

**UNIVERSIDAD CATÓLICA DE SANTA MARÍA ESCUELA  
PROFESIONAL DE INGENIERÍA DE SISTEMAS**

**SESIÓN N° 02:****MÓDULOS EN PYTHON****I****OBJETIVOS**

- ❖ Conocer los principios básicos que rigen el uso de módulos en Python.
- ❖ Aprender a diseñar módulos en Python.
- ❖ Apreciar las características y ventajas provistas por los módulos en la solución de problemas.

**II****TEMAS A TRATAR**

- ❖ Introducción.
- ❖ Importación absoluta y relativa.
- ❖ ¿Qué es un módulo?
- ❖ ¿Cómo importar un módulo?
- ❖ La palabra clave as.
- ❖ Módulos Incorporados.
- ❖ Paquetes.

**III****MARCO TEÓRICO****1. INTRODUCCIÓN**

El código de computadora tiene una tendencia a crecer. Podemos decir que el código que no crece probablemente sea completamente inutilizable o esté abandonado. Un código real, deseado y ampliamente utilizado se desarrolla continuamente, ya que tanto las demandas de los usuarios como sus expectativas se desarrollan de manera diferente.

El código creciente es, de hecho, un problema creciente. Un código más grande siempre significa un mantenimiento más difícil. La búsqueda de errores siempre es más fácil cuando el código es más pequeño.

Si se desea que dicho proyecto de software se complete con éxito, se deben tener los medios que permitan:

- Dividir todas las tareas entre los desarrolladores.
- Después, unir todas las partes creadas en un todo funcional.

Por ejemplo, un determinado proyecto se puede dividir en dos partes principales:

- La interfaz de usuario (la parte que se comunica con el usuario mediante widgets y una pantalla gráfica).
- La lógica (la parte que procesa los datos y produce resultados).

Cada una de estas partes se puede dividir en otras más pequeñas, y así sucesivamente. Tal proceso a menudo se denomina descomposición.

¿Cómo se divide una pieza de software en partes separadas pero cooperantes? Esta es la pregunta. Los módulos son la respuesta.

Un módulo le permite organizar lógicamente su código de Python. Agrupar el código relacionado en un módulo hace que el código sea más fácil de entender y usar. Un módulo es un objeto de Python con atributos nombrados arbitrariamente que puede vincular y hacer referencia. Simplemente, un módulo es un archivo que consta de código Python. Un módulo puede definir funciones, clases y variables. Un módulo también puede incluir código ejecutable.

## 2. IMPORTACIÓN ABSOLUTA Y RELATIVA

Importar se refiere a permitir que un archivo de Python o un módulo de Python acceda al script desde otro archivo o módulo de Python. Solo puede usar funciones y propiedades a las que puede acceder su programa. Por ejemplo, si desea utilizar funcionalidades matemáticas, primero debe importar el paquete matemático. Esto se debe a que debe definir todo lo que desea usar en Python antes de usarlos.

Las importaciones funcionan de la siguiente manera: Cuando se importa un módulo, el intérprete primero lo busca en `sys.modules` el caché de todos los módulos que se han importado previamente. Si no se encuentra, busca en todos los módulos incorporados con ese nombre; si lo encuentra, el intérprete ejecuta todo el código y lo pone a disposición del archivo. Si no se encuentra el módulo, busca un archivo con el mismo nombre en la lista de directorios proporcionada por la variable `sys.path`.

`sys.path` es una variable que contiene una lista de rutas que contiene bibliotecas, paquetes y un directorio de Python que contiene el script de entrada. Por ejemplo, `math` se importa un módulo llamado, luego el intérprete lo busca en un módulo incorporado, si no se encuentra, busca un archivo nombrado `math.py` en la lista de directorios proporcionada por `sys.path`.

```
import math
print(math.pi)
```

*Ejemplo 1: Importación librería math*

### Importaciones absolutas:

La importación absoluta implica la ruta completa, es decir, desde la carpeta raíz del proyecto hasta el módulo deseado. Un estado de importación absoluto que indica que el recurso se va a importar utilizando su ruta completa desde la carpeta raíz del proyecto.

Por ejemplo, si tenemos la siguiente estructura de directorios:

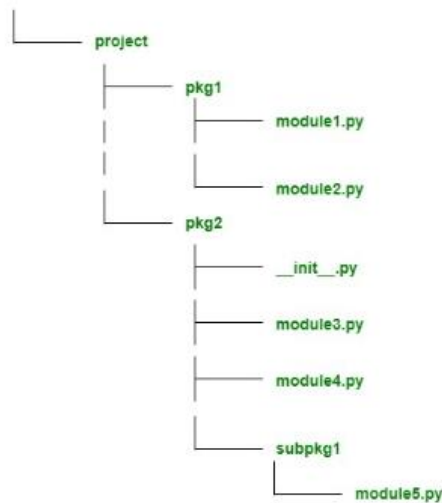


Imagen 1: Estructura de directorios

Aquí un directorio llamado proyecto, en virtud del cual dos subdirectorios a saber pkg1, pkg2. pkg1 tiene dos módulos: módulo1 y módulo2.

pkg2 contiene tres módulos: module3, module4 e \_\_init\_\_.py y un nombre de subpaquete subpkg1 que contiene module5.py. Asumamos lo siguiente:

- pkg1 / module1.py contener una función, fun1
- pkg2 / module3.py contener una función, fun2
- pkg2 / subpkg1 / module5.py contener una función fun3

```
from pkg1.module1 import fun1
from pkg2 import module3
from pkg2.module3 import fun2
from pkg2.subpkg1.module5 import fun3
```

Ejemplo 2: Importación de módulos y funciones de diversos paquetes

En este ejemplo, estamos importando los módulos escribiendo la ruta completa desde su carpeta raíz.

### Pros y contras de las importaciones absolutas:

#### Pros:

- Las importaciones absolutas son muy útiles porque son claras y van al grano.
- La importación absoluta es fácil de saber exactamente desde dónde está el recurso importado, con solo mirar la declaración.
- La importación absoluta sigue siendo válida incluso si cambia la ubicación actual de la declaración de importación.

#### Contras:

- Si la estructura del directorio es muy grande, el uso de importaciones absolutas no es significativo. En tal caso, el uso de importaciones relativas funciona bien.

```
from pkg1.subpkg2.subpkg3.subpkg4.module5 import fun6
```

Ejemplo 3: Importación absoluta

**Importaciones relativas:**

La importación relativa especifica el objeto o módulo importado desde su ubicación actual, que es la ubicación donde reside la declaración de importación. Hay dos tipos de importaciones relativas:

**Importaciones relativas implícitas:**

La importación relativa implícita se ha rechazado en Python(3.x).

**Importaciones relativas explícitas:**

La importación relativa explícita se aprobó en Python(3.x).

**Sintaxis y ejemplos prácticos:**

La sintaxis de la importación relativa depende de la ubicación actual, así como de la ubicación del módulo u objeto a importar. Las importaciones relativas utilizan la notación de puntos(.) Para especificar una ubicación. Un solo punto especifica que el módulo está en el directorio actual, dos puntos indican que el módulo está en su directorio padre de la ubicación actual y tres puntos indican que está en el directorio de los abuelos, etc.

Supongamos lo siguiente:

- pkg1 / module1.py contener una función, fun1
- pkg2 / module3.py contener una función, fun2
- pkg2 / subpkg1 / module5.py contener una función fun3

```
from .module1 import fun1
from .module3 import fun2
from .subpackage1.module5
```

*Ejemplo 4: Importación relativa de funciones*

**Pros y contras de las importaciones relativas:****Pros:**

- Trabajar con importaciones relativas es conciso y claro.
- Según la ubicación actual, reduce la complejidad de una declaración de importación.

**Contras:**

- Las importaciones relativas no son tan legibles como las absolutas.
- Usar importaciones relativas no es fácil porque es muy difícil saber la ubicación de un módulo.

**Ejemplo de importaciones relativas explícitas:**

```
from .moduleY import spam
from .moduleY import spam as ham
from . import moduleY
from ..subpackage1 import moduleY
from ..subpackage2.moduleZ import eggs
from ..moduleA import foo
from ...package import bar
from ...sys import path
```

*Ejemplo 5: Importación relativa de funciones con alias*

### 3. ¿QUÉ ES UN MÓDULO?

Es un archivo que contiene definiciones y sentencias de Python, que se pueden importar más tarde y utilizar cuando sea necesario.

El manejo de los módulos consta de dos cuestiones diferentes:

- El primero (probablemente el más común) ocurre cuando se desea utilizar un módulo ya existente - escrito por otra persona o creado por el programador mismo en algún proyecto complejo: en este caso, se considera al programador como el usuario del módulo.
- El segundo ocurre cuando se desea crear un nuevo módulo, ya sea para uso propio o para facilitar la vida de otros programadores, en este caso tu eres el proveedor del módulo.

Todos estos módulos, junto con las funciones integradas, forman la Biblioteca Estándar de Python - un tipo especial de biblioteca donde los módulos desempeñan el papel de libros (incluso podemos decir que las carpetas desempeñan el papel de estanterías).

La lista completa de todos los «volúmenes» recopilados en esa biblioteca, se puede encontrar aquí: <https://docs.python.org/3/library/index.html>.

Cada módulo consta de entidades (como un libro consta de capítulos). Estas entidades pueden ser funciones, variables, constantes, clases y objetos. Si se sabe cómo acceder a un módulo en particular, se puede utilizar cualquiera de las entidades que almacena.

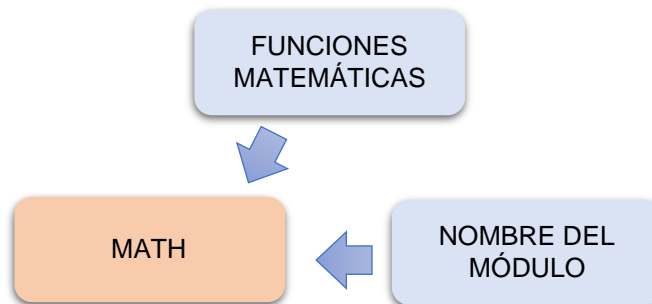


Imagen 2: Módulo Math

Los módulos en Python son grupos de funciones, variables o clases alojadas dentro de un archivo .py. Un módulo le permite organizar lógicamente su código de Python. Agrupar el código relacionado en un módulo hace que el código sea más fácil de entender y usar. Un módulo es un objeto de Python con atributos nombrados arbitrariamente que puede vincular y hacer referencia. Vamos a crear un ejemplo, un carrito de compra (ecommerce) y calcular el impuesto de un artículo en específico dado su precio.

```
def calcularImpuesto(precio, impuesto):  
    precioNuevo = precio / 100 * (100 + impuesto)  
    return precioNuevo
```

Ejemplo 6: calcular Impuesto

Observemos que fue sencillo de realizar, lo que hicimos fue definir una función llamada calcularImpuesto la cual toma dos argumentos precio e impuesto, para luego calcular y retornar el valor total del artículo.

Para que este código sea o cumpla como un módulo en Python, lo único que tenemos que hacer es guardarlo con la extensión .py en el mismo directorio donde residen nuestros otros scripts de la aplicación. Vamos a guardar el archivo con el siguiente nombre: finanzas.py y así nuestro módulo será el módulo de finanzas.

### 4. ¿CÓMO IMPORTAR UN MÓDULO?

Para que un módulo sea utilizable, hay que importarlo (piensa en ello como sacar un libro del estante). La importación de un módulo se realiza mediante una instrucción llamada import. Supongamos que desea utilizar dos entidades proporcionadas por el módulo math:

Un símbolo (constante) que representa un valor preciso (tan preciso como sea posible usando aritmética de punto flotante doble) de  $\pi$  (aunque usar una letra griega para nombrar una variable es totalmente posible en Python, el símbolo se llama pi - es una solución más conveniente, especialmente para esa parte del mundo que ni tiene ni va a usar un Teclado Griego).

Una función llamada `sin()` (el equivalente informático de la función matemática seno).

Ambas entidades están disponibles a través del módulo `math`, pero la forma en que se pueden usar depende en gran medida de cómo se haya realizado la importación.

```
import math
```

*Ejemplo 7: Importar módulo Math*

La instrucción puede colocarse en cualquier parte del código, pero debe colocarse antes del primer uso de cualquiera de las entidades del módulo.

Si quiere (o tiene que) importar más de un módulo, puede hacerlo repitiendo la cláusula `import` (preferida):

```
import math
import sys
```

*Ejemplo 8: Importar dos módulos*

o enumerando los módulos después de la palabra clave reservada `import`, como aquí:

```
import math, sys
```

*Ejemplo 9: Importar dos módulos en una sola línea*

Observe el siguiente ejemplo solo se desea imprimir el valor de  $\sin(\frac{1}{2}\pi)$ .

```
import math
print(math.sin(math.pi/2))
```

*Ejemplo 10: Hacer uso de la función seno del módulo Math*

Ahora demostraremos cómo pueden dos namespaces (el tuyo y el del módulo) coexistir.

```
import math

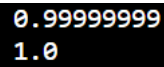
def sin(x):
    if 2 * x == pi:
        return 0.99999999
    else:
        return None

pi = 3.14

print(sin(pi/2))
print(math.sin(math.pi/2))
```

*Ejemplo 11: Uso de dos namespaces*

Como puede ver, las entidades no se afectan entre sí. El resultado que se muestra es el siguiente:



```
0.99999999
1.0
```

*Imagen 3: Resultado obtenido de dos namespaces*

En el segundo método, la sintaxis del import señala con precisión que entidad (o entidades) del módulo son aceptables en el código:

```
from math import pi
```

*Ejemplo 12: Segundo método de importación función pi*

Observa el siguiente ejemplo:

La línea 01: lleva a cabo la importación selectiva.

La línea 03: hace uso de las entidades importadas y obtiene el resultado esperado (1.0).

Las líneas 05 a la 12: redefinen el significado de pi y sin - en efecto, reemplazan las definiciones originales (importadas) dentro del namespace del código.

La línea 15: retorna 0.99999999, lo cual confirma nuestras conclusiones.

```
from math import sin, pi

print(sin(pi / 2))

pi = 3.14

def sin(x):
    if 2 * x == pi:
        return 0.99999999
    else:
        return None
```

*Ejemplo 13: Segundo método de importación*

La salida que muestra el código anterior es:



```
1.0
```

*Imagen 4: Resultado obtenido del segundo método de importación*

Realice otra prueba. Observa el código a continuación:

```
def sin(x):
    if 2 * x == pi:
        return 0.99999999
    else:
        return None

pi=3.14
print(sin(pi / 2))
from math import sin, pi
print(sin(pi / 2))
```

*Ejemplo 14: Segundo método de importación con dos namespaces*

El resultado que se muestra es:

```
0.99999999
1.0
```

*Imagen 5: Resultado obtenido del segundo método de importación con dos namespaces*

En el tercer método, la sintaxis del import es una forma más agresiva que la presentada anteriormente:

```
from module import *
```

*Ejemplo 15: Tercer método de importación*

Como puede ver, el nombre de una entidad (o la lista de nombres de entidades) se reemplaza con un solo asterisco (\*). Tal instrucción importa todas las entidades del módulo indicado.

Veamos como importar el ejemplo en donde teníamos el módulo de finanzas (finanzas.py):

```
import finanzas
```

*Ejemplo 16: Importar módulo Finanzas*

La otra manera es usando la palabra clave from, la cual importará única y exclusivamente la función que uno le pasa como parámetro, es decir, imaginemos que tenemos un módulo que posee miles y miles de funciones, para que importar todas esas funciones dentro de nuestro script si solo vamos a necesitar una, es en este caso donde viene a la mano el from.

Veamos como importar una funcionalidad del módulo de finanzas:

```
from finanzas import calcularImpuesto
```

*Ejemplo 17: Importar la función calcularImpuesto del módulo Finanzas*

Observemos que necesitamos especificar de qué módulo queremos obtener nuestra función con from + MÓDULO y luego la función que queremos importar import + FUNCIÓN.

En nuestro caso, del módulo de finanzas importaremos la función calcularImpuesto. También puedes importar varias funciones a la vez, solo necesitas separarlas por comas.

```
from finanzas import calcularImpuesto, calcularDescuento
```

*Ejemplo 18: Importar dos funciones del módulo Finanzas*

Incluso se puede importar todas las funciones usando un asterisco:

```
from finanzas import *
```

*Ejemplo 19: Importar todas las funciones del módulo Finanzas*

## 5. LA PALABRA CLAVE AS

Si se importa un módulo y no se está conforme con el nombre del módulo en particular puede dársele otro nombre: esto se llama aliasing o renombrado.



Aliasing (renombrado) hace que el módulo se identifique con un nombre diferente al original. Esto también puede acortar los nombres originales.

La creación de un alias se realiza junto con la importación del módulo, y exige la siguiente forma de la instrucción import:

```
import math as m
print(m.sin(m.pi/2))
```

*Ejemplo 20: Uso de alias en la importación de un módulo*

La frase name as alias puede repetirse: puede emplear comas para separar las frases, como a continuación:

```
from math import pi as PI, sin as sine
print(sine(PI/2))
```

*Ejemplo 21: Uso de alias en la importación de varias funciones de un módulo*

## 6. MÓDULOS INCORPORADOS

Existen miles de módulos incorporados dentro de Python. Debido a que el rango es extenso, ahora mostraremos los más útiles.

- math
- random
- os
- datetime

### A. MATH

Comencemos con una vista previa de algunas de las funciones proporcionadas por el módulo math. El primer grupo de funciones de módulo math están relacionadas con trigonometría:

$\sin(x)$  → el seno de  $x$ .

$\cos(x)$  → el coseno de  $x$ .

$\tan(x)$  → la tangente de  $x$ .

Todas estas funciones toman un argumento (una medida de ángulo expresada en radianes) y devuelven el resultado apropiado.

Por supuesto, también están sus versiones inversas:

$\arcsin(x)$  → el arcoseno de  $x$ .

$\arccos(x)$  → el arcocoseno de  $x$ .

$\operatorname{arctan}(x)$  → el arcotangente de  $x$ .

Estas funciones toman un argumento (verifican que sea correcto) y devuelven una medida de un ángulo en radianes.

Para trabajar eficazmente con mediciones de ángulos, el módulo math proporciona las siguientes entidades:

$\pi$  → una constante con un valor que es una aproximación de  $\pi$ .

$\text{radians}(x)$  → una función que convierte  $x$  de grados a radianes.

$\text{degrees}(x)$  → una función que convierte  $x$  de radianes a grados.

El módulo de math nos provee el acceso a funciones y constantes matemáticas. Por ejemplo:

```
import math
print(math.pi) #Pi, 3.14...
print(math.e) #Número de Euler, 2.71...
print(math.degrees(2)) #2 radianes = 114.59 grados
print(math.radians(60)) #60 grados = 1.04 radianes
print(math.sin(2)) #Seno de 2 radianes
print(math.cos(0.5)) #Coseno de 0.5 radianes
print(math.tan(0.23)) #Tangente de 0.23 radianes
print(math.factorial(5)) #1 * 2 * 3 * 4 * 5 = 120
print(math.sqrt(49)) #Raíz cuadrada de 49 = 7
```

*Ejemplo 22: Uso de funciones del módulo Math*

El resultado que se muestra es como sigue:

```
3.141592653589793
2.718281828459045
114.59155902616465
1.0471975511965976
0.9092974268256817
0.8775825618903728
0.23414336235146527
120
7.0
```

*Imagen 6: Resultado de las funciones del módulo Math*

```
from math import pi, radians, degrees, sin, cos, tan, asin

ad = 90
ar = radians(ad)
ad = degrees(ar)

print(ad == 90.)
print(ar == pi / 2.)
print(sin(ar) / cos(ar) == tan(ar))
print(asin(sin(ar)) == ar)
```

*Ejemplo 23: Otra forma de importar funciones del módulo Math*

El resultado obtenido es el siguiente:

```
True
True
True
True
```

*Imagen 7: Resultados segunda forma de importación módulo Math*

Existe otro grupo de las funciones math relacionadas con la exponenciación:

$e \rightarrow$  una constante con un valor que es una aproximación del número de Euler ( $e$ )

$\exp(x) \rightarrow$  encontrar el valor de  $e^x$ .

$\log(x) \rightarrow$  el logaritmo natural de  $x$ .

$\log(x, b) \rightarrow$  el logaritmo de  $x$  con base  $b$ .

$\log_{10}(x) \rightarrow$  el logaritmo decimal de  $x$  (más preciso que  $\log(x, 10)$ )

$\log_2(x) \rightarrow$  el logaritmo binario de  $x$  (más preciso que  $\log(x, 2)$ )

Nota: la función `pow` es una función incorporada y no se tiene que importar.

```
from math import e, exp, log

print(pow(e, 1) == exp(log(e)))
print(pow(2, 2) == exp(2 * log(2)))
print(log(e, e) == exp(0))
```

*Ejemplo 24: Funciones exponenciales del módulo Math*

El resultado se muestra a continuación:

```
True
True
True
```

*Imagen 8: Resultados de funciones exponenciales del módulo Math*

El último grupo de funciones consta de algunas funciones de propósito general como:

`ceil(x)` → devuelve el entero más pequeño mayor o igual que `x`.

`floor(x)` → el entero más grande menor o igual que `x`.

`trunc(x)` → el valor de `x` truncado a un entero (ten cuidado, no es equivalente a `ceil` o `floor`).

`factorial(x)` → devuelve `x!` (`x` tiene que ser un valor entero y no negativo).

`hypot(x, y)` → devuelve la longitud de la hipotenusa de un triángulo rectángulo con las longitudes de los catetos iguales a `(x)` y `(y)` (lo mismo que `sqrt(pow(x, 2) + pow(y, 2))` pero más preciso).

```
from math import ceil, floor, trunc

x = 1.4
y = 2.6

print(floor(x), floor(y))
print(floor(-x), floor(-y))
print(ceil(x), ceil(y))
print(ceil(-x), ceil(-y))
print(trunc(x), trunc(y))
print(trunc(-x), trunc(-y))
```

*Ejemplo 25: Funciones de propósito general del módulo Math*

Los datos que se obtienen son:

```
1 2
-2 -3
2 3
-1 -2
1 2
-1 -2
```

*Imagen 9: Resultado de funciones de propósito general del módulo Math*

## B. RANDOM

La función general llamada `random()` (no debe confundirse con el nombre del módulo) produce un número flotante `x` entre el rango `(0.0, 1.0)`; en otras palabras: `(0.0 <= x < 1.0)`.

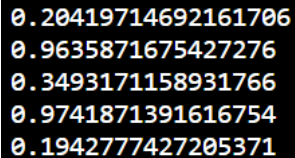
El programa de ejemplo a continuación producirá cinco valores pseudoaleatorios ya que sus valores están determinados por el valor semilla actual (bastante impredecible).

```
from random import random

for i in range(5):
    print(random())
```

*Ejemplo 26: Uso de la función random del módulo Random*

El resultado que se obtiene es:



```
0.20419714692161706
0.9635871675427276
0.3493171158931766
0.9741871391616754
0.1942777427205371
```

*Imagen 10: Resultado de la función random del módulo Random*

La función seed() es capaz de directamente establecer la semilla del generador.

seed() - establece la semilla con la hora actual.

seed(int\_value) - establece la semilla con el valor entero int\_value.

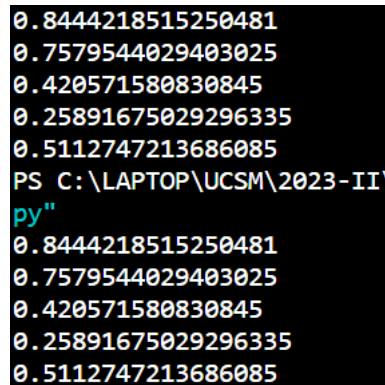
```
from random import random, seed

seed(0)

for i in range(5):
    print(random())
```

*Ejemplo 27: Uso de la función random del módulo Random*

En este caso si ejecutamos dos veces verificamos que muestra los mismos números aleatorios generados:



```
0.8444218515250481
0.7579544029403025
0.420571580830845
0.25891675029296335
0.5112747213686085
PS C:\LAPTOP\UCSM\2023-II\
py"
0.8444218515250481
0.7579544029403025
0.420571580830845
0.25891675029296335
0.5112747213686085
```

*Imagen 11: Resultado de la función seed del módulo Random*

Si desea valores aleatorios enteros, una de las siguientes funciones encajaría mejor:

randrange(fin)

randrange(inicio, fin)

randrange(inicio, fin, incremento)

randint(izquierda, derecha)

La última función es equivalente a randrange(izquierda, derecha+1) - genera el valor entero i, el cual cae en el rango [izquierda, derecha] (sin exclusión en el lado derecho).

```
from random import randrange, randint

print(randrange(1), end=' ')
print(randrange(0, 1), end=' ')
print(randrange(0, 1, 1), end=' ')
print(randint(0, 1))
```

*Ejemplo 28: Uso de las funciones randrange y randint del módulo Random*

**0 0 0 1**

*Imagen 12: Resultado de las funciones randrange y randint del módulo Random*

Como puedes ver, esta no es una buena herramienta para generar números para la lotería. Afortunadamente, existe una mejor solución que escribir tu propio código para verificar la singularidad de los números "sorteados". Es una función con el nombre de choice:

choice(secuencia)

sample(secuencia, elementos\_a\_elegir=1)

Observa el código a continuación:

```
from random import choice, sample

my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(choice(my_list))
print(sample(my_list, 5))
print(sample(my_list, 10))
```

*Ejemplo 29: Uso de las funciones choice y sample del módulo Random*

**10**  
**[7, 5, 3, 1, 2]**  
**[1, 2, 10, 7, 8, 5, 9, 6, 4, 3]**

*Imagen 13: Resultado de las funciones choice y sample del módulo Random*

El módulo random nos permite generar números aleatorios. Por ejemplo:

```
import random
# Flotante aleatorio >= 0 y < 1.0
print(random.random())
# Flotante aleatorio >= 1 y <10.0
print(random.uniform(1,10))
# Entero aleatorio de 0 a 9, 10 excluido
print(random.randrange(10))
# Entero aleatorio de 0 a 100
print(random.randrange(0,101))
# Entero aleatorio de 0 a 100 cada 2 números, múltiplos de 2
print(random.randrange(0,101,2))
# Entero aleatorio de 0 a 100 cada 5 números, múltiplos de 5
print(random.randrange(0,101,5))
```

*Ejemplo 30: Uso de funciones varias del módulo Random*

El resultado obtenido es:

```
0.7398548189670224
8.38316413088923
0
40
84
75
```

Imagen 14: Resultado de las funciones varias del módulo Random

### C. DATETIME

Si necesitas trabajar con fechas y tiempos, entonces el módulo datetime es el indicado. Veamos el siguiente ejemplo:

```
import datetime
from datetime import date
import time
#Retorna el número de segundos desde el 1 de enero de 1970 (Unix
    Epoch)
print(time.time())
#Convierte número de segundos en un objeto tipo date
print(date.fromtimestamp(123456789))
#Podemos combinar las funciones time y fromtimestamp para
    representar el tiempo actual
print(date.fromtimestamp(time.time()))
currentDate = date.fromtimestamp(time.time())
#Creamos una variable con la representación del tiempo actual
print(currentDate)
#Le coloca el siguiente formato a la fecha DD/MM/YY
print(currentDate.strftime("%d/%m/%y"))
#Le coloca el formato estándar ISO a esa fecha
print(currentDate.isoformat())
```

Ejemplo 31: Uso de las funciones del módulo Datetime

Los datos que se muestran son:

```
1691520144.5048826
1973-11-29
2023-08-08
2023-08-08
08/08/23
2023-08-08
```

Imagen 15: Resultado de las funciones varias del módulo Random

### D. OS

Este módulo te permite trabajar con el sistema operativo en el cual Python este ejecutándose, ya sea Windows, Mac o Linux. Nos enfocaremos en la funcionalidad path ya que es la más común. Path nos permite manipular y encontrar propiedades de los archivos y carpetas que existen en el sistema. A continuación, se detalla un ejemplo:

```
from os import path

print(path.exists("C:/LAPTOP/UCSM/2023-II/LP1/Prácticas
2023/Lab02/Actividad1_1/Modulos/modulo.py"))
#Retorna la fecha en la cual ese directorio fue accedido por última
    vez
print(path.getatime("C:/LAPTOP/UCSM/2023-II/LP1/Prácticas
2023/Lab02/Actividad1_1/Modulos"))
#Retorna la fecha en la cual ese directorio fue modificado por
    ultima vez
print(path.getmtime("C:/LAPTOP/UCSM/2023-II/LP1/Prácticas
2023/Lab02/Actividad1_1/Modulos"))
```

```
#Retorna el tamaño en bytes de ese archivo
print(path.getsize("C:/LAPTOP/UCSM/2023-II/LP1/Prácticas
2023/Lab02/Actividad1_1/Modulos/modulo.py"))
#Retorna la siguiente dirección "C:/Users"
path.join("C:", "Laptop")
```

*Ejemplo 32: Uso de las funciones del módulo OS*

El resultado que se muestra es:

```
True
1691520738.2453914
1691510528.9737148
565
```

*Imagen 16: Resultado de las funciones del módulo OS*

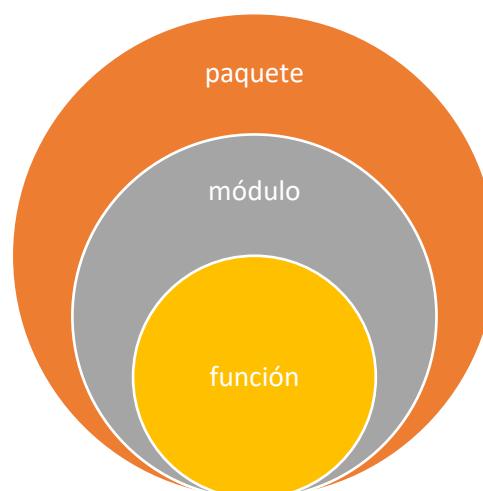
## 7. PAQUETES

Un módulo es un contenedor lleno de funciones, puede empaquetar tantas funciones como desees en un módulo y distribuirlo por todo el mundo.

Por supuesto, no es una buena idea mezclar funciones con diferentes áreas de aplicación dentro de un módulo (al igual que en una biblioteca: nadie espera que los trabajos científicos se incluyan entre los cómics), así que se deben agrupar las funciones cuidadosamente y asignar un nombre claro e intuitivo al módulo que las contiene (por ejemplo, no le des el nombre videojuegos a un módulo que contiene funciones destinadas a particionar y formatear discos duros).

Crear muchos módulos puede causar desorden: tarde que temprano querrás agrupar tus módulos de la misma manera que previamente has agrupado funciones: ¿Existe un contenedor más general que un módulo?

Sí lo hay, es un paquete: en el mundo de los módulos, un paquete juega un papel similar al de una carpeta o directorio en el mundo de los archivos.



*Imagen 17: Representación función, módulo y paquete*

## IV

## ACTIVIDADES

## I. Uso de módulos

1. Cree una carpeta lab\_02 en el explorador de Windows y selecciónela desde VSCode.
2. Crear una subcarpeta Actividad1.
3. Dentro de dicha carpeta crea un archivo vacío y asigne el nombre modulo.py.

4. El segundo archivo contiene el código que utiliza el nuevo módulo. Su nombre es principal.py. Su contenido es el siguiente:

```
import module
```

Nota: ambos archivos deben estar ubicados en la misma carpeta.  
Ejecuta el archivo principal.py. ¿Qué es lo que ves?

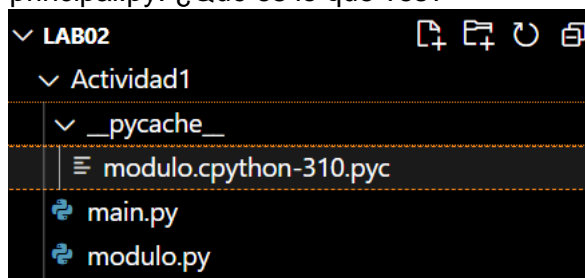


Imagen 18: Estructura de la carpeta Actividad1

Ha aparecido una nueva subcarpeta, `__pycache__`. Hay un archivo llamado `module.cpython-xy.pyc` donde `x` y `y` son dígitos derivados de tu versión de Python (por ejemplo, serán 3 y 10 si utilizas Python 3.10).

El nombre del archivo es el mismo que el de tu módulo. La parte posterior al primer punto dice qué implementación de Python ha creado el archivo (CPython) y su número de versión. La última parte (`.pyc`) viene de las palabras Python y compilado.

Puede mirar dentro del archivo: el contenido es completamente ilegible para los humanos. Tiene que ser así, ya que el archivo está destinado solo para uso del uso de Python. Cuando Python importa un módulo por primera vez, traduce el contenido a una forma algo compilada.

Gracias a eso, cada importación posterior será más rápida que interpretar el código fuente desde cero.

5. Ahora modificamos el archivo del módulo.py:

```
print("Me gusta ser un módulo.")
```

6. Volvamos al archivo principal.py y lo ejecutamos:



```
Me gusta ser un módulo.
```

Cuando un módulo es importado, su contenido es ejecutado implícitamente por Python. Le da al módulo la oportunidad de inicializar algunos de sus aspectos internos (por ejemplo, puede asignar a algunos variables valores útiles).

Nota: la inicialización se realiza sólo una vez, cuando se produce la primera importación, por lo que las asignaciones realizadas por el módulo no se repiten innecesariamente.

7. Python puede hacer mucho más que solo importar el módulo. También crea una variable llamada `__name__`.

A continuación, modifica el archivo `módulo.py`:

```
print("Me gusta ser un módulo.")
print(__name__)
```

Ahora ejecuta el archivo `module.py`. Deberías ver las siguientes líneas:

```
Me gusta ser un módulo.
__main__
```

Ahora ejecuta el archivo `principal.py`. Observa lo que se muestra:

```
Me gusta ser un módulo.
modulo
```

8. De esta forma se puede ver el uso de la variable `__principal__` para detectar el contexto en el cual se activó el código, modifica el archivo `modulo.py`:

```
if __name__ == "__main__":
    print("Prefiero ser módulo.")
else:
    print("Me gusta ser un módulo.")
```

9. Este módulo contendrá dos funciones simples, y si desea saber cuántas veces se han invocado las funciones, necesita un contador inicializado en cero cuando se importe el módulo. Puede hacerlo de esta manera, modifique el archivo `modulo.py`:

```
counter = 0

if __name__ == "__main__":
    print("Prefiero ser un módulo.")
else:
    print("Me gusta ser un módulo.")
```

10. El introducir tal variable es absolutamente correcto, pero puede causar importantes efectos secundarios que debes tomar en cuenta. Analiza el archivo modificado `principal.py`:

```
import module
print(module.counter)
```

Se imprime en pantalla:

```
Me gusta ser un módulo.
0
```

A diferencia de muchos otros lenguajes de programación, Python no tiene medios para permitirte ocultar tales variables a los ojos de los usuarios del módulo. Solo puede informar a sus usuarios que esta es su variable, que pueden leerla, pero que no deben modificarla bajo ninguna circunstancia.

Esto se hace anteponiendo al nombre de la variable `_` (un guión bajo) o `__` (dos guiones bajos), pero recuerda, es solo un acuerdo. Los usuarios de tu módulo pueden obedecerlo o no.

Ahora pongamos dos funciones en el módulo: evaluarán la suma y el producto de los números recopilados en una lista.

11. Escribamos un código nuevo en nuestro archivo `modulo.py`. El módulo actualizado está listo aquí:

```
#modulo.py - un ejemplo de módulo Python

__counter = 0

def suml(the_list):
    global __counter
    __counter += 1
    the_sum = 0
    for element in the_list:
        the_sum += element
    return the_sum

def prodl(the_list):
    global __counter
    __counter += 1
    prod = 1
    for element in the_list:
        prod *= element
    return prod

if __name__ == "__main__":
    print("Prefiero ser un módulo, pero puedo hacer algunas pruebas para usted.")
    my_list = [i+1 for i in range(5)]
    print(suml(my_list) == 15)
    print(prodl(my_list) == 120)
```

12. Ahora es posible usar el nuevo módulo, esta es una forma de hacerlo:

```
from modulo import suml, prodl

zeroes = [0 for i in range(5)]
ones = [1 for i in range(5)]
print(suml(zeroes))
print(prodl(ones))
```

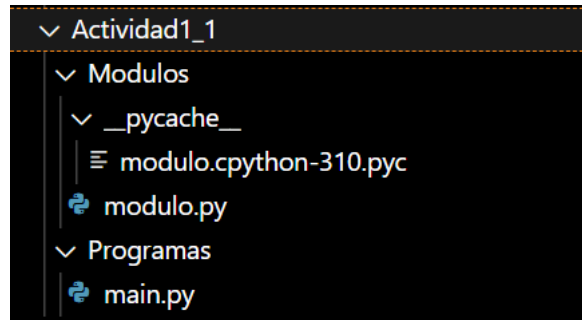
Ello muestra como salida lo siguiente:

0

1

13. Es hora de hacer este ejemplo más complejo: hemos asumido aquí que el archivo Python principal se encuentra en la misma carpeta o directorio que el módulo que se va a importar.

Renunciemos a esta suposición y realicemos el siguiente experimento, dada la siguiente estructura:



14. ¿Cómo se puede llamar al modulo.py desde el principal.py estando en diferentes carpetas? Una de las varias soluciones posibles se ve así el archivo principal.py:

```
import sys
sys.path.append('C:/LAPTOP/UCSM/2023-II/LP1/Prácticas
2023/Lab02/Actividad1_1/Modulos')

import modulo

zeroes = [0 for i in range(5)]
ones = [1 for i in range(5)]
print(modulo.sum1(zeroes))
print(modulo.prodl(ones))
```

## II. Creación de la estructura y llamado a módulos

1. Cree una carpeta Actividad2.
2. Cree una carpeta Modulos dentro de la carpeta Actividad2.
3. Cree un archivo modulo.py dentro de la carpeta Modulos.
4. Cree un archivo operaciones.py dentro de la carpeta Actividad2.

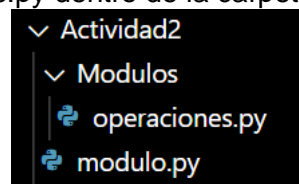


Imagen 19: Estructura de archivos Actividad2

5. El módulo operaciones.py contendrá el siguiente código:

```
#Módulo operaciones

def suma(num01, num02):
    return num01 + num02
```

6. En el archivo modulo.py llamaremos al módulo operaciones de la siguiente manera:

```
#Importar Módulo Operaciones Modalidad 01

import Modulos.operaciones as modulo

numero01 = int(input("Ingrese primer número: "))
numero02 = int(input("Ingrese segundo número: "))

resultado_suma = modulo.suma(numero01, numero02)
print(f"El resultado de la suma es {resultado_suma}")
```

7. Verificar el funcionamiento, comentar todas las líneas anteriores en el archivo modulo.py.

8. Otra modalidad para llamar al módulo operaciones es la siguiente:

```
#Importar Módulo Operaciones Modalidad 02

from Modulos.operaciones import suma

numero01 = int(input("Ingrese primer número: "))
numero02 = int(input("Ingrese segundo número: "))

resultado_suma = suma(numero01, numero02)
print(f"El resultado de la suma es {resultado_suma}")
```

9. Verificar el funcionamiento, comentar todas las líneas anteriores en el archivo modulo.py.

10. Otra modalidad para llamar al módulo operaciones es la siguiente:

```
#Importar Módulo Operaciones Modalidad 03

from Modulos.operaciones import *

numero01 = int(input("Ingrese primer número: "))
numero02 = int(input("Ingrese segundo número: "))

resultado_suma = suma(numero01, numero02)
print(f"El resultado de la suma es {resultado_suma}")
```

11. Verificar el funcionamiento e indicar las diferencias entre las diferentes modalidades.

### III. Llamado a constantes y variables desde diferentes módulos:

1. En el módulo operaciones.py defina la constante TEXTO y la variable, como se muestra a continuación.

```
TEXTO="Esto es mi constante"
variable=0
```

2. Desde el archivo modulo.py imprima los datos de la constante y la variable del módulo operación.

```
import Modulos.operaciones as modulo
#Llamada a una constante
print(modulo.TEXTO)

#Llamada a una variable
print(modulo.variable)
```

#### IV. Uso del módulo MATH

1. Muestre la raíz cuadrada de un número entero ingresado por teclado. Para ello en el archivo modulo.py

```
import math
numero = int(input("Ingrese un número: "))
raiz = round(math.sqrt(numero),2)
print(f"la raiz cuadrada de {numero} es {raiz}")
```

#### V. Uso del módulo OS

1. Muestre la carpeta raíz haciendo uso del módulo OS. Para ello en el archivo modulo.py

```
import os
print(os.getcwd())
```

#### VI. Uso del módulo DATETIME

1. Muestre la carpeta raíz haciendo uso del módulo OS. Para ello en el archivo modulo.py

```
from datetime import datetime
fecha=datetime.now()
print(fecha) #Devuelve fecha completa 2022-08-05 11:19:12.555601
```

#### VII. Uso de PAQUETES

1. Imagina que en un futuro no muy lejano, tu y tus socios escriben una gran cantidad de funciones en Python. Tu equipo decide agrupar las funciones en módulos separados, y este es el resultado final:

```
""" module: alpha """

def funA():
    return "Alpha"
```

```

if __name__ == "__main__":
    print("Prefiero ser un módulo.")

def suma(num01, num02):
    return num01 + num02

```

Nota: hemos presentado todo el contenido solo para el módulo alpha.py, supongamos que todos los módulos tienen un aspecto similar (contienen una función denominada funX, donde X es la primera letra del nombre del módulo).

<b>alpha.py</b> <pre> #!/usr/bin/env python3  """ module: alpha """  def funA():     return "Alpha"  if __name__ == "__main__":     print("I prefer to be a module.") </pre>	<b>beta.py</b> <pre> def funB(): ... </pre>
	<b>iota.py</b> <pre> def funI(): ... </pre>
	<b>sigma.py</b> <pre> def funS(): ... </pre>
<b>tau.py</b> <pre> def funT(): ... </pre>	<b>psi.py</b> <pre> def funP(): ... </pre>
	<b>omega.py</b> <pre> def funO(): ... </pre>

Imagen 20: Estructura de archivos de la Actividad7

- De repente, alguien se da cuenta de que estos módulos forman su propia jerarquía, por lo que colocarlos a todos en una estructura plana no será una buena idea. Después de algo de discusión, el equipo llega a la conclusión de que los módulos deben agruparse. Todos los participantes están de acuerdo en que la siguiente estructura de árbol refleja perfectamente las relaciones mutuas entre los módulos:

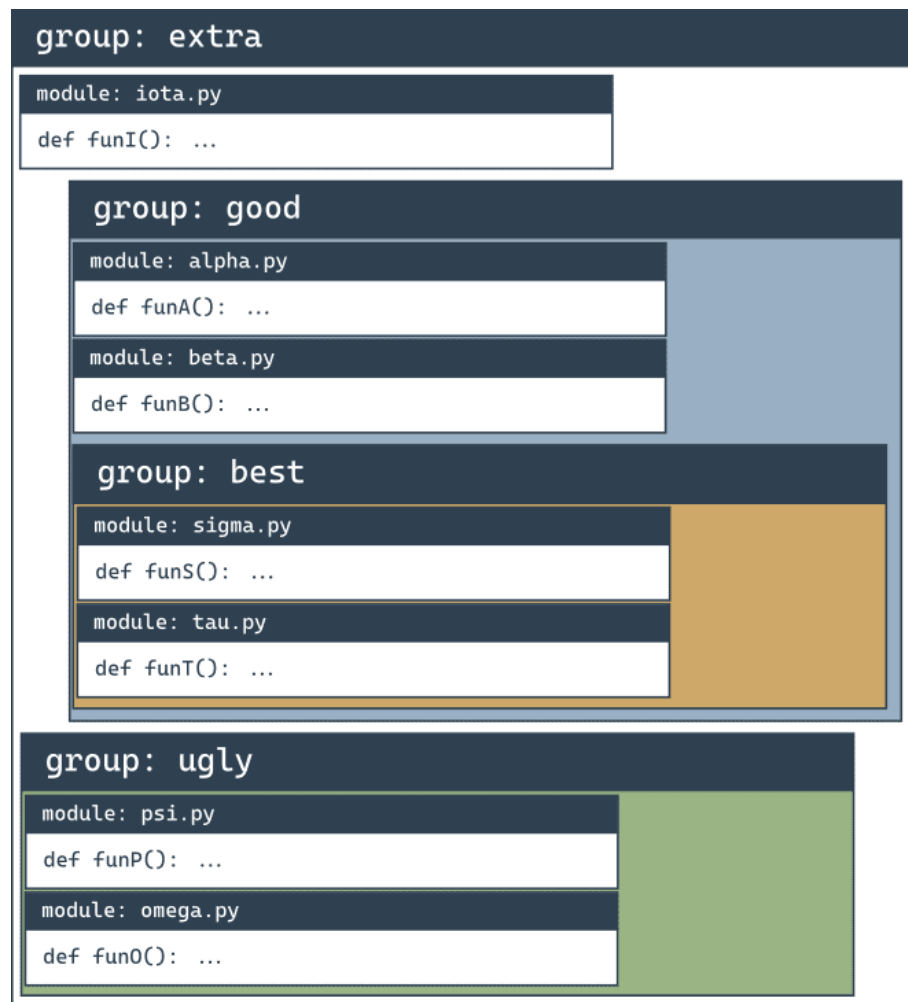


Imagen 21: Jerarquía de los archivos de la Actividad 7

3. Así es como se ve el árbol actualmente:

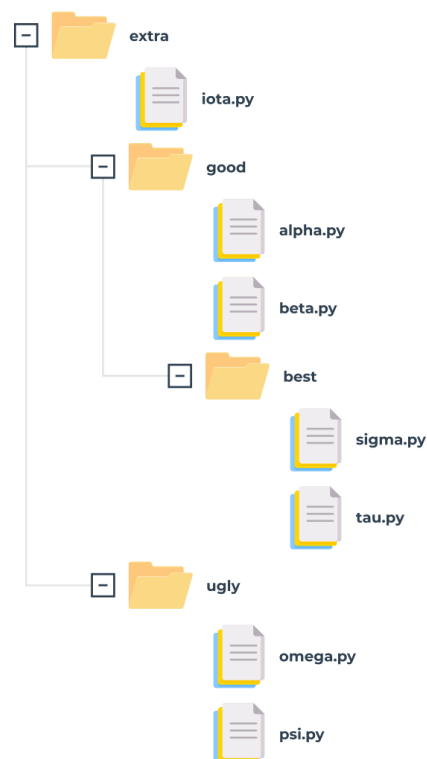


Imagen 22: Árbol de jerarquía de archivos Actividad 7

Si asume que extra es el nombre de un paquete recientemente, impondrá una regla de nomenclatura que te permitirá nombrar claramente cada entidad del árbol. Por ejemplo:

La ubicación de una función llamada funT() del paquete tau puede describirse como:

extra.good.best.tau.funT()

4. Ahora se deben responder dos preguntas:
- ¿Cómo se transforma este árbol en un paquete real de Python?
  - ¿Dónde se coloca el subárbol para que Python pueda acceder a él?
- La primera pregunta tiene una respuesta Python espera que haya un archivo con un nombre muy exclusivo dentro de la carpeta del paquete: `__init__.py`.

El contenido del archivo se ejecuta cuando se importa cualquiera de los módulos del paquete. Si no desea ninguna inicialización especial, puede dejar el archivo vacío, pero no debes omitirlo.

5. Recuerde: la presencia del archivo `__init__.py` finalmente compone el paquete:



Imagen 23: Paquete extra



Nota: no solo la carpeta raíz puede contener el archivo `__init__.py`, también puedes ponerlo dentro de cualquiera de sus subcarpetas (subpaquetes). Puede ser útil si algunos de los subpaquetes requieren tratamiento individual o un tipo especial de inicialización.

Ahora es el momento de responder la segunda pregunta, ¿Dónde se coloca el subárbol para que sea accesible para Python? La respuesta es simple: donde quiera. Solo tienes que asegurarte de que Python conozca la ubicación del paquete.

Estás listo para usar tu primer paquete.

#### 6. Supongamos que el entorno de trabajo se ve de la siguiente manera:

Este equipo > Disco local (C:) > LAPTOP > UCSM > 2023-II > LP1 > Prácticas 2023 > Lab02 > Actividad7 > extra

Nombre	Fecha de modificación	Tipo
good	19/03/2019 12:51 p. m.	Carpeta de archivos
ugly	19/03/2019 12:51 p. m.	Carpeta de archivos
__init__.py	19/03/2019 12:51 p. m.	Python File
iota.py	19/03/2019 12:51 p. m.	Python File

#### 7. Vamos a acceder a la función `funI()` del módulo `iota` del paquete `extra`. Nos obliga a usar nombres de paquetes calificados (asocia esto al nombramiento de carpetas y subcarpetas).

```
from sys import path
path.append('C:/LAPTOP/UCSM/2023-II/LP1/Prácticas
2023/Lab02/Actividad7')

import extra.iota
print(extra.iota.funI())
```

Una pequeña variación es lo siguiente:

```
from sys import path
path.append('C:/LAPTOP/UCSM/2023-II/LP1/Prácticas
2023/Lab02/Actividad7')

from extra.iota import funI
print(funI())
```

#### 8. Ahora vamos hasta el final del árbol: así es como se obtiene acceso a los módulos `sigma` y `tau`:

```
from sys import path
path.append('C:/LAPTOP/UCSM/2023-II/LP1/Prácticas
2023/Lab02/Actividad7')

import extra.good.best.sigma
from extra.good.best.tau import funT

print(extra.good.best.sigma.funS())
print(funT())
```

O haciendo uso de alias:

```
from sys import path
path.append('C:/LAPTOP/UCSM/2023-II/LP1/Prácticas
2023/Lab02/Actividad7')

import extra.good.best.sigma as sig
import extra.good.alpha as alp

print(sig.funS())
print(alp.funA())
```

**V****EJERCICIOS****(La práctica tiene una duración de 4 horas)**

Haciendo uso del lenguaje Python desarrollar los siguientes ejercicios:

1. Crear una carpeta propuesto1 desde VSCode, la cual este ubicada dentro de lab\_02, dentro de ella crear los archivos geometria.py, aritmetica.py y validaciones.py.
  - En el módulo geometria.py considerar las siguientes funciones: area\_triangulo, area\_cuadrado, area\_rectangulo.
  - En el módulo aritmetica.py considerar las siguientes funciones: suma, resta, multiplicacion, division, potencia y resto de dos números enteros. En las funciones del módulo deberá de haber tratamiento de errores para evitar que se quede bloqueada una funcionalidad, eso incluye la división entre cero.
  - En el módulo de validaciones.py considerar la función que solo pueden ingresar números, los cuales pasarán como argumentos en las funciones del módulo de geometría y aritmética.

Crear un archivo principal.py desde el cual debe llamar a los tres módulos: geometría, aritmética y validaciones, deberá ejecutar las diversas funciones que cada uno de estos tiene.
2. Crear una carpeta propuesto2, en la cual desarrolle un programa que permita cargar una lista con 10 valores enteros, los cuales deben ser números aleatorios entre 0 y 100. Mostrar la lista obtenida por pantalla. Dividir el programa en dos archivos funciones.py y principal.py (donde se encontrará la ejecución de las funciones).
3. Crear una carpeta propuesto3, en él debe realizar un programa que solicite un valor entero. Después mostrar por pantalla la raíz cuadrada, raíz cúbica, el valor elevado al cuadrado y al cubo de dicho número. (Utilizar el módulo math de python). Dividir el programa en dos archivos funciones\_math.py y principal.py (donde se encontrará la ejecución de las funciones y deberá contar con una función que muestre las opciones de menú como: raíz cuadrada, raíz cúbica, potencia al cuadrado, potencia al cubo). Los resultados debe mostrarlos redondeado a dos decimales. Puede realizar una o cuatro funciones para lo requerido en el archivo funciones.py, controlar errores como raíz cuadrada de un valore negativo. (Usar excepciones).
4. Crear una carpeta propuesto4, en el desarrollar un módulo para validación de contraseñas. Dicho módulo, deberá cumplir con los siguientes criterios de aceptación:
  - La contraseña debe contener un mínimo de 8 caracteres.
  - Una contraseña debe contener al menos una letra minúscula, una letra mayúscula y un número.
  - La contraseña no puede contener espacios en blanco.
  - Contraseña válida, retorna True.
  - Contraseña no válida, retorna el mensaje "La contraseña elegida no es segura".

Dividir el programa en dos archivos validaciones.py y principal.py. Utilice los métodos de string como: islower(), isdigit(), isupper(), isspace().

5. Crear una carpeta propuesto5 y desarrollar un programa que valide una fecha en el formato dd/mm/yyyy, use el módulo datetime. Dividir el programa en dos archivos validaciones.py y principal.py.
6. Crear una carpeta propuesto6 y desarrollar un programa agenda con 3 opciones:
  - I. Opcion 1: Registrar agenda.
  - II. Opcion 2: Listar agenda.
  - III. Opcion 3: Salir.
  - Si el usuario ingresa otra opción, el sistema debe de volver a mostrar las opciones.
  - Si el usuario ingresa el valor de 1, el programa debe solicitar:
    - ✓ Registrar nombre (considerar la primera letra del nombre en mayúscula, en caso de que se ingrese en minúscula el texto y en caso el texto se encuentre vacío el programa debe de volver a solicitar).
    - ✓ Registrar dirección (considerar la primera letra en mayúscula) y número de teléfono o celular (validar que solo se ingrese números). Al finalizar el registro, este se debe de almacenar en un diccionario.
  - Una vez que el usuario termine de realizar el registro se debe de volver a mostrar las opciones.
  - Si el usuario ingresa a la opción 2 le debe de mostrar la lista con los registros ingresados. Una vez que el usuario termine de mostrar el listado se debe de volver a mostrar las opciones.
  - Si el usuario ingresa la opción 3 la aplicación debe de finalizar.

Dividir el programa en dos archivos validaciones.py y principal.py.

## VI

### CUESTIONARIO

1. ¿Qué es un módulo?
2. ¿Qué es el diseño modular?
3. ¿Qué ventajas ofrece el uso de módulos?
4. ¿Para qué son útiles los módulos?
5. ¿Cómo hacemos uso de un módulo?
6. ¿De qué forma se importa un módulo?
7. ¿Cómo se hace la importación de un módulo con import?
8. ¿Cómo se hace la importación con from?
9. ¿Cuándo hacemos uso de import?
10. ¿Cuándo hacemos uso de from para la importación?

## VII

### BIBLIOGRAFIA Y REFERENCIAS

- [1] C. N. Academy, «Fundamentos de Python 2,» 2023.
- [2] Luz Nolasco Valenzuela, Jorge Nolasco Valenzuela, Javier Gamboa Cruzado, Fundamentos de Programación con Python 3, Lima: Empresa Editora Macro EIRL, 2020.
- [3] S. CHAZALLET, Python 3 Los fundamentos del lenguaje (3ª edición), Éditions ENI, 2020.