UD 03 – Modularidad y Funciones

La modularidad es la característica de un sistema que permite que sea estudiado, visto o entendido como la unión de varias partes que interactúan entre sí y que trabajan solidariamente para alcanzar un objetivo común, realizando cada una de ellas una tarea necesaria para la consecución de dicho objetivo.

Cada una de esas partes en que se encuentre dividido el sistema recibe el nombre de módulo. Idealmente un módulo debe poder cumplir las condiciones de caja negra, es decir, ser independiente del resto de los módulos y comunicarse con ellos (con todos o sólo con una parte) a través de entradas y salidas bien definidas

Funciones:

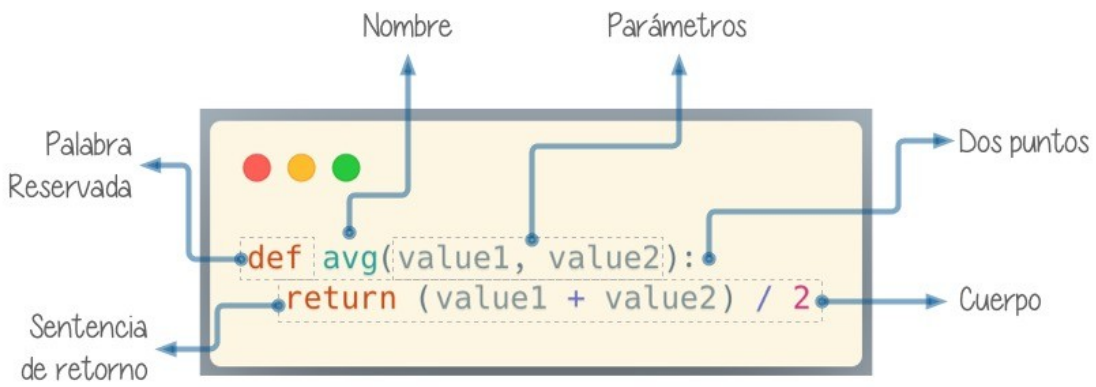
El concepto de función es básico en prácticamente cualquier lenguaje de programación. Se trata de una estructura que nos permite agrupar código. Persigue dos objetivos claros:

- No repetir fragmentos de código en un programa.
- Reutilizar el código en distintos escenarios.

Una función viene definida por su nombre, sus parámetros y su valor de retorno. Esta parametrización de las funciones las convierten en una poderosa herramienta ajustable a las circunstancias que tengamos. Al invocarla estaremos solicitando su ejecución y obtendremos unos resultados.

Definir una función

Para definir una función utilizamos la palabra reservada def seguida del nombre 6 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`.



Hagamos una primera función sencilla que no recibe parámetros:

```
def say_hello():
    print('Hello!')
```

Los nombres de las funciones siguen las mismas reglas que las variables y, como norma general, se suelen utilizar **verbos en infinitivo** para su definición: `load_data`, `store_values`, `reset_cart`, `filter_results`, `block_request`, …

Invocar una función

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 say_hello():

    print('Hello!')


say_hello()
```

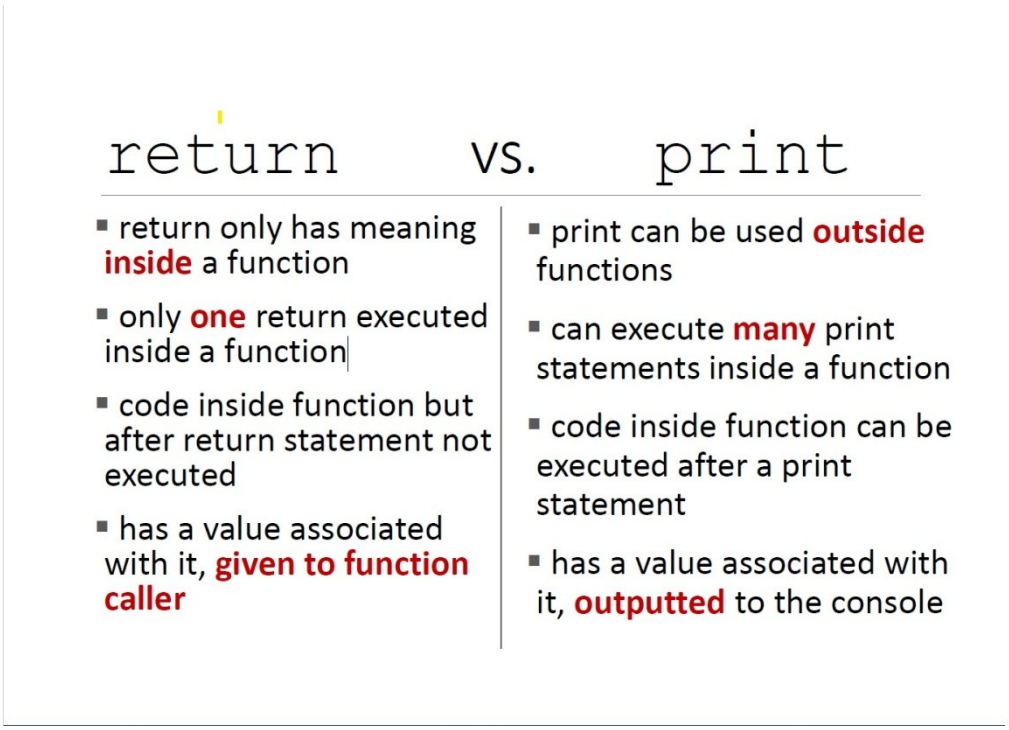
Retornar un valor

Las funciones pueden retornar (o «devolver») un valor. Veamos un ejemplo muy sencillo

```
def one():
    return 1

one()
```

No confundir return con print(). El valor de retorno de una función nos permite usarlo fuera de su contexto. El hecho de añadir print() al cuerpo de una función es algo «coyuntural» y no modifica el resultado de la lógica interna.



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

```
if one() == 1:
    print('It works!')
else:
    print('Something is broken')
```

Parámetros y argumentos

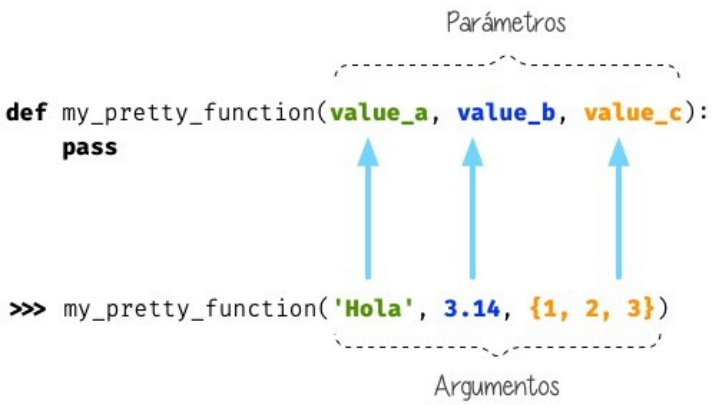
Si una función no dispusiera de valores de entrada estaría muy limitada en su actuación. Es por ello que los parámetros nos permiten variar los datos que consume una función para obtener distintos resultados. Vamos a empezar a crear funciones que reciben parámetros.

En este caso escribiremos una función que recibe un valor numérico y devuelve su raíz cuadrada

```
def sqrt(value):
    return value ** (1/2)


sqrt(4)
```

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



Veamos otra función con dos parámetros y algo más de lógica de negocio:

```
def _min(a, b):
    if a < b:
        return a

    else:
        return b


_min(7, 9)
```

Nótese que la sentencia return puede escribirse en múltiples ocasiones y puede encontrarse en cualquier lugar de la función, no necesariamente al final del cuerpo. Esta técnica puede ser beneficiosa en múltiples escenarios.

Uno de esos escenarios se relaciona con el concepto de cláusula guarda: una pieza de código que normalmente está al comienzo de la función y comprueba una serie de condiciones para continuar con la ejecución o cortarla

Teniendo en cuenta que la sentencia return finaliza la ejecución de una función, es viable eliminar la sentencia else del ejemplo visto anteriormente:

```
def _min(a, b):
    if a < b:
        return a
    return b

_min(7, 9)
```

Documentación

Ya hemos visto que en Python podemos incluir comentarios para explicar mejor determinadas zonas de nuestro código.

Del mismo modo podemos (y en muchos casos debemos) adjuntar documentación a la definición de una función incluyendo una cadena de texto (docstring) al comienzo de su cuerpo

```
def sqrt(value):
    'Returns the square root of the value'
    return value ** (1/2)
```

La forma más normal de escribir un docstring es utilizando triples comillas:

```
def closest_int(value):
    '''Returns the closest integer to the given value.
    The operation is:
        1. Compute distance to floor.
        2. If distance less than a half, return floor.
           Otherwise, return ceil.
    '''
    floor = int(value)
    if value - floor < 0.5:
        return floor
    else:
        return floor + 1
```

Funciones recursivas

La recursividad es el mecanismo por el cual una función se llama a sí misma:

Veamos ahora un ejemplo más real en el que computar el enésimo término de la Sucesión de Fibonacci utilizando una función recursiva:

```
def fibonacci(n: int) -> int:
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

```
fibonacci(10)
fibonacci(20)
```

Ejercicio: Calcula el factorial de un numero, pero esta vez de forma recursiva

Consejos para programar

Chris Staudinger comparte estos 7 consejos para mejorar tu código:

1. Las funciones deberían hacer una única cosa.

   Por ejemplo, un mal diseño sería tener una única función que calcule el total de una cesta de la compra, los impuestos y los gastos de envío. Sin embargo esto se debería hacer con tres funciones separadas. Así conseguimos que el código sea más fácil de matener, reutilizar y depurar.

2. Utiliza nombres descriptivos y con significado.

   Los nombres autoexplicativos de variables y funciones mejoran la legibilidad del código. Por ejemplo – deberíamos llamar «total_cost» a una variable que se usa para almacenar el total de un carrito de la compra en vez de «x» ya que claramente explica su propósito.

3. No uses variables globales.

   Las variables globales pueden introducir muchos problemas, incluyendo efectos colaterales inesperados y errores de programación difíciles de trazar. Supongamos que tenemos dos funciones que comparten una variable global. Si una función cambia su valor la otra función podría no funcionar como se espera.

4. Refactorizar regularmente.

   El código inevitablemente cambia con el tiempo, lo que puede derivar en partes obsoletas, redundantes o desorganizadas. Trata de mantener la calidad del código revisando y refactorizando aquellas zonas que se editan.

5. No utilices «números mágicos» o valores «hard-codeados».

   *No es lo mismo escribir «99 * 3» que «price * quantity». Esto último es más fácil de entender y usa variables con nombres descriptivos haciéndolo autoexplicativo. Trata de usar constantes o variables en vez de valores «hard-codeados».*

6. Escribe lo que necesites ahora, no lo que pienses que podrías necesitar en el futuro.

   Los programas simples y centrados en el problema son más flexibles y menos complejos.

7. Usa comentarios para explicar el «por qué» y no el «qué».

El código limpio es autoexplicativo y por lo tanto los comentarios no deberían usarse para explicar lo que hace el código. En cambio, los comentarios debería usarse para proporcionar contexto adicional, como por qué el código está diseñado de una cierta manera.

https://docs.python.org/es/3/library/index.html