, Good morning!
Hello Bruce, How do you do?
```



# Funciones - Argumentos

## Argumentos de palabra clave ( Keyword Arguments)

Cuando llamamos a una función con algunos valores, estos valores se asignan a los argumentos según su posición.

Por ejemplo, en la función anterior greet(), cuando la llamamos como greet("Bruce", "How do you do?"), el valor "Bruce" se asigna al argumento <name> y de manera similar "How do you do?" a <msg>.

Python permite llamar a funciones usando argumentos de palabras clave. Cuando llamamos funciones de esta manera, el orden (posición) de los argumentos se puede cambiar. Las siguientes llamadas a la función anterior son todas válidas y producen el mismo resultado.

```
# 2 keyword arguments
greet(name = "Bruce",msg = "How do you do?")

# 2 keyword arguments (out of order)
greet(msg = "How do you do?",name = "Bruce")

1 positional, 1 keyword argument
greet("Bruce", msg = "How do you do?")
```

Aclaración: Podemos mezclar argumentos posicionales con argumentos de palabras clave durante una llamada de función. Pero debemos tener en cuenta que los argumentos de palabras clave deben seguir a los argumentos posicionales.



```
greet(name="Bruce","How do you do?")
```

# Funciones - Argumentos

## Argumentos arbitrarios (Arbitrary)

A veces, no sabemos de antemano la cantidad de argumentos que se pasarán a una función. Python nos permite manejar este tipo de situaciones a través de llamadas a funciones con un número arbitrario de argumentos.

En la definición de la función, usamos un asterisco (\*) antes del nombre del parámetro para indicar este tipo de argumento. Aquí hay un ejemplo.

```
def greet(*names):  
    """This function greets all  
    the person in the names tuple."""  
  
    # names is a tuple with arguments  
    for name in names:  
        print("Hello", name)  
  
greet("Monica", "Luke", "Steve", "John")
```

### Output

```
Hello Monica  
Hello Luke  
Hello Steve  
Hello John
```

# Funciones Recursivas

---

## Recursividad

La recursividad (recursión) es la propiedad por la cual una función se llama a sí misma.

Se puede considerar la recursividad como una alternativa a la iteración. La recursión permite especificar soluciones naturales, sencillas, que serían, en caso contrario, difíciles de resolver. Toda función recursiva debe contemplar un caso base o condición de salida, para terminar, o la recursividad no podrá terminar nunca.

```
funcion_recursiva( /* parámetros recibidos por la función */ )  
  
{  
  
    /* Código caso base */  
  
    funcion_recursiva( ); /* llamada a la función misma */  
  
}
```

# Funciones Recursivas

---

## Caso "Factorial con recursividad"

Ejemplo el factorial de 7 es

$$7! = 7 * 6 * 5 * 4 * 3 * 2 * 1$$

Y en general el factorial cualquier número es

$$n! = n * (n-1) * (n-2) * (n-3) * \dots * 2 * 1$$

Si comparas con el factorial de 6, verás que se parece mucho al de 7:

$$6! = 6 * 5 * 4 * 3 * 2 * 1$$

De modo que podrías escribir el factorial de 7 a partir del factorial de 6:

$$7! = 7 * (6!)$$

Sólo falta pensar cual es el "**caso base**", la solución más sencilla posible para este problema. En el caso del factorial, se trata del factorial de 0, que tiene como valor 1:

$$0! = 1$$



# Funciones Recursivas

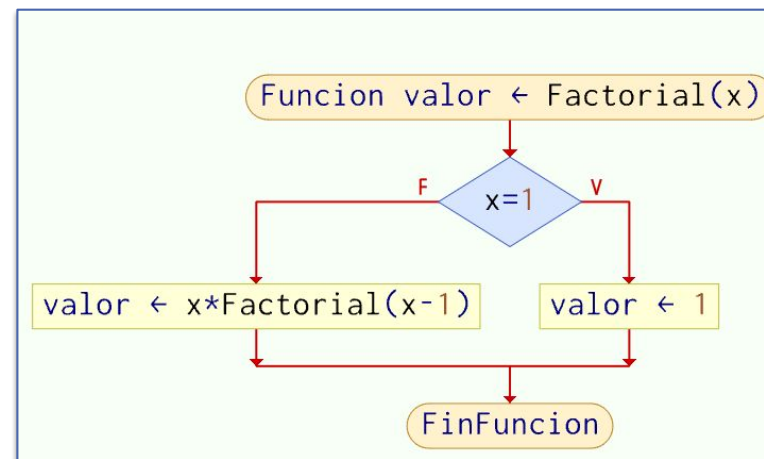
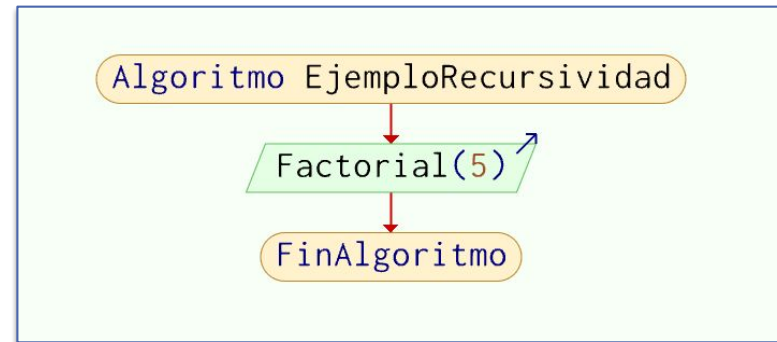
## Caso "Factorial con recursividad"

Ejemplo el factorial de 5 es

$$5! = 5 * 4 * 3 * 2 * 1$$

```

1  Algoritmo EjemploRecursividad
2
3  Escribir Factorial(5)
4
5  FinAlgoritmo
6
7  Funcion valor ← Factorial(x)
8      Si x = 1 Entonces
9          valor ← 1
10     Sino
11         valor ← x * Factorial(x-1)
12     FinSi
13 FinFuncion
14 |
  
```



Calculo	Resultado
5*4!	120
4*3!	24
3*2!	6
2*1!	2
1!	Return 1

# Funciones Recursivas

## Recursividad en Python

La recursividad (recursión) es la propiedad por la cual una función se llama a sí misma. En Python, sabemos que una función puede llamar a otras funciones. Incluso es posible que la función se llame a sí misma. Estos tipos de construcción se denominan funciones recursivas.

La siguiente imagen muestra el funcionamiento de una función recursiva llamada, en este ejemplo <recurse>



# Funciones Recursivas

## Caso “Factorial con recursividad” con Python

```
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))

num = 3
print("The factorial of", num, "is", factorial(num))
```

```
x = factorial(3)

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

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

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

3\*2 = 6  
is returned

2\*1 = 2  
is returned

1  
is returned

Working of a recursive factorial function

Calculo	Resultado
3*2!	6
2*1!	2
1!	Return 1

# Funciones Recursivas

---

## Caso “Factorial Utilizando Iteraciones” con Python

```
1 def factorial(n):  
2     fact=1  
3     if int(n) >= 1:  
4         for i in range (1,int(n)+1):  
5             fact = fact * i  
6         return fact  
7  
8 num = int(input("Enter the Number: "))  
9  
10 print("factorial of ",num," : ",end="")  
11 print(factorial(num))
```

# Funciones Recursivas

---

## Algunas Conclusiones

### Ventajas de la recursividad

- Las funciones recursivas hacen que el código se vea limpio y elegante.
- Una tarea compleja se puede dividir en sub-problemas más simples usando recursividad.
- La generación de secuencias es más fácil con la recursividad que con alguna iteración anidada.

### Desventajas de la recursividad

- A veces, la lógica detrás de la recursividad es difícil de seguir.
- Las llamadas recursivas son costosas (ineficientes) ya que ocupan mucha memoria y tiempo.
- Las funciones recursivas son difíciles de depurar.

### Conclusión.

La recursividad puede ser una alternativa o herramienta algorítmica a la hora de poder solucionar un problema que no se pueda resolver de forma iterativa, pero debido al costo de memoria, puede que esta solución derive en un código más lento y de mayor demanda de recursos al ordenador. Usar la recursividad solo si es necesario.

# Funciones y Ámbito de las variables

## Variables Globales

En Python, una variable declarada fuera de la función o en el ámbito global se conoce como variable global. Esto significa que se puede acceder a una variable global dentro o fuera de la función.

```
x = "global"

def foo():
    print("x inside:", x)

foo()
print("x outside:", x)
```

## Output

```
x inside: global
x outside: global
```

# Funciones y Ámbito de las variables

## Variables Locales

Una variable declarada dentro del cuerpo de la función o en el ámbito local se conoce como variable local.

### Ejemplo 2: Acceder a la variable local fuera del alcance

```
def foo():  
    y = "local"  
  
foo()  
print(y)
```

### Output

```
NameError: name 'y' is not defined
```

# Funciones y Ámbito de las variables

## Variables Locales y Globales

```
x = 5

def foo():
    x = 10
    print("local x:", x)

foo()
print("global x:", x)
```

### Output

```
local x: 10
global x: 5
```



# Funciones y Ámbito de las variables

## Variables Nonlocal

Las variables no locales se utilizan en funciones anidadas cuyo alcance local no está definido. Esto significa que la variable no puede estar ni en el ámbito local ni en el global.

### Ejemplo 6: crear una variable no local

```
def outer():  
    x = "local"  
  
    def inner():  
        nonlocal x  
        x = "nonlocal"  
        print("inner:", x)  
  
    inner()  
    print("outer:", x)  
  
outer()
```

### Output

```
inner: nonlocal  
outer: nonlocal
```

En el código aquí presentado, hay una función `inner()` anidada. Usamos `nonlocal` como palabra clave para crear una variable no local. La función `inner()` se define en el ámbito de otra función `outer()`.

**Nota:** Si cambiamos el valor de una variable no local, los cambios aparecen en la variable local.

# Funciones y Ámbito de las variables

## Uso de la palabra clave “Global”

En Python, global la palabra clave le permite modificar la variable fuera del alcance actual. Se utiliza para crear una variable global y realizar cambios en la variable en un contexto local.

## Reglas de palabra clave “global”

- Cuando creamos una variable dentro de una función, es local por defecto.
- Cuando definimos una variable fuera de una función, es global por defecto. No tienes que usar la palabra clave global.
- Usamos la palabra clave global para leer y escribir una variable global dentro de una función.
- El uso de una palabra clave global fuera de una función no tiene efecto.

```
c = 0 # global variable

def add():
    global c
    c = c + 2 # increment by 2
    print("Inside add():", c)

add()
print("In main:", c)
```

## Output

```
inner: nonlocal
outer: nonlocal
```

# Gracias

