

# Desarrollo con Python



# ¿QUIEN ES SLPROCES?

Satisfacer a nuestros clientes con las mejores soluciones en consultoría y capacitación, somos Profesionales, contamos con conocimiento y flexibilidad, en busca de mejora continua, seguimiento personalizado del proceso de capacitación basado en métricas y un informe con sugerencias para su continuidad.

## Desarrollo Organizacional



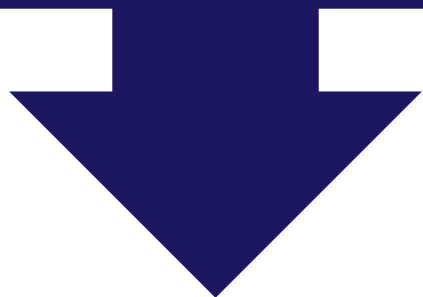
**Seguridad Industrial y  
Protección Civil**



**Tecnologías de la  
información e  
innovación**



# INICIAMOS



## NUESTRO CURSO

### Desarrollo con Python

.



# Módulo 1

## Introducción a Python

## ¿Qué es Python?

Es un lenguaje de programación de alto nivel procesado por un interprete para producir un resultado dado. Fue desarrollado por guido van rossum en los ochentas y noventas.

## ¿Para qué se usa python?

En este lenguaje multipropósito podemos desarrollar aplicaciones:

1. Web
2. Escritorio
3. Machine learning
4. Ciencia de datos

## Características de Python

1. Es gratuito
2. Es fácil de aprender
3. Tiene una larga lista de módulos
4. Es portable
5. Es interpretado
6. Es un lenguaje de alto nivel
7. Es orientado a objetos y estructurado
8. Es multi facetico

## Entornos de desarrollo

### **PyCharm** (de JetBrains)

- **Versión Community:** Gratis, ideal para Python puro.
- **Versión Professional:** De pago, con soporte para frameworks web y bases de datos.



# Entornos de desarrollo

## **Visual Studio Code**

- Editor ligero con extensiones, como la de Python (de Microsoft).
- Soporta linting, autocompletado, Jupyter, debugging y más.
- Muy popular y personalizable.

## Software a usar

- Python 3.13  
[Download Python | Python.org](#)
- Visual Studio Code  
[Download Visual Studio Code - Mac, Linux, Windows](#)
- PostgreSQL  
[PostgreSQL: Downloads](#)

## Consola interactiva

La consola interactiva de Python (también llamada intérprete interactivo o REPL) es un entorno donde se puede escribir y ejecutar código Python línea por línea, de forma inmediata.

REPL (Read–Eval–Print Loop)

## Ejecución de scripts

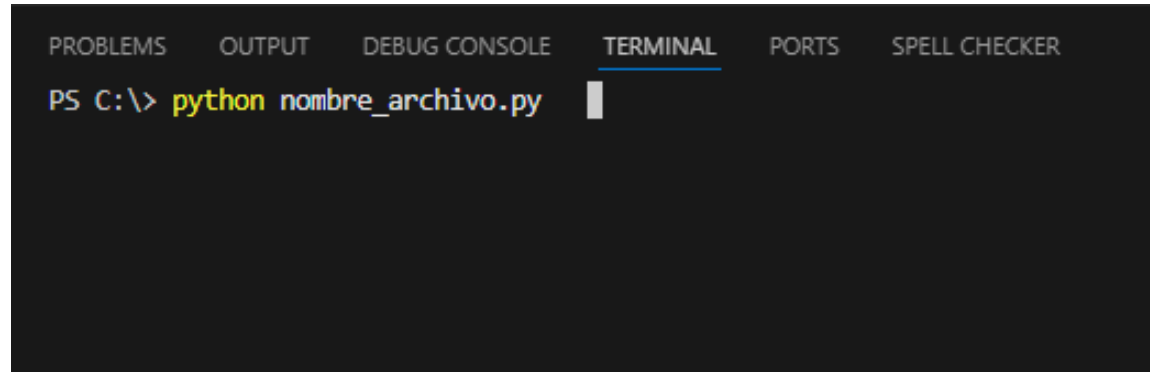
Un script de Python es simplemente un archivo de texto que contiene código Python y que se puede ejecutar para realizar una tarea o conjunto de instrucciones.

- Tiene extensión .py
- Contiene instrucciones que se ejecutan de arriba hacia abajo.
- Puede tener funciones, estructuras de control (if, for, etc.), llamadas a módulos, etc.

## Ejecución de scripts

Para ejecutar un script:

- En Windows se puede usar "Símbolo del sistema" o "PowerShell"; en macOS o Linux, puede usar la aplicación "Terminal".
- Se debe navegar hasta la carpeta donde se encuentra el archivo .py y ejecutar el archivo.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SPELL CHECKER
PS C:\> python nombre_archivo.py
```

## ¿Qué es una librería?

Una librería responde al conjunto de funcionalidades que permiten al usuario llevar a cabo nuevas tareas que antes no se podían realizar.

Cualquier archivo con código Python reutilizable se conoce como módulo o librería.

## ¿Librerías más populares?

1. Matplotlib
2. Bokeh
3. Scikit-Learn
4. Pandas
5. NumPy

# PIP

pip es un sistema de gestión de paquetes utilizado para instalar y administrar paquetes de software escritos en Python.



## 1. Instalar PIP

```
c:> python -m ensurepip --upgrade  
c:> python -m pip install --upgrade pip
```

## 2. Instalar un paquete

```
c:> python -m pip install pandas  
c:> python -m pip install pandas=1.5.1
```

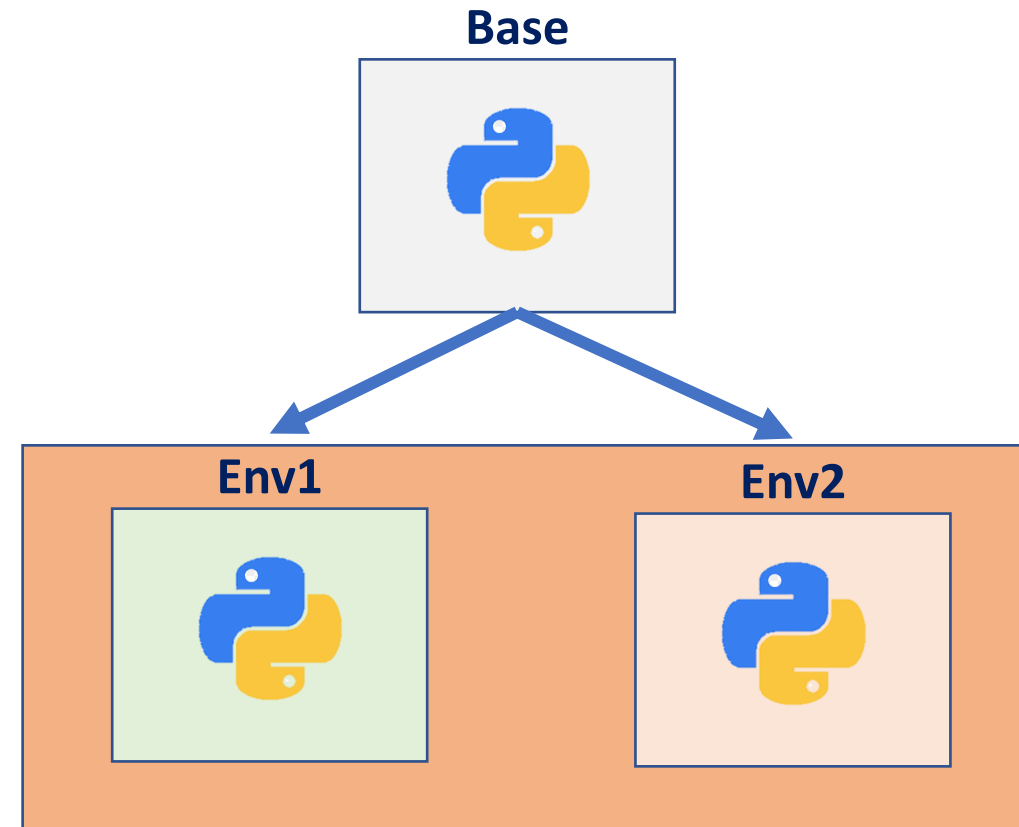
## 3. Listar paquetes o información del paquete

```
c:> python -m pip list  
c:> python -m pip show pandas
```

## 4. Eliminar un paquete

```
c:> python -m pip uninstall pandas
```

Un entorno aislado de ejecución que permite a los usuarios de Python y a las aplicaciones instalar y actualizar paquetes de distribución de Python sin interferir con el comportamiento de otras aplicaciones de Python en el mismo sistema.



## 1. Crear un ambiente virtual “A” y activarlo

```
c:> python -m venv [nombre_ambiente_A]
```

```
c:> \[nombre_ambiente_A]\scripts\activate.bat
```

## 2. Crear un ambiente virtual “B” y activarlo

```
c:> python -m venv [nombre_ambiente_B]
```

```
c:> \[nombre_ambiente_B]\scripts\activate.bat
```

## 3. Instalar un mismo paquete en ambos ambientes con diferente versión



# Módulo 2

## Generalidades del lenguaje

# Tipos de variables

## Enteras

```
anio: int = 2023  
dia = 20
```

```
print(anio)  
print(dia)
```

```
anio = "2023"  
dia = "20"
```

```
print(anio)  
print(dia)
```

# Tipos de variables

## Flotantes

```
pi: int = 3.14
estatura = 1.85

print(pi)
print(estatura)

pi = "3.14"
estatura = "1.85"

print(pi)
print(estatura)
```

# Tipos de variables

## Boleanas

```
is_cold: bool = True  
is_hot = False  
  
print(is_cold)  
print(is_hot)
```

# Tipos de variables

## Cadenas de texto

```
instructor = "Juan Carlos Trejo"  
curso = 'Python basico'  
- cursos = ""  
1. Pthon  
2. Java  
3. Javascript  
- ""
```

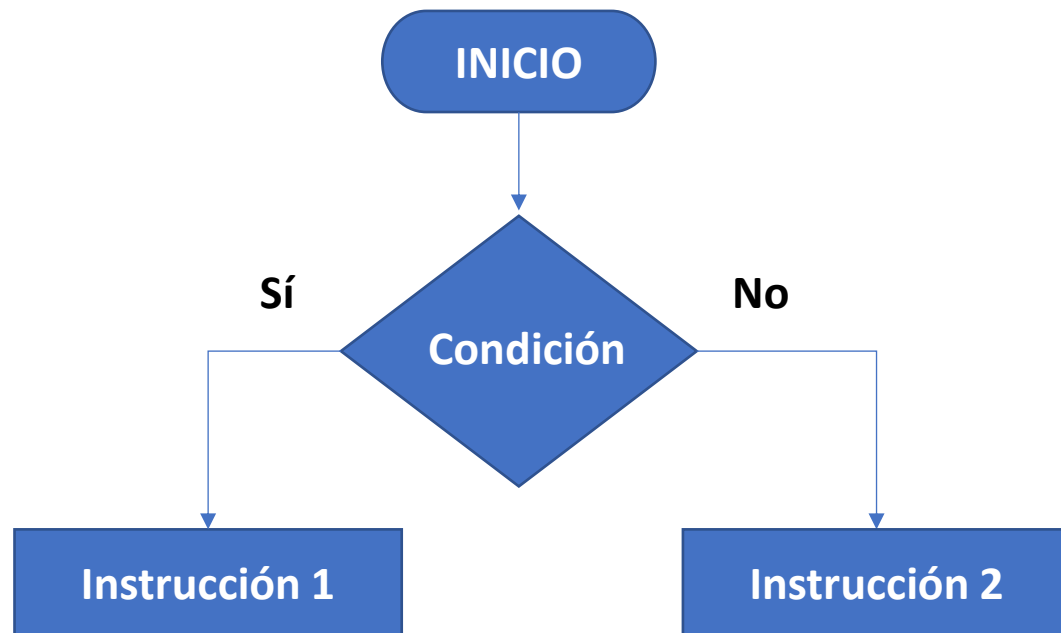


## Asignaciones múltiples

```
a, b, c = 5, 6, 7  
print(c, b, a)
```

```
a, b, c, d = "Juan"  
print(a, b, c, d)
```

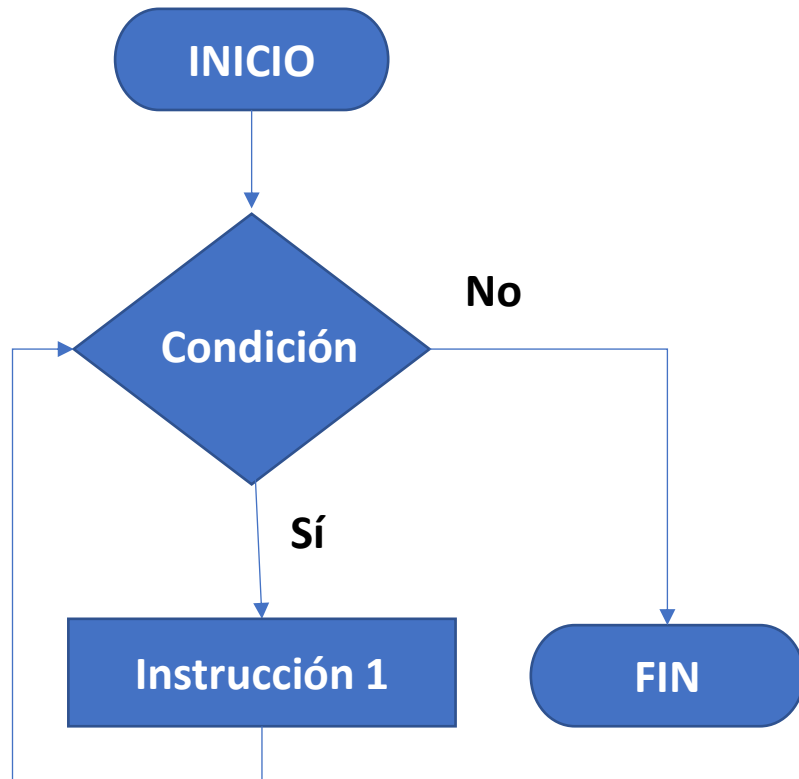
## if - else



```
variable = 1

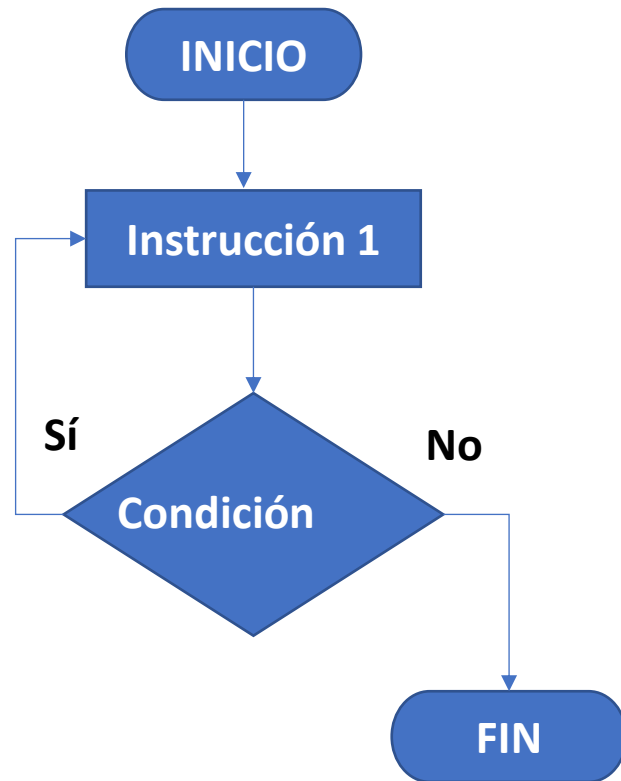
if variable < 8:
    print(variable)
elif variable > 8:
    variable = variable + 1
else:
    variable = variable + 2
```

## while



```
variable = 1  
  
while variable > 8:  
    print(variable)  
    variable = variable + 1
```

## do - while

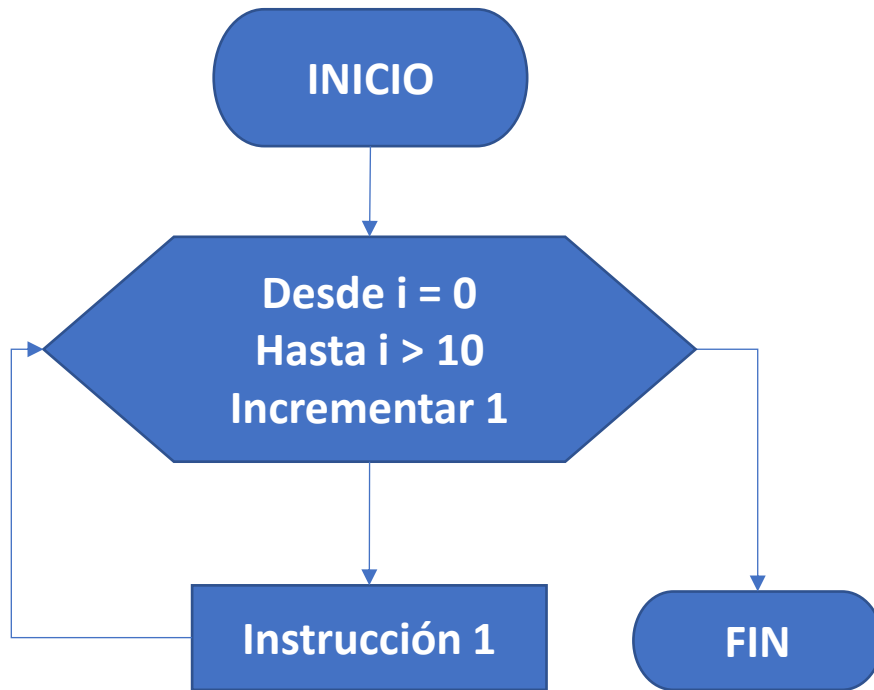


```
variable = 1

while true:
    print(variable)
    variable = variable + 1

    if variable > 8:
        break
```

for



```
variable = 1  
  
for variable in range(10):  
    print(variable)
```

## ¿Qué es una función?

Se puede entender una función como un conjunto de líneas de código que realizan una tarea específica y pueden tomar “Argumentos” para diferentes “Parámetros” que modifiquen su funcionamiento y los datos de salida.

## Sintaxis general

```
def nombre_funcion(argumentos):  
    # código  
    return retorno
```

## Argumentos por posición

```
def resta(a, b):  
    return a-b  
  
resta(5, 3) # 2  
resta(1) # Error!  
resta(1,5,3) # Error!
```

Los argumentos por posición o posicionales son la forma más básica e intuitiva de pasar parámetros. Al tratarse de parámetros posicionales, se interpretará que el primer número es la a y el segundo la b.



## Argumentos por nombre

```
def resta(a, b):  
    return a-b  
  
resta(a=5, b=3) # 2  
resta(b=5, a=3) # -2  
resta(b=5, c=3) # Error!
```

Otra forma de llamar a una función, es usando el nombre del argumento con = y su valor.

## Argumentos por defecto

```
def suma(a=1, b=2, c=0):  
    return a + b + c  
  
suma(5,5,3) # 13  
suma(5,5) # 10  
suma(5) # 7  
suma() # 3
```

Otra forma de llamar a una función, es usando el nombre del argumento con = y su valor.

## Argumentos de longitud variable

```
def suma(numeros):  
    total = 0  
  
    for n in numeros:  
        total += n  
    return total  
  
suma([1,2,3]) # 6
```

Una forma de pasar argumentos de longitud variables sería a través de una estructura de datos, como una lista.

## Argumentos de longitud variable

```
def suma(*numeros):  
    total = 0  
    for n in numeros:  
        total += n  
  
    return total  
  
suma(1, 2, 3) # 6
```

El parámetro especial `*args` en una función se usa para pasar, de forma opcional, un número variable de argumentos posicionales.

- El parámetro recibe los argumentos como una tupla.
- Es un parámetro opcional. Se puede invocar a la función haciendo uso del mismo, o no.
- El número de argumentos al invocar a la función es variable.

## Argumentos de longitud variable

```
def suma(**kwargs):  
    suma = 0;  
    for key, value in kwargs.items():  
        print(key, value)  
        suma += value  
    return suma  
  
suma(a=1, b=2, c=3) # 6
```

El parámetro especial `**kwargs` en una función se usa para pasar, de forma opcional, un número variable de argumentos con nombre.

- El parámetro recibe los argumentos como un diccionario.
- Al tratarse de un diccionario, el orden de los parámetros no importa. Los parámetros se asocian en función de las claves del diccionario.

## Argumentos de longitud variable

```
def suma(**kwargs):  
    suma = 0;  
    for key, value in kwargs.items():  
        print(key, value)  
        suma += value  
    return suma  
  
suma(a=1, b=2, c=3) # 6
```

El parámetro especial `**kwargs` en una función se usa para pasar, de forma opcional, un número variable de argumentos con nombre.

- El parámetro recibe los argumentos como un diccionario.
- Al tratarse de un diccionario, el orden de los parámetros no importa. Los parámetros se asocian en función de las claves del diccionario.

# Excepciones

Las excepciones son errores que ocurren durante la ejecución de un programa. Cuando sucede algo inesperado (como dividir por cero o acceder a un archivo que no existe), Python lanza una excepción

# Cómo manejar excepciones

## try - except

El bloque try ejecuta un bloque de código y en caso de ocurrir un error, el bloque except atrapa ese error.

Se usa la palabra clave except en lugar de catch, pero el propósito es el mismo: atrapar errores durante la ejecución.

```
try:
    # Código que puede fallar
    pass
except TipoDeError:
    # Código para manejar el error
    pass
else:
    # Se ejecuta solo si no hubo error
    pass
finally:
    # Se ejecuta siempre
    pass
```



# Cómo manejar excepciones

## finally - else

El bloque else es útil para poner código que solo debe ejecutarse si todo salió bien en el bloque try

```
try:
    x = int(input("Ingresa un número: "))
    resultado = 10 / x
except ZeroDivisionError:
    print("No se puede dividir entre cero.")
except ValueError:
    print("Entrada no valida.")
else:
    print("El resultado es:", resultado)
```



El bloque else **NO se ejecuta** si ocurre una excepción.

# Cómo manejar excepciones

## finally - else

El bloque finally se usa para realizar tareas de limpieza: cerrar archivos, liberar recursos, cerrar conexiones, etc.

```
try:
    archivo = open("datos.txt")
    contenido = archivo.read()
except FileNotFoundError:
    print("Archivo no encontrado.")
else:
    print("Archivo leído correctamente.")
finally:
    print("Cerrando el archivo")
```



El bloque finally siempre se **ejecuta** ocurra o no una excepción.

# Tipos de excepciones

## Características

- El lenguajes como Java el compilador obliga a manejarla con try-catch o throws.
- En Python todas las excepciones son “unchecked”.
- No es obligatorio atraparlas.
- Se puede lanzar y manejar excepciones libremente, pero si no atrapan y ocurre un error, el programa se detiene

# Jerarquía de excepciones

La jerarquía de excepciones es la forma en que las clases de excepciones están organizadas en una estructura de herencia (como un árbol). Esto permite que se pueda capturar múltiples tipos de errores de manera flexible y específica.

# Jerarquía de excepciones

Excepción	Cuándo ocurre
ZeroDivisionError	División entre cero, como <code>10 / 0</code>
ValueError	Conversión inválida, como <code>int("abc")</code>
TypeError	Operación con tipos incompatibles, como <code>len(5)</code>
IndexError	Índice fuera del rango de una lista, como <code>lista[10]</code>
KeyError	Clave no encontrada en un diccionario
AttributeError	Objeto no tiene el atributo solicitado
NameError	Variable usada sin haber sido definida
FileNotFoundError	Archivo no existe al intentar abrirlo
PermissionError	No se tienen permisos para acceder a un archivo o directorio
IOError	Error genérico de entrada/salida (anterior a Python 3.3)
ImportError	Error al importar un módulo

# Jerarquía de excepciones

Excepción	Cuándo ocurre
ModuleNotFoundError	El módulo no fue encontrado
RuntimeError	Error general que no cae en otras categorías
StopIteration	Un iterador ha terminado (por ejemplo en un for)
RecursionError	Se superó el límite de recursión máxima
AssertionError	Cuando falla una afirmación con assert
Exception	Clase base para la mayoría de las excepciones
BaseException	Clase base de todas las excepciones.

# Lanzar una excepción manualmente

Se puede lanzar (o disparar) una excepción manualmente usando la palabra clave `raise`.

```
edad = -5
if edad < 0:
    raise ValueError("La edad no puede ser negativa.")
```

## Excepciones personalizadas

Las excepciones personalizadas son clases que el mismo desarrollador define para representar errores específicos en el programa, más allá de las que ya vienen integradas.

```
class EdadInvalidaError(Exception):
    def __init__(self, edad, mensaje="Edad no válida"):
        self.edad = edad
        self.mensaje = mensaje
        super().__init__(f"{mensaje}: {edad}")

def registrar_edad(edad):
    if edad < 0:
        raise EdadInvalidaError(edad)
    print("Edad registrada:", edad)

try:
    registrar_edad(-3)
except EdadInvalidaError as e:
    print("Error personalizado:", e)
```



1. (*Triples de Pitágoras*). Un triángulo recto puede tener lados cuyas longitudes sean valores enteros. El conjunto de tres valores enteros para las longitudes de los lados de un triángulo recto se conoce como triple de Pitágoras. Las longitudes de los tres lados deben satisfacer la relación que establece que la suma de los cuadrados de dos lados es igual al cuadrado de la hipotenusa. Escriba una aplicación para encontrar todos los triples de Pitágoras para lado1, lado2, y La hipotenusa, que no sean mayores de 500.
2. (*Adivina el número*). Crear un script que permita jugar a adivinar un numero. El objetivo es adivinar un numero generado aleatoriamente. El usuario debe proponer un numero, si este es mayor, se visualizará un mensaje que diga que el numero introducido es alto. Por el contrario si el numero es menor al numero generado se visualizará un mensaje que diga que el numero propuesto es bajo. De este modo el usuario puede ir acotando el numero hasta adivinarlo. Cuando el numero sea acertado, se visualizará un mensaje alusivo a esto y el numero de intentos hechos.



# Módulo 3

## Programación Orientada a Objetos

# ¿Qué es la POO?

Programación Orientada a Objetos, es un método para estructurar un programa al agrupar propiedades y comportamientos relacionados en objetos individuales.

# Clase

Es la estructura o “molde a partir del cual se podrán instanciar, construir u obtener objetos con las propiedades y métodos de esa misma clase.

```
class Cliente:  
    pass
```

# Atributos de clase

Los atributos son las características individuales que diferencian un objeto de otro y determinan su apariencia, estado u otras cualidades.

```
class Cliente:  
    nombre = ""  
    primer_apellido = ""
```

# Constructor

Al crear o instanciar un objeto se asignan por valores por default a sus propiedades o atributos, ese es el estado inicial de un objeto. Para proporcionar valores iniciales a un objeto utilizamos un constructor.

```
class Cliente:

    nombre = ""
    primer_apellido = ""

    def __init__(self, nombre, primer_apellido):
        self.nombre = nombre
        self.primer_apellido = primer_apellido

#Instanciacion de un objeto
cliente1 = Cliente("Juan", "Hernandez")
print(cliente1)
```

# Encapsulación

Es el principio de ocultar los detalles internos de una clase y permitir que el acceso a sus atributos o métodos se haga solo a través de interfaces definidas, generalmente métodos.

# Encapsulación

Por default, en Python todos los atributos de clase son públicos. Para poder reducir la visibilidad de un atributo, basta con agregar doble guión bajo (\_\_) al principio del nombre del atributo

```
class Cliente:  
  
    def __init__(self, nombre, apellido):  
        self.nombre = nombre #publico  
        self.__apellido = apellido #privado
```



# Métodos setter y getter

Los métodos get y set, son métodos que se asocian a un atributo de la clase y que se usan en las clases para mostrar (get) o modificar (set) el valor de un atributo. El nombre del método siempre será get o set y a continuación el nombre del atributo.

```
class Cliente:  
  
    def __init__(self, nombre):  
        self.nombre = nombre  
  
    def get_nombre(self):  
        return self.nombre  
  
    def set_nombre(self, value):  
        self._nombre = value
```

## Métodos setter y getter

Los métodos get y set, son métodos que se asocian a un atributo de la clase y que se usan en las clases para mostrar (get) o modificar (set) el valor de un atributo. El nombre del método siempre será get o set y a continuación el nombre del atributo.

```
class Cliente:

    def __init__(self, nombre):
        self.nombre = nombre

    @property
    def nombre(self):
        return self._nombre

    @nombre.setter
    def nombre(self, value):
        self._nombre = value
```

# Herencia

Es el mecanismo por el cual una clase permite heredar las características (atributos y métodos) de otra clase. Aprenda más a continuación.

```
class Persona:

    def __init__(self, nombre):
        self.nombre = nombre

class Empleado(Persona):

    def __init__(self, nombre, clave):
        super().__init__(nombre)
        self.clave = clave
```

# Herencia múltiple

Es cuando una clase hereda de más de una clase base al mismo tiempo.

```
class A:
    def metodo_a(self):
        print("Método de A")

class B:
    def metodo_b(self):
        print("Método de B")

class C(A, B):
    pass

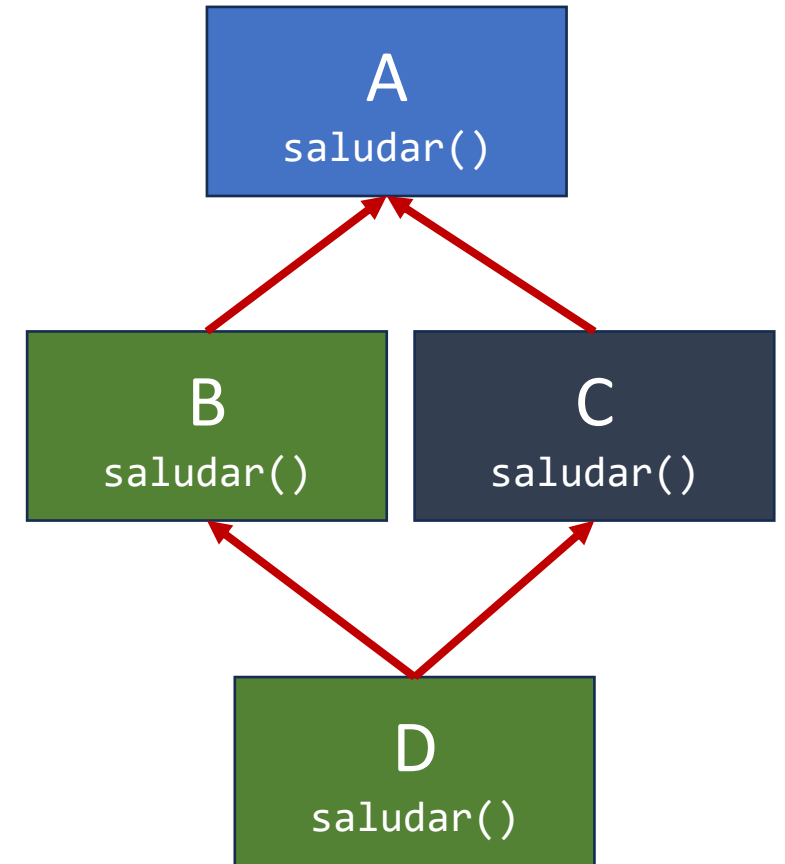
c = C()
c.metodo_a() # Hereda de A
c.metodo_b() # Hereda de B
```

# Herencia múltiple

## Problema del diamante

El problema del diamante (herencia diamante) es una situación que ocurre en lenguajes con herencia múltiple, como Python, cuando una clase hereda de dos clases que a su vez heredan de una misma superclase.

Esto genera ambigüedad: *¿de cuál clase intermedia se debe heredar un método o atributo común?*



# Herencia múltiple

¿Qué pasa si hay métodos con el mismo nombre?

Python sigue un orden para decidir de qué clase heredar en caso de conflictos. Este orden se llama MRO (Method Resolution Order) y usará el método de la primera clase en el orden de herencia.

```
class A:
    def saludar(self):
        print("Hola desde A")

class B:
    def saludar(self):
        print("Hola desde B")

class C(A, B):
    pass

c = C()
c.saludar() # "Hola desde A"
```

# Composición

La composición en POO es un principio donde una clase contiene objetos de otras clases como parte de su estructura, en lugar de heredar de ellas.

```
class Motor:
    def arrancar(self):
        print("Motor en marcha")

class Auto:
    def __init__(self):
        self.motor = Motor() # Composición

    def encender(self):
        self.motor.arrancar()
        print("Auto encendido")
```

## Ejercicio

1. Crear una clase llamada Persona. Sus atributos son: nombre, edad e id. Construir los siguientes métodos para la clase:

- Un constructor, donde los datos pueden estar vacíos.
- Los setters y getters para cada uno de los atributos.
- El método mostrar(): Muestra los datos de la persona.
- El esMayorDeEdad(): Devuelve un valor booleano indicando si es mayor de edad.



## Ejercicio

2. Crea una clase llamada Cuenta que tendrá los siguientes atributos: titular (que es una persona) y cantidad (puede tener decimales). El titular será obligatorio y la cantidad es opcional. Construye los siguientes métodos para la clase:

- Un constructor, donde los datos pueden estar vacíos.
- Los setters y getters para cada uno de los atributos. El atributo no se puede modificar directamente, sólo ingresando o retirando dinero.
- El método mostrar(): Muestra los datos de la cuenta.
- El método depositar(cantidad): se ingresa una cantidad a la cuenta, si la cantidad introducida es negativa, no se hará nada.
- El método retirar(cantidad): se retira una cantidad a la cuenta. La cuenta puede estar en números rojos.

# Módulo 4

## Estructuras de datos

# ¿Qué es una estructura de datos?

Una estructura de datos modela una colección de datos, como una lista de números, una fila en una hoja de cálculo o un registro en una base de datos.

- Listas
- Tuplas
- Diccionarios

# Listas

Las listas contienen elementos indexados por números enteros, comenzando con 0. Las listas son mutables, lo que significa que puede cambiar el valor en un índice incluso después de que se haya creado la lista.

["Este", "es", "un", "curso", "de", "Python"]					
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

# Listas

## Operaciones básicas

<code>len([1, 2, 3])</code>	3	Longitud
<code>[1, 2, 3] + [4, 5, 6]</code>	<code>[1, 2, 3, 4, 5, 6]</code>	Concatenación
<code>["Saludos"] * 2</code>	<code>["Saludos", "Saludos"]</code>	Repetición
<code>3 in [1, 2, 3]</code>	True	Miembro
<code>for x in [1, 2, 3]: print(x)</code>	1 2 3	Iteración

# Listas

## Operaciones básicas

<code>list.append(obj)</code>	Agrega objeto obj a la lista.
<code>list.count(obj)</code>	Devuelve el recuento de cuántas veces aparece obj en la lista.
<code>list.index(obj)</code>	Devuelve el índice más bajo en la lista que aparece obj.
<code>list.insert(index, obj)</code>	Inserta el objeto obj en la lista en el índice dado.
<code>list.remove(obj)</code>	Remueve el objeto obj de la lista.
<code>list.reverse()</code>	Invierte los objetos de la lista en su lugar
<code>list.sort()</code>	Ordena los objetos de la lista.

# Tuplas

Son un tipo o estructura de datos que permite almacenar datos de una manera muy parecida a las listas, con la salvedad de que son inmutables.

(“Este”, “es”, “un”, “curso”, “de”, “Python”)

0

1

2

3

4

5

-6

-5

-4

-3

-2

-1

# Tuplas

## Operaciones básicas

<code>len((1, 2, 3))</code>	3	Longitud
<code>(1, 2, 3) + (4, 5, 6)</code>	<code>(1, 2, 3, 4, 5, 6)</code>	Concatenación
<code>("Saludos") * 2</code>	<code>("Saludos", "Saludos")</code>	Repetición
<code>5 in (1, 2, 3)</code>	False	Miembro
<code>for x in (1, 2, 3): print(x)</code>	1 2 3	Iteración



# Tuplas

## Operaciones básicas

<code>len()</code>	Da la longitud total de la lista.
<code>max()</code>	Devuelve el elemento de la lista con el valor máximo.
<code>min()</code>	Devuelve el elemento de la lista con valor mínimo.
<code>list()</code>	Convierte una tupla en lista.
<code>tuple()</code>	Convierte una lista en tupla.

# Comprehension

Una lista o tupla por comprensión es una forma abreviada de un ciclo for.

```
>>> numbers = (1, 2, 3, 4, 5)
>>> squares = [num**2 for num in numbers]
```

```
>>> str_numbers = ["1.5", "2.3", "5.25"]
>>> float_numbers = [float(value) for value in str_numbers]
```

## Ejercicios

1. Dada una lista con elementos repetidos, devolver una nueva lista con los elementos únicos, pero manteniendo el orden original.
2. Dada una lista de números enteros (positivos y negativos), encontrar la **sublista contigua** con la **mayor suma** posible.
3. Dada una lista de números enteros, agrupar los que sean consecutivos en sublistas.
4. Escribir una función que reciba una lista y devuelva **todas las permutaciones posibles** de sus elementos (ordenaciones distintas).
5. Dada una lista de enteros positivos y un número objetivo, devolver **todas las combinaciones posibles** de números que sumen exactamente ese valor.

# Diccionarios

Los diccionarios de Python, como listas y tuplas, almacenan una colección de objetos. Sin embargo, en lugar de almacenar objetos en una secuencia, los diccionarios contienen información en pares de datos llamados pares clave-valor.

- Las claves son únicas.
- Los valores pueden ser de cualquier tipo.
- Se pueden componer diccionarios de diccionarios.
- No existe límite de espacio para un diccionario.

```
>>> capitals = {  
    "California": "Sacramento",  
    "New York": "Albany",  
    "Texas": "Austin",  
}
```

# Diccionarios

## Operaciones básicas

<code>dict.clear()</code>	Remueve todos los elementos del diccionario.
<code>dict.copy()</code>	Regresa una copia del diccionario.
<code>dict.get(obj)</code>	Devuelve el valor asociado a una clave o un valor por defecto si no se encuentra la clave.
<code>dict.items()</code>	Regresa una lista que contiene una tupla por cada par de clave-valor.
<code>dict.keys()</code>	Regresa una lista con las claves del diccionario.
<code>dict.pop(obj)</code>	Remueve un elemento con la clave especificada.
<code>dict.values()</code>	Regresa una lista con los valores del diccionario.
<code>dict.update()</code>	Actualiza el diccionario con clave-valor.

## Ejercicios

1. El directorio de los empleados de una empresa está organizado en una cadena de texto como la de más abajo, donde cada línea contiene la información del nombre, email, teléfono, id, y el descuento que se le aplica. Las líneas se separan con el carácter de cambio de línea \n y la primera línea contiene los nombres de los campos con la información contenida en el directorio.

```
"id;nombre;email;teléfono;descuento\n01234567L;Luis  
González;luisgonzalez@mail.com;656343576;12.5\n71476342J;Macarena  
Ramírez;macarena@mail.com;692839321;8\n63823376M;Juan José  
Martínez;juanjo@mail.com;664888233;5.2\n98376547F;Carmen  
Sánchez;carmen@mail.com;667677855;15.7"
```

Escribir un programa que genere un diccionario con la información del directorio, donde cada elemento corresponda a un cliente y tenga por clave su id y por valor otro diccionario con el resto de la información del cliente. Los diccionarios con la información de cada cliente tendrán como claves los nombres de los campos y como valores la información de cada cliente correspondientes a los campos.

## Ejercicios

2. Dado el siguiente diccionario. Resolver los siguientes ejercicios:
- a) Obtener la edad promedio de los usuarios.
  - b) Verificar cuántos usuarios tienen la habilidad "base\_de\_datos" activada.
  - c) Crear una lista con los nombres de los usuarios que tienen el curso de "Curso de Python" y la habilidad "programacion" activada.
  - d) Agregar una nueva medalla "avanzado" para cada usuario que tenga la habilidad "base\_de\_datos" activada.
  - e) Mostrar el nombre de los usuarios y la cantidad de medallas que tienen.

```
usuarios = {  
    '1': {  
        'nombre': 'Juan',  
        'edad' : 23,  
        'curso': 'Curso de Python',  
        'skills':{  
            'programacion' : True,  
            'base_de_datos': False  
        },  
        'medallas' : ['básico', 'intermedio']  
    },  
    '2': {  
        'nombre': 'Carlos',  
        'edad' : 30,  
        'curso': 'Curso de Java',  
        'skills':{  
            'programacion' : True,  
            'base_de_datos': True  
        },  
        'medallas' : ['básico']  
    }  
}
```

# Collection module

En Python, el módulo `collections` es una biblioteca estándar que proporciona tipos de datos adicionales más avanzados que las estructuras de datos básicas como `list`, `dict`, `set` y `tuple`.

Estos tipos especializados están diseñados para resolver problemas comunes de forma más eficiente y legible.



# Collection module

## namedtuple

Es una subclase de tupla que permite asignar nombres a los campos, lo que hace que los elementos sean accesibles tanto por posición como por nombre, mejorando la legibilidad del código.

```
from collections import namedtuple

# Definimos una tupla llamada Persona con campos "nombre"
# y "edad"
Persona = namedtuple('Persona', ['nombre', 'edad'])

# Creamos una persona
juan = Persona(nombre='Juan', edad=30)

print(juan.nombre) # Accedemos por nombre -> Juan
print(juan.edad)   # Accedemos por nombre -> 30
print(juan[0])     # También se puede por índice -> Juan
```

# Collection module

## deque

Es una estructura de datos tipo cola de doble extremo que permite agregar y eliminar elementos rápidamente desde ambos extremos (inicio y final).

```
from collections import deque

# Creamos una cola con algunos elementos
cola = deque(['a', 'b', 'c'])

cola.append('d')      # Agregar al final
cola.appendleft('z')  # Agregar al inicio

print(cola)  # deque(['z', 'a', 'b', 'c', 'd'])

cola.pop()           # Eliminar del final
cola.popleft()       # Eliminar del inicio

print(cola)  # ¿?
```

# Collection module

## Counter

Es una subclase de diccionario diseñada para contar la cantidad de veces que aparece cada elemento en una colección como una lista, cadena o cualquier iterable.

```
from collections import Counter

# Contamos letras en una palabra
letras = Counter('banana')

print(letras)           # ¿?
print(letras['a'])       # ¿?
print(letras.most_common(1)) # ¿?
```

# Collection module

## defaultdict

Es una subclase de diccionario que proporciona un valor predeterminado automáticamente para claves inexistentes, evitando errores al acceder a ellas por primera vez.

```
from collections import defaultdict

# Los valores serán enteros con valor inicial 0
conteo = defaultdict(int)

palabra = 'banana'
for letra in palabra:
    conteo[letra] += 1

print(conteo) # ¿?
```



# Módulo 5

## Programación funcional

# Programación funcional

Es un paradigma para escribir código usando funciones que no cambian los estados, sino que toman datos, los procesan y devuelven nuevos datos o estados.

# Programación funcional

**Evitar modificar datos:** En lugar de cambiar una lista, se crea una nueva lista con los cambios.

```
# Imperativo (modifica la lista)
numeros = [1, 2, 3]
for i in range(len(numeros)):
    numeros[i] = numeros[i] * 2

print(numeros)  # [2, 4, 6]

# Funcional (crea una nueva lista)
numeros = [1, 2, 3]
nueva_lista = list(map(lambda x: x * 2, numeros))

print(nueva_lista)  # [2, 4, 6]
print(numeros)      # [1, 2, 3] ← sigue igual
```

# Programación funcional

**Usar funciones como materia prima:** Las funciones se usan para transformar datos, como una máquina que entra algo y saca un resultado.

```
# Una funcion que toma otra funcion
def aplicar_operacion(lista, funcion):
    return [funcion(x) for x in lista]

# Funcion para elevar al cuadrado
def cuadrado(x):
    return x * x

# Funcion para duplicar
def duplicar(x):
    return x * 2

numeros = [1, 2, 3]

# [1, 4, 9]
print(aplicar_operacion(numeros, cuadrado))
# [2, 4, 6]
print(aplicar_operacion(numeros, duplicar))
```



# Programación funcional

**No depender del estado:** Una función siempre da el mismo resultado si se le da los mismos datos, sin importar lo que pase fuera de ella.

```
# No funcional, depende de una variable
factor = 3

def multiplicar(x):
    return x * factor

print(multiplicar(2))

# Funcional, solo usa lo que se le pasa
def multiplicar_puro(x, factor):
    return x * factor

print(multiplicar_puro(2, 3))
print(multiplicar_puro(2, 4))
```

# Funciones

## de orden superior

Una función de orden es una función que:

- **Recibe una o más funciones como argumento**
- Devuelve otra función como resultado

```
# Funcion de orden superior
def aplicar(f, x):
    return f(x)

# Funcion normal
def cuadrado(n):
    return n * n

print(aplicar(cuadrado, 5))
```

# Funciones

## de orden superior

Una función de orden es una función que:

- Recibe una o más funciones como argumento
- **Devuelve otra función como resultado**

```
# Funcion de orden superior
def saludar_persona(nombre):
    def saludo():
        return f"Hola, {nombre}"
    return saludo

f = saludar_persona("Juan")
print(f()) # Hola, Juan
```

# Funciones

## lambda

Una función lambda es una función anónima, es decir, una función sin nombre, que se define en una sola línea.

```
# Funcion normal
def nombre(argumentos):
    return expresion

# Funcion lambda
lambda argumentos: expresion
```

# Funciones

## lambda

Una función lambda es una función anónima, es decir, una función sin nombre, que se define en una sola línea.

```
# Funcion normal
def cuadrado(x):
    return x * x

# Funcion lambda
cuadrado = lambda x: x * x

print(cuadrado(4)) # 16
```

## Funciones incorporadas (built-in functions)

map()	Aplica una función a cada elemento de una secuencia
filter()	Filtra elementos que cumplen una condición
reduce()	Reduce una secuencia a un solo valor
zip()	Une elementos de varias listas en pares
enumerate()	Devuelve índice y valor de una lista
all()	Devuelve True si todos son verdaderos
any()	Devuelve True si al menos uno es verdadero

## Ejercicios

1. Dada una lista de números, crear una nueva lista con cada número multiplicado por 2 usando map.
2. Dada una lista de palabras, filtrar y conserva solo las que tienen 4 letras o menos usando filter
3. Dada una lista de números, crear una nueva lista con el cuadrado de los que son pares, usando filter y map.
4. Dada una lista de palabras, contar cuántas empiezan con vocal (a, e, i, o, u) usando filter y len.
5. Dada una lista de números enteros, sumar todos los dígitos de los números que son impares. Usa filter, map y reduce

# Decoradores

Un decorador es una función que recibe otra función como argumento y devuelve una nueva función con comportamiento modificado o extendido y que su función es para añadir funcionalidades a funciones ya existentes y reutilización de código.

```
def mi_decorador(func):  
    def nueva_funcion():  
        print("Antes de la función")  
        func()  
        print("Después de la función")  
    return nueva_funcion  
  
@mi_decorador  
def saludar():  
    print("Hola, mundo")  
  
saludar()
```



# Decoradores

Un decorador con argumentos requiere un nivel más de anidación. Primero se ejecuta con los argumentos del decorador, luego retorna un decorador normal.

```
def repetir(n):  
    def decorador(func):  
        def envoltura(*args, **kwargs):  
            for _ in range(n):  
                func(*args, **kwargs)  
            return envoltura  
        return decorador  
  
@repetir(3)  
def saludar():  
    print("Hola!")  
  
saludar()
```

# Decoradores

Los decoradores anidados ocurren cuando se aplican múltiples decoradores a una misma función.

En otras palabras, una función es decorada por varios decoradores, uno encima del otro.

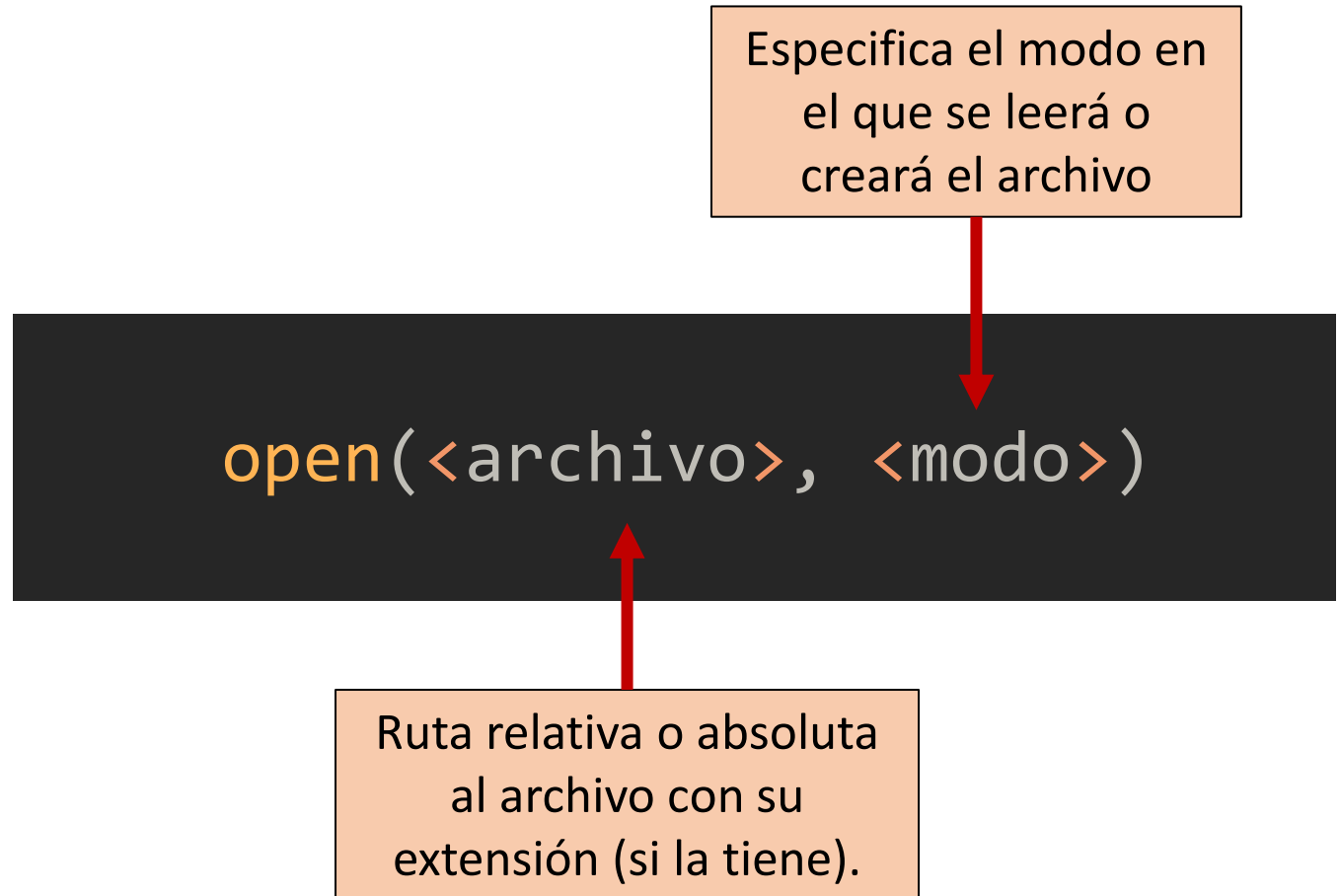
```
def decorador_1(func):  
    def envoltura():  
        print("Decorador 1 antes")  
        func()  
        print("Decorador 1 despues")  
    return envoltura  
  
def decorador_2(func):  
    def envoltura():  
        print("Decorador 2 antes")  
        func()  
        print("Decorador 2 despues")  
    return envoltura  
  
@decorador_1  
@decorador_2  
def saludar():  
    print("Hola!")  
  
saludar()
```

# Módulo 6

## Manipulación de archivos

# La función open

La función `open()`, es una función incorporada (built-in) que abre y/o crea un archivo y permite que un script tenga acceso a él.



# Modos de apertura

Modo	Descripción
'r'	Abierto para lectura (por defecto)
'w'	Abierto para escritura, truncando primero el fichero
'x'	Abierto para creación en exclusiva, falla si el fichero ya existe
'a'	Abierto para escritura, añadiendo al final del fichero si este existe
'+'	Abierto para actualizar (lectura y escritura).
't'	Abrir el archivo en modo texto (por defecto).
'b'	Abrir el archivo en modo binario.

## ¿Qué es un objeto de tipo archivo?

Un objeto que expone una API orientada a archivos (con métodos como `read()` o `write()`) al objeto subyacente.

En pocas palabras, la función `open` devuelve un objeto archivo.

```
archivo = open("alumnos.txt", "r")
```

Función	Descripción
<code>read()</code>	Retorna todo el contenido del archivo como una cadena de caracteres.
<code>close()</code>	Cierra el archivo en uso.
<code>readline()</code>	Lee una línea del archivo. Mantiene el carácter de salto de línea ( <code>\n</code> ) al final de la cadena de caracteres.
<code>readlines()</code>	Retorna una lista que contiene todas las líneas del archivo como elementos individuales de la lista (cadenas de caracteres).
<code>write(line)</code>	Permite agregar una línea al contenido de un archivo.
<code>writelines([lines])</code>	Permite agregar varias líneas al contenido de un archivo a través de un lista.
<code>os.remove(arch)</code>	Esta función toma la ubicación o la ruta (path) del archivo como argumento y elimina el archivo automáticamente.
<code>os.mkdir(name)</code>	Crea un directorio en la ruta especificada.
<code>os.rename(r1, r2)</code>	Renombra y/o mueve un archivo a una ubicación dada.

## Leer un archivo

```
archivo = open("c:\\archivo1.txt", "r")  
print(archivo.read())  
archivo.close()
```

- Abrir un archivo en modo lectura e imprimir su contenido.

```
archivo = open("archivo.txt", "r")  
print(archivo.readline())  
archivo.close()
```

- Abrir un archivo en modo lectura e imprimir una línea del contenido del archivo.



## Leer un archivo

```
archivo = open("c:\\archivo1.txt", "r")  
print(archivo.readlines())  
archivo.close()
```

- Abrir un archivo en modo lectura e imprimir su contenido a través de una lista.

## Modificar un archivo

```
archivo = open("archivo.txt", "x")  
archivo.write("Linea nueva")  
archivo.close()
```

```
archivo = open("archivo.txt", "a")  
archivo.write("Agregar otra linea nueva")  
archivo.flush()  
archivo.close()
```

- Abrir un archivo en modo creación y escribir una línea nueva al archivo.
- Abrir un archivo en modo de agregado y escribir una nueva línea en el mismo archivo. ¿Cómo queda el nuevo contenido del archivo?

## Modificar un archivo

```
archivo = open("archivo.txt", "a")  
print(archivo.read())  
  
archivo = open("archivo.txt", "w")  
print(archivo.read())
```

- ¿Qué sucede al ejecutar el siguiente código?  
¿Por qué? ¿Solución?

## Ejercicio

1. Realizar una clase que administre una agenda. Se debe almacenar para cada contacto el nombre, primer apellido, el teléfono y el email. Además deberá mostrar un menú con las siguientes opciones:

- Añadir contacto
- Listar contactos.
- Buscar contacto por uno o mas atributos de la clase.
- Editar contacto.
- Borrar contacto.
- Salir

La información del contacto se deberá guardar en un archivo de texto plano. Cada contacto se guardará en una línea y tendrá el formato:

*nombre|primer\_apellido|email|telefono*

## Ejercicio

2. De acuerdo al siguiente diccionario, transformar los datos para que puedan ser guardados en un archivo 'csv'

```
data = {  
    'country1': {  
        'name': 'Albania',  
        'area': 28748,  
        'country_code': 'AL'  
    },  
    'country2': {  
        'name': 'USA',  
        'area': 9846,  
        'country_code': 'US'  
    },  
    'country3': {  
        'name': 'Mexico',  
        'area': 52,  
        'country_code': 'MX'  
    }  
}
```

## Ejercicio

3. De acuerdo a la siguiente información en formato csv, transformar los datos para que puedan ser guardados en un archivo 'json'

```
nombre,primer_apellido,telefono,calle,colonia  
Juan,García,5551234567,Av. Reforma 123,Centro  
María,López,5552345678,Calle Juárez 45,Del Valle  
Luis,Hernández,5553456789,Insurgentes Sur 300,Nápoles  
Ana,Ramírez,5554567890,Calle Morelos 78,Condesa  
Carlos,Pérez,5555678901,Av. Universidad 567,Coyoacán  
Lucía,Morales,5556789012,Calle Hidalgo 90,Polanco  
Miguel,Ruiz,5557890123,Av. Tláhuac 12,Iztapalapa  
Fernanda,Torres,5558901234,Calle Nezahualcóyotl 67,Tlalpan  
Jorge,Sánchez,5559012345,Av. Patriotismo 234,San Pedro de los Pinos  
Patricia,Cruz,5550123456,Calle Emiliano Zapata 11,Xochimilco
```

# Módulo 7

## Conexión a base de datos

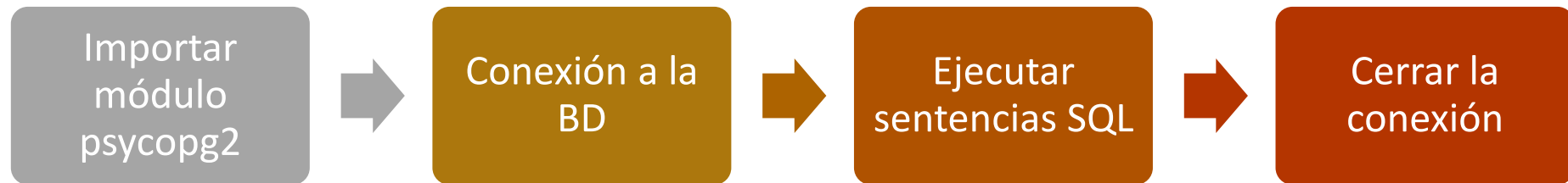
# Introducción

Una base de datos es un sistema estructurado para almacenar datos.

Python viene con una base de datos SQL liviana llamada SQLite que es perfecta para aprender a trabajar con bases de datos.



## Flujo de conexión



# Cursor

Nombre	Apellido	Edad
		55

El término cursor se refiere a un objeto que se usa para obtener resultados de una consulta de la base de datos una fila a la vez.

Aunque los objetos `psycopg2` se usan para esta operación, también hacen mucho más de lo que normalmente se espera de un cursor.

Nueva fila

## Métodos del objeto cursor

first()	Mueve el cursor a la primera posición.
last()	Mueve el cursor a la ultima posición.
next()	Mueve el cursor a la siguiente posición.
prev()	Mueve el cursor a la anterior posición.
execute(str)	Ejecuta una sentencia SQL.
fetchone()	Devuelve la siguiente del conjunto de resultados.
fetchall()	Devuelve todas las filas que la consulta haya capturado.

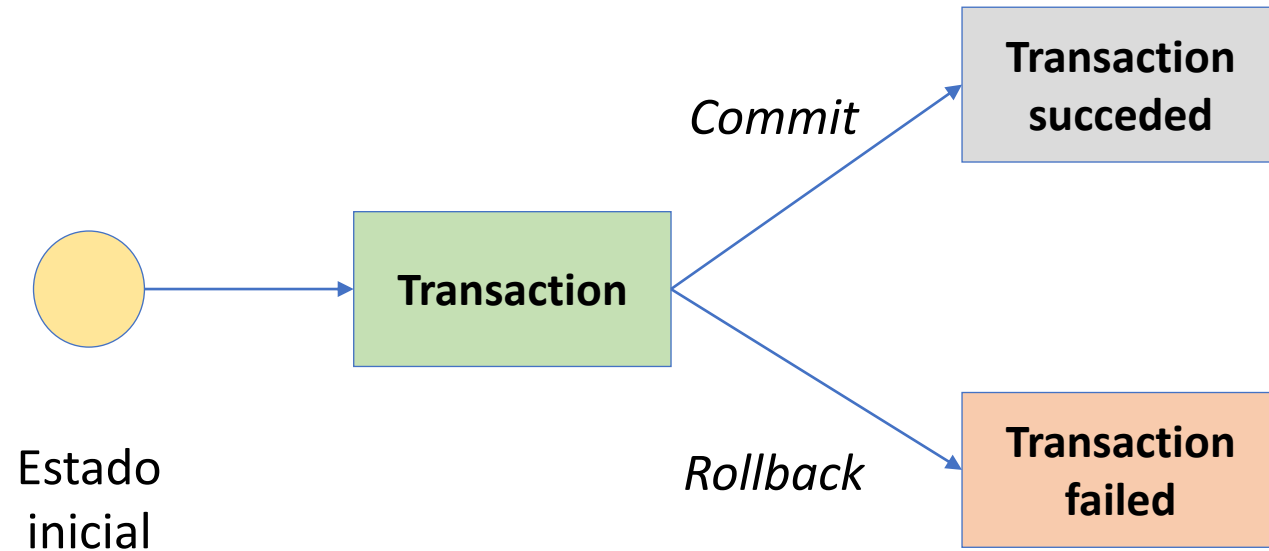
# Transacción

Una transacción es una unidad de trabajo que consiste en una o más operaciones que se deben ejecutar de forma completa o no ejecutarse en absoluto, garantizando la integridad de los datos.

Propiedad	
Atomacidad	Todas las operaciones se completan o ninguna se realiza.
Consistencia	El estado de la base de datos sigue siendo válido antes y después.
Isolamiento	Las transacciones independientes no interfieren entre sí.
Durabilidad	Una vez confirmada, la transacción es permanente, incluso tras un fallo.

# Transacción

Una transacción es una unidad de trabajo que consiste en una o más operaciones que se deben ejecutar de forma completa o no ejecutarse en absoluto, garantizando la integridad de los datos.





¡Gracias!

