Fundamentos de Programación. Primer Curso de ASIR.

UT05.01 – Funciones. Conceptos Básicos.

UT05.01 – Funciones. Conceptos Básicos.
1.- Introducción.

- Hasta ahora hemos usado las funciones y métodos sin saber realmente lo que eran. Tan solo nos proporcionaban una utilidad y no nos hemos preguntado nada mas.
- En la programación estructurada se introduce el concepto de **modularidad**.

  La modularidad consiste básicamente en dividir el código en partes con una misión y **asignarles un nombre**.
- Dependiendo de sus características, estos módulos reciben el nombre de funciones, módulos, paquetes, métodos, etc.
- Aquí vamos a ver las características y uso de las **funciones** y aprenderemos a construir las nuestras propias.

UT05.01 – Funciones. Conceptos Básicos.
2.- Definición de una función.
2.1.- Nomenclatura.

- **En programación, una función es una secuencia de sentencias que realizan una operación y que reciben un nombre**.
- Una vez definida, se puede **“llamar”** o **“invocar”** por dicho nombre.
- **Ejemplo**. En la siguiente llamada:

  >>> int(“32”)

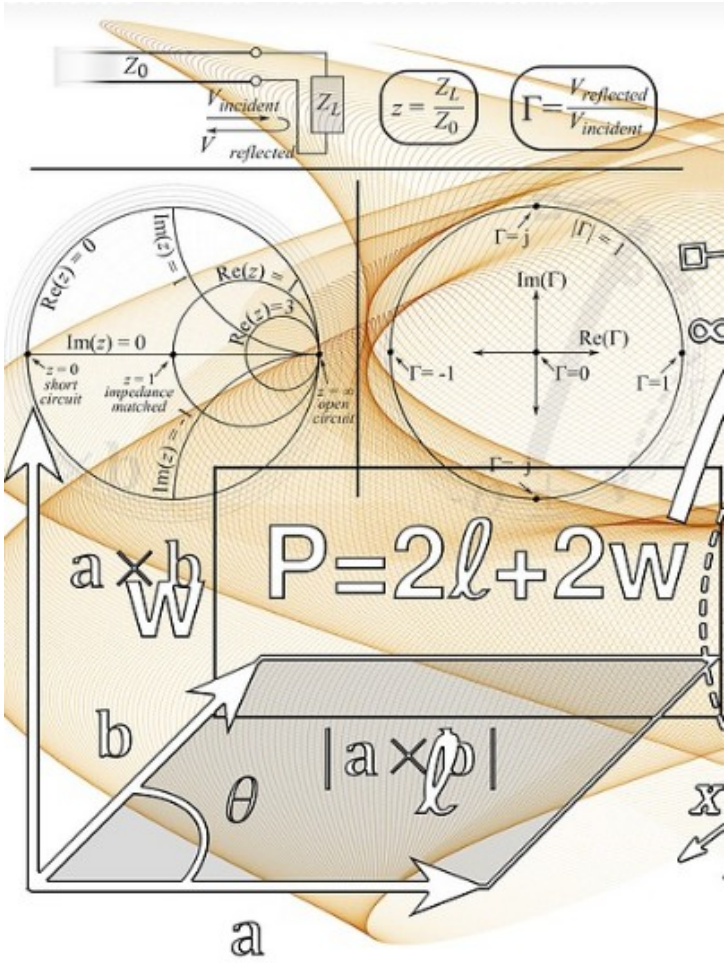
  32

  - El nombre de la función es int.
  - La expresión entre paréntesis recibe el nombre de **parámetro** de la función. En este caso es una cadena de tipo String. **Una función puede ser invocada con un número indeterminado de argumentos, incluso sin argumentos**.
  - El resultado de la función int es la conversión, si es posible, de esa cadena a entero.
- **En la nomenclatura de programación se dice que una función toma o recibe una serie de parámetros y retorna o devuelve un resultado.**

UT05.01 – Funciones. Conceptos Básicos.
2.- Definición de una función.
2.2.- Funciones propias del lenguaje y de usuario.

- Los lenguajes de programación tienen una serie de funciones ya predefinidas. Estas funciones vienen dadas por el estándar o bien son diseñadas por los creadores de una herramienta en particular para dicho lenguaje.
- En décadas anteriores cada entorno de desarrollo (IDE) disponía de su propio conjunto de funciones. Eso imponía una extrema rigidez a los programadores.

  En la actualidad se intenta que dichas funciones sean consensuadas mediante un estándar
- Hay dos tipos de funciones predefinidas en un lenguaje de programación:
  - **Funciones incrustadas en el lenguaje (built-in)**. Estas funciones vienen ya definidas como palabras clave del lenguaje. **Estas funciones son de uso muy común**.
  - **Funciones incluidas en paquetes**. Son funciones que se usan menos frecuentemente. Estas organizadas por **paquetes** que engloban conjuntos de funciones con una utilidad determinada (matemáticas, cadenas, listas, etc).

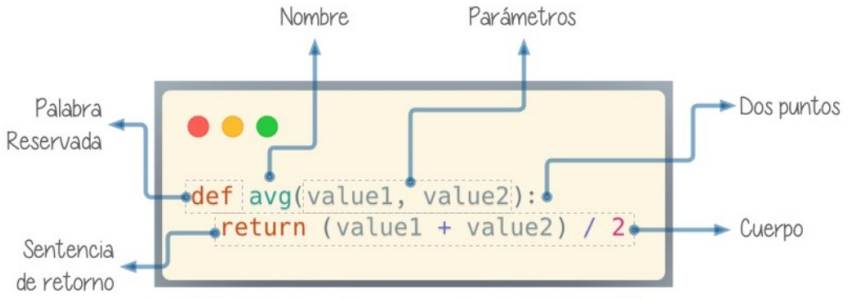


UT05.01 – Funciones. Conceptos Básicos.
2.- Definición de una función.
2.2.- Funciones definidas por el programador (I). Definición.

- Los lenguajes de programación permiten al programador desarrollar funciones propias.

Definición de una función.

- El formato básico de definición de una función es el siguiente:



- Como puede verse:
  - La definición empieza con la palabra **def**.
  - El código va tabulado con respecto a la linea de definición.
  - La última sentencia es la que devuelve el resultado.

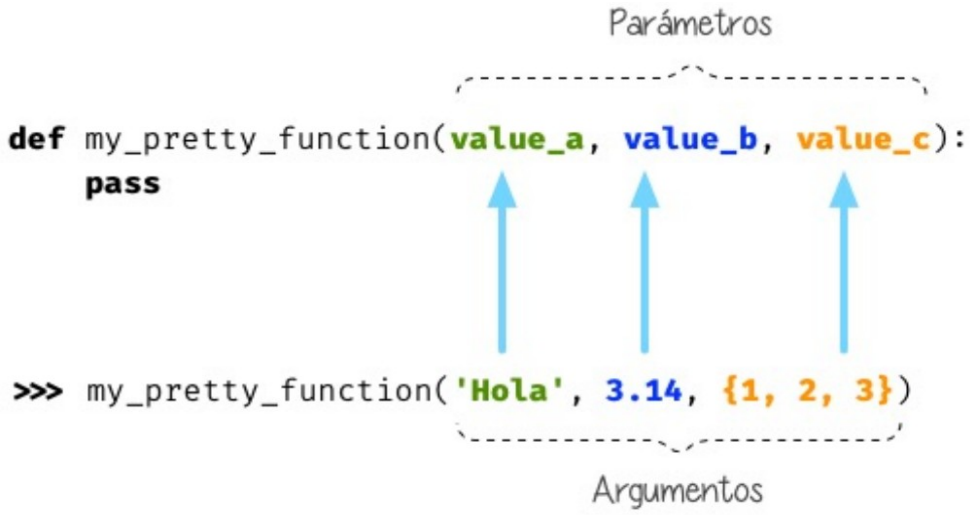
```
>>> def saluda(nombre):
...     print(f"¡Hola {nombre}!.")
...
>>> saluda("Pedro")
¡Hola Pedro!.
>>> saluda("María José")
¡Hola María José!.

>>> def hipotenusa(cateto1,cateto2):
...     resultado = cateto1**2 + cateto2**2
...     resultado = resultado**0.5
...     return resultado

>>> hipotenusa(2,3)
3.605551275463989
>>> hipotenusa(3,4)
5.0
```

UT05.01 – Funciones. Conceptos Básicos.
2.- Definición de una función.
2.2.- Funciones definidas por el programador (II). Argumentos.

- **<u>Importante.</u> La función debe estar definida antes de ser invocada y su nombre no puede coincidir con el de una variable.**
- Cuando una función es invocada a los parámetros se les asigna un valor en particular. A esos valores se les denomina **argumentos**.



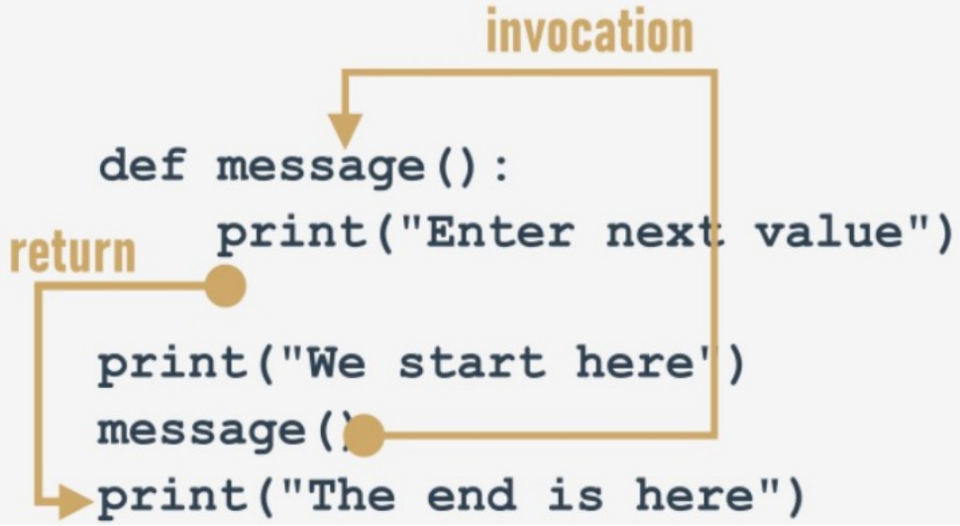

```
print("Inicio de programa.")
hola()
print("Programa terminado.")

def hola():
  print("¡Hola!")

Inicio de programa.
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'hola' is not defined
```

UT05.01 – Funciones. Conceptos Básicos.
2.- Definición de una función.
2.2.- Funciones definidas por el programador (III). Flujo de programa.

- Cuando una función es invocada el flujo de programa se desvía al código de la función, siendo sustituidos los parámetros con los argumentos correspondientes.
- Si la función retorna un resultado este podrá ser utilizado en el código principal de diversas formas: asignándoselo a una variable, argumento de una función, etc.




```
print("Inicio de programa.")
hola()
print("Programa terminado.")

def hola():
  print("¡Hola!")

Inicio de programa.
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'hola' is not defined
```

UT05.01 – Funciones. Conceptos Básicos.
3.- Argumentos de llamada a una función.
3.1.- Argumentos posicionales.

```
>>> primero = 3
>>> segundo = 2
>>> def difCuadrados(a, b):
...     return a**2-b**2

>>> difCuadrados(primero, segundo)
5
>>> difCuadrados(segundo, primero)
-5

>>> difCuadrados(primero)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: difCuadrados() missing 1 required
positional argument: 'b'

>>> tercero = 6
>>> difCuadrados(primero, segundo, tercero)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: difCuadrados() takes 2 positional
arguments but 3 were given
```

- Hay varias formas de asignar mediante argumentos valores a los parámetros. La primera forma es los **argumentos posicionales**.
- Con los argumentos posicionales los valores se asignan en su correspondiente orden. Es decir, se realiza un mapeo directo entre argumentos y parámetros.
- Una clara desventaja del uso de argumentos posicionales es que **se necesita recordar el orden y el número de los parámetros de la función**.

  Un error en la posición de los argumentos puede causar resultados indeseados.

UT05.01 – Funciones. Conceptos Básicos.
3.- Argumentos de llamada a una función.
3.2.- Argumentos nominales. Combinación de ambos.

- Los argumentos no son copiados en un orden específico sino que se asignan por nombre a cada parámetro igualando dicho parámetro al valor del argumento.
- Esto permite no tener que conocer cuál es el orden de los parámetros en la definición de la función.
- Python permite **mezclar argumentos posicionales y nominales en la llamada a una función**.
- **Los argumentos posicionales siempre deben ir antes que los argumentos nominales**.

  De no hacerlo así, Python no tendría forma de discernir a qué parámetro corresponde cada argumento:

```
>>> def areaTriangulo(base, altura):
...     area = 0.5*base*altura
...     return area
...
>>> areaTriangulo(base=6, altura=3)
9.0
>>> areaTriangulo(altura=3, base=5)
7.5

>>> def printValores(a, b, c):
...   print(f"a={a}, b={b}, c={c}")

>>> printValores(3, c=2, b=7)
a=3, b=7, c=2
```

UT05.01 – Funciones. Conceptos Básicos.
3.- Argumentos de llamada a una función.
3.3.- Empaquetado de argumentos posicionales.

- Python nos ofrece la posibilidad de empaquetar y desempaquetar argumentos cuando estamos invocando a una función.
- Esta propiedad permite que podamos utilizar un número variable de argumentos en una función.
- **Si utilizamos el operador * delante del nombre de un parámetro posicional, estaremos indicando que los argumentos pasados a la función se empaqueten en una tupla**.
- Por convención a los parámetros empaquetados posicionalmente se les pone el nombre de **args**.
- Una posibilidad es **establecer en una función una serie de parámetros como requeridos y recibir el resto de argumentos como opcionales y empaquetados**.

```
>>> def sumatorio(*args):
...     suma = 0
...     for numero in args:
...         suma = suma + numero
...     return suma

>>> sumatorio(2,5,3)
10
>>> sumatorio(1,7,4,2,1)
15

>>> def printValores(inicio, final, *args):
...     suma=0
...     for numero in args:
...         suma = suma + numero
...     print(f"{inicio} {suma} {final}")

>>> printValores("Tenemos","acumulado", 2,3,4,1)
Tenemos 10 acumulado
```

UT05.01 – Funciones. Conceptos Básicos.
3.- Argumentos de llamada a una función.
3.3.- Empaquetado de argumentos nominales.

- Si utilizamos el operador **** delante del nombre de un parámetro nominal, **los argumentos pasados a la función se empaqueten en un diccionario**.

  Por lo tanto, el orden de los parámetros no importa. Los parámetros se asocian en función de las claves de dicho diccionario.
- Por convención se usa el nombre **kwargs** al parámetro nominal empaquetado.
- Existe la posibilidad de usar doble asterisco en la llamada a la función, para desempaquetar los argumentos nominales:

```
>>> def idiomas(nombre,**kargs):
...  print(f"La persona cuyo nombre es {nombre} tiene:")
...   for idioma in kargs.items():
...     lengua, nivel = idioma
...     print(f"En {lengua} tiene nivel {nivel}")

>>> idiomas("Cristina", ingles="Medio", frances="Bajo", portugues="Nativo")
La persona cuyo nombre es Cristina tiene:
En ingles tiene nivel Medio
En frances tiene nivel Bajo
En portugues tiene nivel Nativo
```

UT05.01 – Funciones. Conceptos Básicos.
4.- Parámetros por defecto.

```
>>> def printValores(a, b, c=0):
...   print(f"a={a}, b={b}, c={c}")
...
>>> printValores(2,4)
a=2, b=4, c=0
>>> printValores(2,4,6)
a=2, b=4, c=6
```

- Python permite especificar valores por defecto en los parámetros de una función.
- En el caso de que no se proporcione un valor al argumento, el parámetro correspondiente tomará el valor definido por defecto.
- Los valores por defecto en los parámetros se especifican cuando se define la función.

  Para ello, **se iguala el parámetro al valor por defecto**.

UT05.01 – Funciones. Conceptos Básicos.
5.- Resultado de una función.
5.1.- Funciones que no devuelven ningún valor.

```
>>> def mensajeInicio():
...     print(f"Hola. Bienvenido al programa.")

>>> mensajeInicio()
Hola. Bienvenido al programa.
```

- Son aquellas que **no tienen una clausula return en su código**.
- Se usan simplemente para mostrar mensajes o establecer el valor de un conjunto de varias en forma de proceso por lotes.

UT05.01 – Funciones. Conceptos Básicos.
5.- Resultado de una función.
5.2.- Funciones que devuelven valores.

- En estos casos existe una clausula return.
- En este caso, las funciones pueden devolver uno o varios valores:
  - **Un solo valor.** El valor devuelto por return es **un tipo de datos simple**.
  - **Varios valores**. Los valores se devuelven mediante un tipo de dato compuesto (lista, tupla, diccionario, etc).

```
>>> def salarioNeto(salarioBruto, irpf):
...     deduc = irpf + 6.75
...     neto = salarioBruto * (1 - deduc/100)
...     return neto

>>> salarioNeto(2000,15)
1565.0

>>> def vocalesEnFrase(frase):
...   vocales = {}
...   for letra in "AEIOU":
...     vocales[letra] = 0
...   for letra in frase.upper():
...     if letra in "AEIOU":
...       vocales[letra] = vocales[letra] + 1
...   return vocales

>>> vocales = vocalesEnFrase("Esto es una prueba")
>>> vocales
{'A': 2, 'E': 3, 'I': 0, 'O': 1, 'U': 2}
```

UT05.01 – Funciones. Conceptos Básicos. 6.- Documentación de una función (I).

- Documentar el código aporta información tanto a la persona que lo desarrollo como a otras que deseen realizar el mantenimiento de dicho código en sucesivas intervenciones.
- En este caso veremos como adjuntar documentación a la definición de una función incluyendo una cadena de texto de nominada **“docstring”** al comienzo de su cuerpo.
- **Para insertar un docstring en una función se inserta el texto entre tres comillas simples. El texto puede contener varias lineas**.

```
>>> def fibonacci(numero):
...     '''
...     Devuelve el valor de la función de Fibonacci para el número que se introduce como
...     argumento. La función de Fibonacci se calcula de la forma siguiente:
...
...       F(n) = F(n-1) + F(n-2) con F(0)=0 y F(1)=1.
...
...     '''
...     # Si numero es cero o es uno.
...     if numero==0:
...       resultado = 0
...     elif numero==1:
...       resultado = 1
```

UT05.01 – Funciones. Conceptos Básicos. 6.- Documentación de una función (II).

```
...    # En cualquier otro caso.
...    else:
...      indice, ultimo, penultimo = 2, 1 ,0
...      while indice<=numero:
...        resultado = ultimo + penultimo
...        ultimo, penultimo = resultado, ultimo
...        indice = indice + 1
...    return resultado
```

- Para ver el docstring de una función, basta con utilizar **help(funcion)**. Esto es aplicables a funciones propias y a funciones built-in del lenguaje y a las incluidas en módulos.

```
>>> help(fibonacci)

Help on function fibonacci in module __main__:

fibonacci(numero)

    Devuelve el valor de la función de Fibonacci para el número que se introduce como
    argumento. La función de Fibonacci se calcula de la forma siguiente:

            F(n) = F(n-1) + F(n-2) con F(0)=0 y F(1)=1.
(END)
```

UT05.01 – Funciones. Conceptos Básicos.
7.- Funciones lambda.

```
>>> difCuadrados = lambda x, y : x**2 - y**2

>>> difCuadrados(5,2)
21

>>> lista = [(1,4,1), (2,1,3), (0,0,5)]
>>> sorted(lista, key=lambda t : t[0])
[(0, 0, 5), (1, 4, 1), (2, 1, 3)]
>>> sorted(lista, key=lambda t : t[1])
[(0, 0, 5), (2, 1, 3), (1, 4, 1)]
>>> sorted(lista, key=lambda t : t[2])
[(1, 4, 1), (2, 1, 3), (0, 0, 5)]
```

- Una función **lambda** es una función pensada para ser definida en una sola linea.
- Tiene las siguientes propiedades:
  - No tiene nombre (**anónima**).
  - Su cuerpo tiene implícito un return.
  - Tiene cualquier número de parámetros.
- Una función lambda se define de la forma siguiente:

  nombre = lambda parámetros : re
- Las funciones lambda son bastante utilizadas como argumentos a otras funciones.

  Un ejemplo claro de ello es la función **sorted** que tiene un parámetro opcional key donde se define la clave de ordenación.

UT05.01 – Funciones. Conceptos Básicos. 8.- Funciones recursivas (I).

- La recursividad es el mecanismo por el cual **una función se llama a sí misma**.

  Existe un número máximo de llamadas recursivas. **Python controla esta situación por nosotros, ya que, de no ser así, podríamos llegar a consumir los recursos del sistema**.
- Para crear una función recursiva hay que delimitar dos tipos de casos:
  - **Caso Base**. Caso en el cual se conoce el valor directamente y no es necesario realizar recursivo.
  - **Amigo Imaginario**. Caso en el cual se desconoce el valor de la función y es necesario recurrir a la recursividad.
- **Ejemplo**. Supongamos que queremos diseñar una función que calcule el valor de la **sucesión de Fibonacci** para un determinado número. Los casos son los valores para cero y uno ya que se conoce su valor directamente. El amigo imaginario sería el encargado de calcular el resto de valores.

```
>>> def fibonacci(numero):
...     if numero==0:
...       return 0
...     elif numero==1:
...       return 1
...     else:
...       return fibonacci(numero-1) + fibonacci(numero-2)

>>> fibonacci(6)
8
```

UT05.01 – Funciones. Conceptos Básicos.
8.- Funciones recursivas (II).

- **Ejemplo**. Supongamos ahora deseamos calcular el valor del factorial de un número dado como argumento.

  El caso base es 0! es igual a cero.

  El amigo imaginario calcula el resto de los valores multiplicando en cada una de las recursiones el número que recibe como parámetro por el resultado de la la función llamada con el numero anterior hasta llegar a cero.

```
>>> def factorial(numero):
...     if numero==0:
...         return 1
...     else:
...         return numero * factorial(numero-1)

>>> for numero in range(6):
...         print(f"{numero}!={factorial(numero)}")

0!=1
1!=1
2!=2
3!=6
4!=24
5!=120
```

UT05.01 – Funciones. Conceptos Básicos.
9.- Ámbito de las variables.
9.1.- Tipos de ámbitos. Variables globales y locales.

```
>>> # Ambito Global.
>>> variable = 6
>>> def funcion(parametro):
...     print(f"parametro={parametro}")
...     print(f"variable={variable}")

>>> funcion(9)
parametro=9
variable=6

# Ambito Local.
>>> variable=6
>>> def funcion():
...     variable=9
...     print(f"variable={variable}")

>>> funcion()
variable=9
```

- Los ámbitos proporcionan un mecanismo de empaquetamiento, de tal forma que podamos tener incluso **nombres iguales que no hacen referencia al mismo objeto**.

Ámbito Global.

- Se define **ámbito global** como el que comprende todo el código de un programa, incluido las distintas funciones que contiene.

Ámbito local.

- Se define **ámbito local** el código de una función determinada.

Variables globales y variables locales.

- En este contexto tendremos **variables globales** y **variables locales**.

UT05.01 – Funciones. Conceptos Básicos.
9.- Ámbito de las variables.
9.2.- Forzando que una variable sea global.

```
>>> variable=6
>>> def funcion():
...   global variable
...   varible = 9
...   print(f"variable={variable}.")

>>> funcion()
variable=9.
>>> variable
9
```

- Python permite que una variable sea definida como global dentro de una función.
- Para ello debemos usar el modificador

  global variable

  Cuando una variable se define de esta forma y su valor es modificado dentro de la función, se modifica en el resto del programa.

Listado de variables locales y globales.

- Hay dos funciones que nos proporcionan sendos diccionarios con un registro de variables globales y locales. Estas son las siguientes.
  - **globals()**. Variables globales.
  - **locals()**. Variables locales.

UT05.01 – Funciones. Conceptos Básicos.
10.- Módulos en Python.
10.1.- Definición. Importación de un módulo.

- Un módulo es un **fichero que contiene definiciones y declaraciones de Python**. El nombre de archivo es el nombre del módulo con el sufijo .py agregado.

  Dentro de un módulo, el nombre del mismo módulo (como cadena) está disponible en el valor de la variable global **__name__**.
- Para hacer uso del código de un módulo usaremos la sentencia **import**. La sintaxis es la siguiente:

  import modulo

  Esto hace que el código y las variables de dicho módulo para estén disponibles en nuestro programa.
- Python tiene dos formas de encontrar un módulo:
  - En la carpeta actual de trabajo.
  - En las rutas definidas en la variable de entorno **PYTHONPATH**.
- Se puede ver los distintos tipos y módulos estándar de la **Biblioteca de Python para la versión 3.10** en la URL:

  https://docs.python.org/es/3.10/library/index.html

```
>>> import math
>>> cat1 = 3.12
>>> cat2 = 5.27
>>> hip = math.sqrt(cat1**2 + cat2**2)
>>> hipotenusa
6.124320370457443
```

UT05.01 – Funciones. Conceptos Básicos.
10.- Módulos en Python.
10.2.- Importación desde un módulo. Alias de función.

```
>>> from math import sqrt
>>> cat1=3.12
>>> cat2 = 4.32
>>> hip = sqrt(cat1**2 + cat2**2)
>>> hip
5.328864794681884

>>> from math import sqrt as raizCuadrada
>>> cat1 = 5.27
>>> cat2 = 4.87
>>> hip =raizCuadrada(cat1**2 + cat2**2)
>>> hip
7.1756393443372
```

Importando partes de un módulo.

- Es posible que no necesitemos todo aquello que está definido en un módulo sino que solo deseamos usar una función en particular.
- Al importar el módulo entero se usa memoria adicional para elementos que no deseamos usar.
- Para realizar la importación de una función o funciones usaremos la sintaxis:

  from modulo import funcion1, funcion2, ...

Importar usando un alias.

- Hay ocasiones en las que interesa (colisión de otros nombres, mejorar la legibilidad, etc) usar un nombre diferente para el módulo u objeto que estamos importando.
- En Python es posible añadiendo as alias detras de la sentencia de importación. La sintaxis completa sería la siguiente:

  import modulo as alias
  from modulo import función as alias

UT05.01 – Funciones. Conceptos Básicos.
11.- Paquetes.

- Un paquete es simplemente una carpeta que contiene ficheros .py.
- Además permite tener una jerarquía con más de un nivel de subcarpetas anidadas.
- En la carpeta raiz del paquete está el fichero **__init__.py** que realiza funciones de inicialización de las funciones de dicho paquete.
- Para importar un paquete se hace mediante la sintaxis:

  from paquete import modulo1, modulo2, ...

```
>>> import math
>>> cat1 = 3.12
>>> cat2 = 5.27
>>> hip = math.sqrt(cat1**2 + cat2**2)
>>> hipotenusa
6.124320370457443
```