

INTRODUCCIÓN A LA PROGRAMACIÓN

CON  python

FUNCIONES



Consejo del día

«Programming is not about typing, it's about thinking.» – Rich Hickey

Funciones


Ya te has familiarizado con el uso de funciones como **len()**, **str()**, **int()**, **input()** y **print()** con anterioridad

Python provee varias funciones predefinidas como las anteriores, pero tú también puedes **escribir tus propias funciones**.



Funciones


A lo largo del tiempo y en diferentes lenguajes el concepto de función aparece definido también por otros nombres: **subprograma**, **subrutina** o **procedimiento** son los más habituales y con diferentes matices denotan lo mismo.



Función


Una **función** es un fragmento de código con un nombre asociado que realiza una tarea específica y devuelve un valor.

Una función es como un **mini programa** dentro de un programa. Puede ser usada cuando quieras y cuantas veces lo necesites.




Función

Cuando quieras realizar una tarea particular que has definido en una función, llamas a la función responsable de la misma.




Función

Si necesitas realizar esa tarea **varias veces** a lo largo de su programa, **no** necesitas escribir todo el código para la misma tarea una y otra vez; sólo tienes que llamar a la función dedicada a manejar esa tarea, y la llamada le dice a Python que ejecute el código dentro de la función.



La necesidad de funciones

El concepto de función es básico en prácticamente cualquier lenguaje de programación. Se trata de una estructura del lenguaje que nos permite agrupar código. Persigue dos **objetivos** claros:

- No repetir trozos de código durante nuestro programa.
 - Reutilizar el código para distintas situaciones.
- 

Funciones

En este tema también aprenderás formas de **pasar información** a funciones. Aprenderás a escribir ciertas funciones cuyo trabajo principal es mostrar información y otras funciones diseñadas para procesar información y devolver un valor o conjunto de valores.

Por último, aprenderás a almacenar funciones en archivos separados llamados **módulos** para ayudar a organizar tus archivos de programa
[NO ENTRA]

Funciones

Descubrirás que el uso de **funciones** hace que sus programas sean más fáciles de escribir, leer, probar y corregir.



Funciones

Una función viene **definida** por su **nombre**, sus **parámetros** y su valor de **retorno**.

```
import math

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

Cuando se quiere utilizar esta función, se la llama (**invoca**). Una llamada a una función le dice a Python que ejecute el código de la función.

Funciones

Para llamar a una función, se escribe el nombre de la función, seguido de la información necesaria entre paréntesis

```
import math

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

diametro = 10
area = area_circulo(diametro/2)
print(f'Área del círculo de diámetro {diametro} es {round(area,2)}.')
```

comienza el programa →

definición

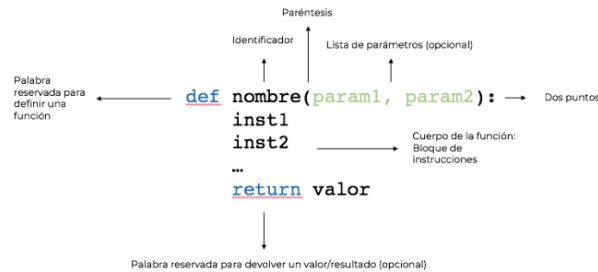
invocación

La parametrización de las funciones las convierte en una poderosa herramienta ajustable a las circunstancias que tengamos.

Funciones. Definición

Para definir una función utilizamos la palabra reservada **def** seguida del **nombre** de la función. A continuación aparecerán 0 o más **parámetros** separados por comas (entre paréntesis), finalizando la línea con **dos puntos** :

En la siguiente línea empezaría el **cuerpo** de la función que puede contener 1 o más sentencias, incluyendo (o no) una **sentencia de retorno** con el resultado mediante **return**.



Funciones. Definición

Hagamos una primera función sencilla que no tiene parámetros

```
def saludar():
    print ("¡Hola!")
```

- Nótese la indentación (sangrado) del cuerpo de la función.
- Los nombres de las funciones siguen las mismas reglas que las variables.

Pero si se ejecuta el código de este programa no hace nada porque solo está la definición de la función. Para que se ejecute la misma hay que **invocarla**.

Funciones. Invocación

Para invocar (o «llamar») a una función sólo tendremos que escribir su nombre seguido de paréntesis. En el caso de la función sencilla (vista anteriormente) se haría así:

```
def saludar():  
    print ("¡Hola!")
```

```
saludar()
```

Como era de esperar, al invocar a esta función obtenemos el mensaje por pantalla `¡Hola!`, fruto de la ejecución del cuerpo de la función.

Funciones. Retornar un valor

Las funciones pueden retornar (o «devolver») un valor. Veamos un ejemplo muy sencillo de una función que calcula la media de dos números

```
def media(x,y):  
    resultado = (x+y)/2  
    return resultado
```

```
a = 2  
b = 3  
m = media(a,b)  
print(m)
```

La ejecución de este código da como resultado `2.5`

Funciones. Retornar un valor

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.
- Devolver uno o varios parámetros, fruto de la ejecución de la función.

En lo relativo a lo primero, una vez se llama a **return** se para la ejecución de la función y se vuelve o retorna al punto donde fue llamada. Es por ello por lo que el código que va después del **return** no es ejecutado en el siguiente ejemplo.

```
def mi_funcion():
    print('Entra en mi_funcion')
    return
    print('No llega')

mi_funcion()
```

Por ello, sólo llamamos a **return** una vez hemos acabado de hacer lo que teníamos que hacer en la función.

Funciones. Retornar un valor

Pero no sólo podemos invocar a la función directamente, también la podemos integrar en otras expresiones. Por ejemplo en condicionales:

```
def media(x,y):
    resultado = (x+y)/2
    return resultado

a = 2
b = 3
if media(a,b) >= 5:
    print("aprobado")
else:
    print("suspense")
```

Funciones. No retornar un valor

Si una función **no incluye un return de forma explícita**, devolverá **None** de forma implícita. Este es el ejemplo de la función saludar

```
def saludar():
    print ("¡Hola!")
```

Funciones. Retornando múltiples valores

Una función puede devolver más de un objeto separándolos por comas tras la palabra reservada return. En tal caso, la función agrupará los objetos en una **tupla** y devolverá la **tupla**.

```
def multiple():
    return 0, 1

resultado = multiple()
print(resultado)
```

mostrará por pantalla: `(0, 1)`

Podremos aplicar el **desempaquetado de tuplas** sobre el valor retornado por la función:

```
def multiple():
    return 0, 1

a, b = multiple()
print(a)
print(b)
```

Nota: más adelante veremos la estructura de datos **tupla**

Funciones. Valores retornados

Una función puede devolver cualquier tipo de valor que necesites, incluyendo estructuras de datos más complicadas como listas y diccionarios. Por ejemplo, la función siguiente toma partes de un nombre y devuelve un **diccionario** que representa a una persona:

```
def crear_persona(nombre, apellido):
    persona = {'nombre': nombre, 'apellido': apellido}
    return persona

cantante = crear_persona('Amy', 'Winehouse')
print(cantante)
```

```
{'nombre': 'Amy', 'apellido': 'Winehouse'}
```

Nota: más adelante veremos la estructura de datos diccionario

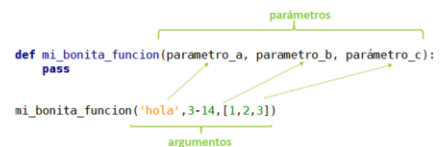
Funciones. Parámetros y argumentos

Si una función no dispusiera de valores de entrada estaría muy limitada en su actuación. Es por ello que los **parámetros** nos permiten variar los datos que consume una función para obtener distintos resultados.

Veamos otra función con parámetros:

```
def raiz_cuadrada(valor):
    return valor ** (1/2)

print(raiz_cuadrada(4))
```



En este caso, el valor 4 es un **argumento** de la función.

El resultado de ejecutar este código es **2.0**

Funciones. Parámetros y argumentos

Cuando llamamos a una función con argumentos, los valores de estos argumentos se copian en los correspondientes parámetros dentro de la función*:

```
def mi_bonita_funcion(parametro_a, parametro_b, parametro_c):
    pass

mi_bonita_funcion('hola', 3-14, [1,2,3])
```

La **sentencia pass** permite «no hacer nada». Es una especie de «placeholder».


*nota: esta es una de las dos formas de correspondencia de argumentos con parámetros

Actividad

1. Escribe una función llamada **ciudad_pais()** que reciba el nombre de una ciudad y su país. La función debe devolver una cadena con el siguiente formato: "Madrid, España". Llama a la función con al menos tres pares ciudad-país, e imprime el valor devuelto.
2. Escribe una función **valor_absoluto()** que reciba un número y devuelva su valor absoluto


Funciones. Parámetros y argumentos

En los apartados anteriores hemos establecido una clara diferencia entre los términos parámetro y argumento, que con frecuencia aparecen usados como sinónimos. A los primeros a veces se les denomina **parámetros formales** y a los segundos **parámetros reales** en otros lenguajes.

- los **argumentos** se utilizan en las llamadas a las funciones
 - los **parámetros** son internos a las funciones y reciben inicialmente los valores de los argumentos.
- 

Funciones. Paso de argumentos

Los argumentos se pueden pasar a una función de dos formas:

- **Argumentos posicionales**: Se hace coincidir cada argumento en la llamada a la función con un parámetro en la definición de la función, en orden de izquierda a derecha.
 - **Argumentos nominales**: Se indica explícitamente el nombre del parámetro al que se asocia un argumento de la forma **parametro = argumento**.
- 

Funciones. Paso de argumentos

Posicionales

```
def bienvenida(nombre, apellido):
    print (f'¡Bienvenido a Python, {nombre} {apellido}!')

bienvenida('Alfredo', 'Rosales')
```

Nominales

```
def bienvenida(nombre, apellido):
    print (f'¡Bienvenido a Python, {nombre} {apellido}!')

bienvenida(apellido='Rosales', nombre='Alfredo')
```

Funciones. Parámetros por defecto

En la definición de una función se puede asignar a cada parámetro un valor por defecto, de manera que si se invoca la función sin proporcionar ningún argumento para ese parámetro, se utiliza el valor por defecto.

```
def bienvenida(nombre, lenguaje='Python'):
    print(f'¡Bienvenida a {lenguaje}, {nombre}!')

bienvenida('Patricia')
bienvenida('Patricia', 'Java')
```

La ejecución de este código produce la salida:

```
¡Bienvenida a Python, Patricia!
¡Bienvenida a Java, Patricia!
```

Los parámetros con un valor por defecto deben indicarse después de los parámetros sin valores por defectos. De lo contrario se produce un error.

Funciones. Valores opcionales

```
def bienvenida(nombre, apellido=None):  
    mensaje = f'Bienvenido a Python, {nombre}'  
    if apellido is None:  
        print (f';{mensaje}!')  
    else:  
        print (f';{mensaje} {apellido}!')  
  
bienvenida('Alfredo')  
bienvenida('Alfredo', 'Rosales')
```

Actividad

3. Crea un programa que pida dos número enteros al usuario y diga si alguno de ellos es múltiplo del otro. Crea una función **es_multiplo()** que reciba los dos números, y devuelve si el primero es múltiplo del segundo.
4. Escribe una función **es_primo()** que reciba un número y devuelva True si el número es primo y False en caso contrario

Funciones. Ámbito de variables: variables locales y globales

Los parámetros y las variables declaradas dentro de una función son de **ámbito local**, mientras que las definidas fuera de ella son de ámbito **global**.

El ámbito de una variable es el lugar en el código fuente donde esta variable se puede usar. Una **variable local** que se declaran dentro de una función no se pueden utilizar fuera de esa función. veamos un ejemplo:

```
def bienvenida():
    lenguaje = 'Python'
    print('¡Bienvenido a', lenguaje + '!')

bienvenida()
print(lenguaje)
```

La ejecución de este código abortará el programa con el siguiente error

```
print(lenguaje)
NameError: name 'lenguaje' is not defined
```

Funciones. Ámbito de variables: variables locales y globales

Las **variables globales** se definen en el cuerpo principal del programa y permiten su acceso desde cualquier lugar.

Si en el ámbito local de una función existe una variable que también existe en el ámbito global, durante la ejecución de la función la variable global queda eclipsada por la variable local y no es accesible hasta que finaliza la ejecución de la función.

```
def bienvenida():
    lenguaje = 'Python'
    print('¡Bienvenido a', lenguaje + '!')

lenguaje = 'Java'
bienvenida()
print(lenguaje)
```

La ejecución de este código dará el siguiente resultado:

```
¡Bienvenido a Python!
Java
```


Funciones. Acceso desde una función a una variable externa

Si tenemos una función definida dentro del programa principal, es cierto que desde la función se puede acceder a variables definidas en el programa principal, siempre que no haya ningún parámetro o variable local con el mismo nombre.

```
def func(b):
    c = 30*a          Las variables b y c son locales y a es global
    print('func a =', a, 'b =', b, 'c =', c)

# Programa principal
a = 5
b = 10
c = 20
func(1)
print('Prog. principal a =', a, 'b =', b, 'c =', c)
```

Resultado

```
En mi_funcion a = 5 b = 1 c = 150
En el programa principal a = 5 b = 10 c = 20
```

Funciones. Acceso desde una función a una variable externa

Este ejemplo que acabamos de ver es una muestra de **pésima programación**, pues perdemos el encapsulamiento del código y el mantenimiento del código se hace imposible.

Normalmente, **no se deben modificar variables globales desde dentro de las funciones**.

Por norma general se podría decir que salvo que estemos muy seguros de lo que estamos haciendo, no es muy común hacer uso de variables globales en funciones. Si quieres hacerlo dentro de una función debes usar la palabra reservada **global** seguido del nombre de la variable.

```
def suma(valor1, valor2):
    global resultado
    resultado = valor1 + valor2
```

La salida de este programa es **8**

```
suma(3,5)
print(resultado)
```

Funciones. Documentación

Ahora que ya tenemos nuestras propias funciones creadas, tal vez alguien se interese en ellas y podamos compartírselas. Las funciones pueden ser muy complejas, y leer código ajeno no es tarea fácil. Es por ello por lo que es importante documentar las funciones. Es decir, añadir comentarios para indicar que hacen y como deben ser usadas.

```
def mi_funcion_suma(a, b):
    """
    Descripción de la función. Como debe ser usada,
    que parámetros acepta y que devuelve
    """
    return a+b
```

Funciones. Documentación


Para ello debemos usar la **triple comilla** `"""` al principio de la función. Se trata de una especie de comentario que podemos usar para indicar como la función debe ser usada. No se trata de código, es un simple comentario un tanto especial, conocido como **docstring**.

Ahora cualquier persona que tenga nuestra función (previamente definida en un módulo), podrá llamar a la función `help()` y obtener la ayuda de como debe ser usada.

```
help(mi_funcion_suma)
```


Ventajas del uso de funciones

Las funciones brindan dos beneficios importantes de los que se pueden derivar muchos otros:

- **Estructura**: Es un recurso que permite descomponer una tarea compleja en varias subtarefas de menor complejidad, que puedan ser abordables con mayores garantías de éxito.
 - **Abstracción**: Las funciones ocultan detalles tras una interfaz pública bien definida. Se hace abstracción de los detalles de implementación y lo que interesa únicamente son los valores de entrada que se le suministran y los resultados que devuelve.
- 

Ventajas del uso de funciones

A partir de estas dos características básicas, se derivan el resto de las ventajas que brinda el uso de las funciones:

- Permiten **reutilizar** código sin tener que reescribirlo cada vez.
 - Permite el **encapsulamiento** del código de la función. Así, por ejemplo, una vez dado por válido el código interno de una función, los errores de un programa no serán imputables a la implementación interna de la función.
 - La implementación interna puede cambiar sin que el programador que use esas funciones tenga que preocuparse de ello.
 - Hace que el código resultante sea más **claro y mantenible**.
- 

Ventajas del uso de funciones

- Las funciones brindan el mecanismo para **dividir un problema** grande en subproblemas pequeños, acotando la interacción entre los mismos a los datos intercambiados a través de su interfaz pública.
- Es el mecanismo ideal para permitir la **colaboración entre varios programadores**: una vez puestos de acuerdo en la interfaz, cada cual tiene la libertad de programar la solución a los subproblemas parciales sin temor a que le afecte lo hecho por otros programadores (siempre que el resultado brindado sea el correcto).



Funciones. Pautas para el diseño de funciones.


1. **Cada función debe tener un único propósito.** Es el **principio de responsabilidad única**.
 - El objetivo perseguido con la función debería ser fácilmente identificado con un nombre corto.
 - Si una función hace múltiples tareas de forma consecutiva, debería rehacerse en múltiples funciones.
2. **No te repitas** (DRY, Don't repeat yourself).
 - Si un fragmento de código aparece varias veces repetido, es una buena oportunidad para darle un nombre e invocarlo múltiples veces.
3. **Las funciones deben ser generales.**
 - No tiene sentido, por ejemplo, definir una función específica para elevar un número a la quinta, cuando podemos definir con carácter general, una función que eleve un número a cualquier exponente.

Principios de responsabilidad única y de generalidad

Desde el punto de vista de la ingeniería del software es deseable que una función realice una única labor, lo se conoce como principio de responsabilidad única. Veamos un ejemplo de **diseño de función desafortunado**:

```
def area_circulo_desafortunado(r):  
    area = 3.1415*r**2  
    print(f'Área del círculo de radio {r} es {area}.')  
    return area  
  
area_circulo_desafortunado(2)
```


Esta función tiene **dos** responsabilidades: calcular el área de un círculo e imprimir su valor por pantalla.



Principios de responsabilidad única y de generalidad

A la hora de programar funciones, se debe buscar que éstas sean lo más generales posible, de forma que puedan ser reutilizadas en diferentes circunstancias. La función anterior no es muy flexible, puesto que siempre imprime en pantalla el resultado y, probablemente, no en todas las ocasiones ese es el comportamiento que se desea.

Por tanto, desde otro punto de vista, violaría un **principio de generalidad**: no en todos los casos en que se quiera calcular el área del círculo, se desea imprimir el resultado por pantalla. Más bien lo contrario.




HASTA AQUÍ



Mutabilidad en Python

Antes de ver el apartado de **paso de parámetros** recordemos que los diferentes tipos de Python u otros objetos en general, pueden ser clasificados atendiendo a su mutabilidad:


- **Mutable**s: Si permiten ser modificados una vez creados.
 - **Inmutable**s: Si no permiten ser modificados una vez creado
- 

Mutabilidad en Python

Son **mutables** los siguientes tipos:

- Listas
- Diccionarios
- Sets
- Clases definidas por el usuario


Y son **inmutables**:

- Booleanos
 - Enteros
 - Float
 - Cadenas
 - Tuplas
 - Range
- 

Mutabilidad en Python

Recordada esta clasificación, tal vez te preguntes porqué es esto relevante. Pues bien, Python trata de manera diferente a los tipos mutables e inmutables, y si no entiendes bien este concepto, puedes llegar a tener comportamientos inesperados en tus programas.

La mutabilidad de los objetos es una característica muy importante cuando tratamos con funciones, ya que Python los tratará de manera distinta.



Mutabilidad en Python

En muchos lenguajes de programación existen los conceptos de paso por **valor** y por **referencia** que aplican a la hora de como trata una función a los parámetros que se le pasan como entrada. Su comportamiento es el siguiente:

- Si usamos un parámetro pasado por **valor**, se creará una copia local de la variable, lo que implica que cualquier modificación sobre la misma no tendrá efecto sobre la original.
- Con una variable pasada como **referencia**, se actuará directamente sobre la variable pasada, por lo que las modificaciones afectarán a la variable original. Lo que realmente hace es copiar en los parámetros la dirección de memoria de las variables que se usan como argumento. Esto implica que realmente hagan referencia al mismo elemento y cualquier modificación del valor en el parámetro afectará a la variable externa correspondiente.

Los tipos inmutables son **pasados por valor** y los tipos mutables son **pasados por referencia**.

Función. Paso de parámetro por valor o por referencia

Las variables en Python son identificadores que referencian a objetos almacenados en bloques de memoria. Si se pasa un valor de un objeto inmutable, su valor no se podrá cambiar dentro de la función. Sin embargo si pasamos un objeto de un tipo mutable podremos cambiar su valor:

Tenemos una función que modifica dos variables:

```
#tenemos una función que modifica dos parámetros
def mi_funcion(a, b):
    a = 10
    b[0] = 10

# x es un entero
x = 5
# y es una lista
y = [1,2,3]
```

Si llamamos a la función con ambas variables, vemos como el valor de **x** no ha cambiado, pero el de **y** sí.

```
mi_funcion(x, y)

print(x)
print(y)
```

El resultado será

```
5
[10, 2, 3]
```


Función. Paso de parámetro por valor o por referencia

Otro ejemplo:

```
def mi_funcion(mi_lista):
    mi_lista.append(5)
    return mi_lista

lista = [1,2]
nueva_lista = mi_funcion(lista)
print(lista)
print(nueva_lista)
```

La ejecución de este código da por resultado la lista pasada como argumento modificada

```
[1, 2, 5]
[1, 2, 5]
```

En general, se espera que una función que recibe parámetros mutables, no los modifique, ya que si se los modifica se podría perder información valiosa. En el caso en que por una decisión de diseño o especificación se modifiquen los parámetros recibidos, esto debe estar claramente documentado.

Función. Paso de parámetro por valor o por referencia

Para evitar que una función modifique una lista puedes enviar una copia de la lista a la función.

```
def mi_funcion(mi_lista):
    mi_lista.append(5)
    return mi_lista

lista = [1,2]
nueva_lista = mi_funcion(lista[:])
print(lista)
print(nueva_lista)
```

La ejecución de este código da por resultado la lista pasada como argumento sin modificar

```
[1, 2]
[1, 2, 5]
```

Función. if `__name__ == "__main__"`

Es común encontrarnos código con esta forma:

```
def hacer_algo():  
    print("algo")  
  
if __name__ == "__main__":  
    hacer_algo()
```

En lugar de, por ejemplo:

```
def hacer_algo():  
    print("algo")  
  
hacer_algo()
```

Función. if `__name__ == "__main__"`

Cuando un intérprete de Python lee un archivo de Python, primero establece algunas variables especiales. Luego ejecuta el código desde el archivo.

Una de esas variables se llama `__name__`

Los archivos de Python se llaman módulos y se identifican mediante la extensión de archivo `.py`. Un módulo puede definir funciones, clases y variables.

Función. if `__name__ == "__main__"`

Entonces, cuando el intérprete ejecuta un programa, el variable `__name__` se establecerá como `__main__` si el módulo que se está ejecutando es el programa principal.

La variable `__name__` para el archivo/módulo que se ejecuta será siempre `__main__`. Pero la variable `__name__` para todos los demás archivos/módulos que se importan se establecerá en el nombre de su módulo.

Función. if `__name__ == "__main__"`

La forma habitual de usar `__name__` y `__main__` se ve así:

```
def media(x,y):
    resultado = (x+y)/2
    return resultado

if __name__ == "__main__":
    a = 2
    b = 3
    m = media(a,b)
    print(m)
```

Explicación completa en el este [enlace](#)

Estructura de un programa

Puedes estructurar tu programa de cualquiera de estas dos formas

```
def media(x,y):
    resultado = (x+y)/2
    return resultado

if __name__ == "__main__":
    a = 2
    b = 3
    m = media(a,b)
    print(m)
```

```
def media(x,y):
    resultado = (x+y)/2
    return resultado


#programa principal
a = 2
b = 3
m = media(a,b)
print(m)
```

Programación estructurada

La **programación estructurada** es un paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de ordenador, utilizando únicamente **subrutinas** (funciones o procedimientos) y tres estructuras: **secuenciales**, **alternativas** y **repetitivas**.

Programación modular

La **programación modular** es un paradigma de programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable.



Programación modular

Al aplicar la **programación modular**, un problema complejo debe ser dividido en varios subproblemas más simples, y estos a su vez en otros subproblemas más simples. Esto debe hacerse hasta obtener subproblemas lo suficientemente simples como para poder ser resueltos fácilmente con algún lenguaje de programación (divide y vencerás).

La programación estructurada y modular se lleva a cabo en python con la definición de funciones.



Funciones. Recursividad

Una **función recursiva** es una función que en su cuerpo contiene una llamada a sí misma.

La recursión es una práctica común en la mayoría de los lenguajes de programación ya que permite resolver las tareas recursivas de manera más natural.

Para garantizar el final de una función recursiva, las sucesivas llamadas tienen que reducir el grado de complejidad del problema, hasta que este pueda resolverse directamente sin necesidad de volver a llamar a la función.

Funciones. Recursividad

Vamos a hacer ahora un programa para calcular un concepto matemático que se llama factorial y que se usa mucho en cálculos de Combinatoria.

Para calcular $N!$, el factorial de N , tenemos que calcular el producto de todos los números enteros desde 1 hasta N .

Por ejemplo: el factorial de 5, que se representa con $5!$ será $5! = 5 * 4 * 3 * 2 * 1$ que valdrá 120.

Vamos a hacer una función que lo calcule usando un variable *resultado* para ir calculando el producto y un bucle *for* que iteraremos entre 1 y N

Funciones. Recursividad

```
def factorial(N):
    resultado = 1 # iremos guardando el resultado parcial
    for numero in range(N,1,-1): # vamos desde N hasta 1
        resultado *= numero
    return resultado
```

Si lo probamos

```
>>> factorial(5)
>>> 120
```

Funciones. Recursividad

Si le damos un poco de vueltas al concepto, veremos que $5! = 5 * 4!$, es decir, podemos calcular el factorial de N en función del factorial de N-1, y este a su vez en función del factorial de N-2, y así sucesivamente hasta 1! que sabemos que es 1.

Sería como desde dentro de la función factorial, volver a llamar a la función factorial pero con el número anterior como argumento y así hasta que al llegar al llamarlo con el argumento 1 devolveremos el valor 1.

El cálculo del factorial es más simple de implementar de forma recursiva

<pre>def factorial(n): if n == 0: return 1 else: return n * factorial(n-1) print(factorial(5))</pre>	<pre>def factorial(N): resultado = 1 # iremos guardando el resultado parcial for numero in range(N,1,-1): # vamos desde N hasta 1 resultado *= numero return resultado</pre>
---	--

Funciones. Recursividad

A las funciones que se llaman a sí mismas de esta manera las denominamos **funciones recursiva**. Una función recursiva puede invocarse a sí misma tantas veces como quiera en su cuerpo.

```
def fibonacci(n):  
    if n <= 1:  
        return n  
    else:  
        return fibonacci(n - 1) + fibonacci(n - 2)  
  
print(fibonacci(6))
```

La ejecución de este código produce el resultado **8**

Referencias web

Fundamentos de programación en Python. Universidad de Valladolid

https://www2.eii.uva.es/fund_inf/python/notebooks/08_Funciones/Funciones.html