# 1. FUNCIONES.

¿*FUNCIONES*?

Una función es un bloque de código que solo se ejecuta cuando se llama/invoca.

El código de la función puede realizar cualquier tipo de acción `(algoritmo)`, pero es aconsejable que realice acciones concretas, lo más genéricas posible, con el fin de ser reutilizable.

+ A la función se pueden pasar datos `(argumentos-parámetros reales)` que son recogidos por variables (parámetros`-formales)`.
+ Y la función puede devolver datos como resultado.

## 1.1. TIPOS DE FUNCIONES EN PYTHON.

En Python, encontramos distintos tipos, entre ellos, según:

- Según quien las declare "creador":
  + Funciones propias/métodos:
    - Incorporadas.
    - En módulos.
    - Métodos de los objetos.
  + Funciones del programador:
    - En módulos.
- Según formato declaración:
  +Funciones con nombre:
    - Declaración en varias líneas
    - Declaración en una línea.
  +Funciones sin nombre/anónimas: **`lambda`** en una sola línea.
  +Funciones asignadas a una variable.
- Según donde se declaren:
  + Funciones **independientes**: dentro de los módulos.
  + Funciones **anidadas**, se declaran dentro del código de otra función.
- Según desde se "llame/invoque" a una función":
  + Desde una línea del código del módulo.
  + Desde otra función:
    Y según desde que parte de la función llamada:
    - Desde el bloque de código de una función:
      - Entre funciones independientes.
      - Entre funciones anidadas/ en el mismo "nido"
      - Desde "fuera" (modulo/ o función independiente) a funciones anidadas.
    - Llamada dentro de un argumento de otra función.
    - Llamada dentro de **return** de una función.

    **Funciones de orden superior** Son aquellas funciones que aceptan como parámetro una función y retornan otra función.
      - Entre ellas funciones: map, filter, etc (otras en módulos).
      - Y las funciones **decoradoras**.
  + Desde el código de la misma función, a la que llama (**Recursividad**).
- Según nos aseguremos que existe una función, antes de invocarla.
  + Llamada "dinámica" o **Llamadas de retorno**.

# 2. DECLARACIÓN DE FUNCIONES DEL PROGRAMADOR.

En Python, encontramos distintos formatos para "declarar" funciones.

## 2.1. "FUNCIONES CON NOMBRE":

Según **PEP 8**:

+ A la definición de una función la deben anteceder **dos líneas en blanco**.
+ La definición de función se realiza mediante la orden def / *def* . seguida del nombre de función, y la lista de parámetros formales entre paréntesis. Finalizando con los dos puntos (inicio bloque).
+ Los parámetros deberán ir separados por comas.
+ Los parámetros por omisión, **no debe** dejarse espacios en blanco ni antes ni después del signo =.
+ Las sentencias que forman el cuerpo de la función empiezan en la línea siguiente, y deben estar con sangría.

¿*FORMATOS*?

**A. En varias líneas**

```
def nombre_funcion ( [parámetros] ) :
        [Cadena documentación de la función -docstring-]
        Código función . . .
        [return valorRetorno]
```

E410-aDec(FNombre).py

**B. En una línea:**

```
def nombre_funcion ( [parámetros] ) : [return] Código_en_una_línea
```

E410-bDec(FNombre).py

nombre_funcion ---- Descriptivo, igual reglas que para el nombre de las variables.

parámetros --------- (parámetros formales). Se indican entre paréntesis. Son las **variables** de **ámbito local**, que recogen los valores enviados por los argumentos (parámetros reales), al ser llamada la función.
+ Una función puede tener de **0** (ninguno) a **N** parámetros (que irán separados por una coma).
+ Al ser variables locales, no pueden ser accedidas fuera de la función.

docstring -------- Comentario como primera sentencia del cuerpo de la función, que será tratado como texto de documentación de la función ( **docstring)**.
+ Que podrá visualizarse por medio de la variable: **__doc__**.
Nota: solo visualiza si es la primera línea del bloque de código.

+ Es una buena práctica incluir **docstrings** en el código que uno escribe, por lo que se debe hacer un hábito de esto.

**return** ----------La orden return /*return* .Se utiliza para salir de una función y devolver "un valor", pueden ser varios valores si es un dato compuesto.
+ Por omisión, cualquier función retorna el valor: **None**.
+ Pueden incluirse varias expresiones **return** en una función, pero sólo se ejecutará la primera que se encuentre.

## 2.2. "FUNCIONES SIN NOMBRE/ANÓNIMAS": (FUNCIÓN LAMBDA)

En Python, Las funciones **lambda** son funciones anónimas, sintácticamente restringidas a una sola expresión, una solo línea de código ("no confundir con una sola sentencia").

+ Se reconocen por la palabra reservada **lambda**.

**lambda** ----------La orden lambda /*lambda*, permite crear pequeñas funciones anónimas.

```
lambda ( [parámetros] ) : Código_en_una_línea
```

E410-cDec(FAnonimas).py

+Una función lambda puede tener de 0 a N parámetros, pero solo puede tener **una expresión**.

- Como en el caso de las funciones con nombre, puede tener o no parámetros, y si los tienen con las mismas situaciones (por defecto, con nombre, etc.).
- La expresión se evalúa y el resultado es el valor devuelto por la función lambda.
- La sentencia **return**, esta "implícita" en una función lambda.

+ La expresión puede contener cualquier tipo de expresión.

-La expresión puede incluir a su vez funciones (incluida la lambda -recursividad), y if-else de una línea.

- Al igual que las funciones anidadas, las funciones **lambda** pueden **hacer referencia a variables desde el ámbito que la contiene.**

+ Para **nombrar** a estas funciones se utiliza el operador de asignación ( = ).

¿*DÓNDE, CUÁNDO UTILIZAR*?

+ Puede aparecer, ser **usadas en cualquier "lugar"** donde sea requerido un objeto de tipo función.

+Suelen ser usadas cuando necesitamos una función una sola vez.

+ Normalmente se crean funciones lambda con el único propósito de pasarlas a funciones de **orden superio**r.

+El poder de lambda se muestra mejor cuando se usan como una **función anónima dentro de otra función.**

### ● Asignando a una variable":

♫**Otros autores: Funciones generadas a partir de otras.**

```
variable = [ nombre_funcion | declaración_lambda ]
```

+ La variable será de tipo **<class 'function>** , y apunta al mismo objeto (tienen igual **id**).

+ Ambas funciones pueden utilizarse de igual forma.

### ● Anidadas y/o "enlazadas":

Python permite definir funciones dentro de otras funciones.

Igualmente, permite llamar a una función dentro de otra, de forma fija y de la misma manera que se la llamaría, desde fuera de dicha función.

¿*QUÉ SUCEDE AL DECLARAR UNA FUNCIÓN*?

La **definición** o declaración **de una función** introduce el nuevo nombre de la función en la **tabla de símbolos actual**.

¿*TABLA DE SÍMBOLOS*?

Una tabla de símbolos contiene la información necesaria sobre el programa actual.

Se puede conocer con las funciones incorporadas: globals(), locals() y vars().

¿*EXISTENCIA FUNCIÓN Y POSIBILIDAD DE SER LLAMADA*?

Si una función existe, su nombre debe haber sido incorporado a la tabla de símbolos.

+ El operador **in**, nos permitirá conocer si un elemento se encuentra dentro de una colección.

Una vez comprobada su existencia, con la función callable() sabremos si esa función puede ser llamada.

## 3. LLAMADA Y EJECUCIÓN DE UNA FUNCIÓN.

Para llamar a una función, use el nombre de la función seguido de paréntesis.

```
nom_funcion ( [argumentos] )
```

¿*QUÉ SUCEDE AL INVOCAR A UNA FUNCIÓN*?

**La llamada y posterior ejecución de una función** genera una **nueva tabla de símbolos** para las variables **locales** de la función.

+ Los **parámetros reales (argumentos)** de una función se introducen en la **tabla de símbolos local** de la función llamada cuando esta es ejecutada; así, los argumentos son pasados por valor (dónde el valor es siempre una referencia a un objeto, no el valor del objeto).

+ Cuando una función llama a otra función**, una nueva tabla** de símbolos local es creada para esa llamada.

## 4. ARGUMENTOS Y PARÁMETROS.

Python permite varios formatos de relación entre argumentos ←→parámetros, y pueden darse juntos.

¿*CORRESPONDENCIA FIJA DE ARGUMENTOS Y PARÁMETROS*?

### 4.1. POR POSICIÓN (NÚMERO FIJO DE PARÁMETROS Y ARGUMENTOS):

`E411-Lla1(XPos_y_Omi).py`

Al llamar a una función, **por posición/fijos**, siempre se le deben **pasar sus argumentos en el mismo orden en el que los espera**. Y el "**mismo número**" entre argumentos y parámetros. Si alguno falta habrá una **excepción**.

¿*PARÁMETROS POR DEFECTO*? O Argumentos por omisión.

En Python, también es posible, asignar valores por defecto a los parámetros de las funciones.

- El valor por omisión, **solo será asignado** si no se ha enviado como argumento.
- Esto significa, que la función podrá ser llamada con menos argumentos de los que espera.
- Nota: Colocar al final los parámetros con valores por defecto, no pueden ir mezclados.

¿*CÓMO INVOCAR UNA FUNCIÓN CON PARÁMETROS POR DEFECTO*?

La función puede ser llamada de distintas maneras:

- Pasando solo los argumentos obligatorios.
- Pasando todos, alguno o ninguno de los parámetros opcionales.

¿*CÓMO FUNCIONARA*?

Los valores por omisión son evaluados en el momento de la **definición de la función**, en el **ámbito** de la definición, es decir, el valor por omisión es **evaluado solo una vez**.

- Cuidado si el valor por omisión es una variable, pues:
  Si es una variable de la función dará error.
  y Si es una variable fuera de su ámbito, debe estar definida **antes**.

### 4.2. POR "NOMBRE" (NÚMERO FIJO DE PARÁMETROS Y ARGUMENTOS):

`E411-Lla2(XNombre).py`

Otro método utilizado para el paso de argumentos a parámetros es por el NOMBRE.

+ En este caso se pasa junto al nombre del parámetro su valor (`nom_parametro=valor`).
+ La posición, ahora, de los argumentos no tiene por qué ser igual a la ocupada por los parámetros.

### 4.3. ARBITRARIOS (NÚMERO DESCONOCIDO DE PARÁMETROS Y ARGUMENTOS):

Al igual que en otros lenguajes de alto nivel, es posible que una función, **espere recibir un número desconocido de argumentos**.

¿*SI HAY MÁS ARGUMENTOS QUE PARÁMETROS*? Argumentos arbitrarios.

- Si no sabemos el **número de argumentos** que podemos recibir, anteceder al parámetro:

`E411-Lla4(Arbitrarios).py`

+ **un asterisco (*)**: los argumentos, serán recogidos en ese parámetro como **tupla**.
+ **dos asteriscos** (**): los argumentos, serán recogidos en la función como un **diccionario**.
  - Cuando los argumentos enviados tienen formato como pares de clave=valor.

¿*Y SI MEZCLO PARÁMETROS ARBITRARIOS (TUPLA, DICT) Y OTROS*?

Hay que respetar las siguientes reglas:

- Si una función espera recibir parámetros fijos y arbitrarios, **los arbitrarios siempre deben suceder a los fijos**.
- Los arbitrarios deben ser primero los de tupla y después de diccionario.

¿*SI HAY MÁS PARÁMETROS QUE ARGUMENTOS*? Argumentos empaquetados.

● Si los argumentos están en formato "empaquetado" (tupla, lista, diccionario,..) y existen parámetros para recibirlos "desempaquetados":

`E411-Lla5(Empaquetados)`

Es decir, que la función **espere una lista fija de parámetros**, **pero** que éstos, en vez de estar disponibles de forma separada, **se encuentren contenidos en una lista o tupla**.

Para "desempaquetar" los argumento, anteceder al argumento:

+ **un asterisco (*)**: los argumentos, serán "desempaquetados" antes de ser pasados a los parámetros.
  -En este caso, el signo asterisco (`*`) deberá preceder al nombre de la lista o tupla que es pasada como parámetro durante la llamada a la función:
+ **dos asteriscos** (**): los argumentos, serán "desempaquetados" antes de ser pasados a los parámetros.
  - El mismo caso puede darse cuando los valores a ser pasados como parámetros a una función, se **encuentren disponibles en un diccionario**. Aquí, deberán pasarse a la función, precedidos de dos asteriscos (`**`):

### 4.4. MEZCLANDO FORMATOS:

**En una llamada a una función**, los argumentos nombrados deben seguir a los argumentos posicionales.

● Si una función espera recibir parámetros fijos y arbitrarios, **los arbitrarios siempre deben suceder a los fijos**.

**Ningún argumento puede recibir más de un valor al mismo tiempo**.

## 5. ÁMBITO DE LAS VARIABLES.

+ Las variables locales definidas en el código que llama a una función no son visibles dentro de la función llamada.
+ Las variables "declaradas" dentro de la función son de ámbito local a la función.
+ Los argumentos `(parámetros actuales)` se pasan a las funciones por **valor -copia**, no por referencia.
  + La función obtiene una **copia local** de la pasado parámetros, de esta forma solo se modifica la copia.
+ Cuando una función finaliza: las variables locales definidas dentro de la función se *pierden*.
+ Se pueden definir **variables globales** prefijando su definición con la palabra reservada global / *global*., pero **debe evitarse** siempre que sea posible.
+ La palabra reservada nonlocal /----. Permite declarar una variable no local, dentro de funciones anidadas, donde la variable no debe pertenecer a la función interna.