

Apuntes de Python

SINTAXIS POR NIVELES

DAVID SUÁREZ CORDÓN

Sintaxis básica:

00. Comentarios en Python

01. Variables

02. Operaciones

03. Strings

04. Listas

05. Tuplas

06. Sets

07. Diccionarios

08. Condicionales

09. Bucles

10. Funciones

11. Clases

12. Excepciones

13. Módulos

00. Comentarios en Python

◆ ¿Qué es un comentario en Python?

Un **comentario** es una línea de texto que **Python no ejecuta**. Su única finalidad es **explicar el código** o **añadir notas** para quien lo lee, incluidos tú mismo en el futuro.

✓ ¿Para qué sirven los comentarios?

- 📖 **Documentar** lo que hace un bloque de código
- □ **Ayudar a comprender** algoritmos complejos
- ✂ **Desactivar código temporalmente** sin borrarlo
- 👤💻 **Colaborar** en proyectos donde otros programadores deben entender tu lógica

◆ Sintaxis básica de los comentarios

▢ *Comentarios de una sola línea*

Se usa el símbolo # al comienzo de la línea:

```
# Este es un comentario  
print("Hola mundo") # Esto imprime un mensaje
```

Todo lo que está a la derecha del # es ignorado por Python.

▢ *Comentarios multilínea (no oficiales)*

Python **no tiene comentarios multilínea "oficiales"** como otros lenguajes, pero hay dos formas comunes de escribirlos:

1. Usar # varias veces:

```
# Este es un comentario  
# que ocupa varias  
# líneas seguidas
```

2. Usar cadenas de texto multilínea con triple comillas ''' o """:

```
'''  
Este también parece un comentario  
pero en realidad es una cadena de texto  
que no se usa, por eso Python lo ignora  
'''
```

⚠ Esta técnica **no es un comentario verdadero**, pero se usa mucho como alternativa rápida.

❏ Características de los comentarios

Característica	Descripción
No ejecutables	Python los ignora completamente
Comienzan con #	Para comentarios de una sola línea
Para humanos	No afectan el funcionamiento del programa
Documentación útil	Facilitan el mantenimiento y colaboración
Alternativas	Se puede usar <code>'''texto'''</code> o <code>"""texto"""</code> como bloque no ejecutado

🔗 Ejemplo práctico de uso de comentarios

```
# Definir el precio base
precio = 100

# Aplicar descuento del 10%
precio = precio * 0.9

# Mostrar el precio final con descuento
print("Precio final:", precio)
```

💡 Buenas prácticas con comentarios

- ☒ Sé breve y claro
- ☒ Comenta el **por qué**, no el **cómo**
- ☒ No repitas lo obvio:

```
x = 5 # Asigna 5 a x ✗ innecesario
```

- ☒ Úsalos para explicar decisiones lógicas o complejas

1. ¿Qué es una variable?

Una **variable** es un nombre que tú le das a un pedazo de información para poder usarlo más adelante en tu código. Piensa en una variable como una **etiqueta** para una caja donde guardas algo.

◆ ¿Cómo se crea una variable en Python?

Simplemente escribes un nombre, un signo igual = y el valor que quieres guardar:

```
mensaje = "Hola mundo"
edad = 30
temperatura = 22.5
```

◆ Tipos de valores que puede tener una variable

Python detecta el tipo automáticamente, según lo que guardes:

Tipo	Ejemplo	Tipo en Python
Texto	"Hola"	str (string)
Entero	25	int (integer)
Decimal	3.14	float
Booleano	True, False	bool
Lista	[1, 2, 3]	list
Diccionario	{"nombre": "Ana"}	dict
Ninguno	None	NoneType

◆ ¿Cómo se usa una variable?

```
nombre = "Carlos"
print("Hola,", nombre) # Usamos la variable en una función
```

◆ Cambiar el valor de una variable

Puedes cambiar el contenido en cualquier momento:

```
edad = 25
edad = edad + 1
print(edad) # Resultado: 26
```

◆ Operaciones con variables

Puedes hacer operaciones dependiendo del tipo de dato:

```
# Números
a = 5
b = 3
resultado = a + b # suma: 8

# Texto
saludo = "Hola"
nombre = "Sofía"
mensaje = saludo + " " + nombre # "Hola Sofía"
```

◆ Nombrar variables: buenas prácticas

- Usa nombres descriptivos: edad, nombre_usuario, total_pedido
- No uses espacios (usa guiones bajos _)
- No empieces con un número
- No uses palabras reservadas como print, if, while, etc.

✓ Ejemplos válidos:

```
numero = 10
nombre_usuario = "Mario"
altura_cm = 170
```

✗ Ejemplos inválidos:

```
2nombre = "error"
mi nombre = "error"
print = "error"
```

◆ Funciones útiles para variables

Aquí algunas funciones y usos prácticos con variables:

type(variable)

Te dice qué tipo de dato contiene la variable:

```
x = 10
print(type(x))  # <class 'int'>
```

Conversión de tipos:

```
# De texto a número
edad = int("25")

# De número a texto
texto = str(100)

# De número decimal a entero
entero = int(4.9)  # resultado: 4
```

◆ Constantes (valores que no cambian)

Python no tiene constantes reales, pero por convención, se escriben en mayúsculas:

```
PI = 3.14159
```

← END Resumen:

- Una **variable** guarda un dato.
- Se crea con `nombre = valor`.
- Python adivina el tipo automáticamente.
- Puedes cambiar su valor.
- Puedes usarlas en operaciones o en funciones como `print()`.
- Existen funciones para revisar o convertir su tipo.

2. ¿Qué es un operador?

Un **operador** es un símbolo que **indica una operación** entre uno o más valores (llamados *operandos*).

✂ Ejemplo:

```
resultado = 5 + 3 # "+" es el operador, 5 y 3 son operandos
```

◆ Tipos de operadores en Python

1. □ Operadores aritméticos

Operador	Descripción	Ejemplo
+	Suma	3 + 2
-	Resta	5 - 1
*	Multiplicación	4 * 2
/	División (decimal)	7 / 2
//	División entera	7 // 2
%	Módulo (residuo)	7 % 2
**	Potencia (exponente)	2 ** 3

2. 🗖 Operadores de comparación

Devuelven `True` o `False`.

Operador	Descripción	Ejemplo
==	Igualdad	5 == 5
!=	Distinto	3 != 2
>	Mayor que	4 > 2
<	Menor que	2 < 5
>=	Mayor o igual que	5 >= 5
<=	Menor o igual que	3 <= 4

3. Operadores lógicos

Usados para combinar condiciones booleanas.

Operador	Descripción	Ejemplo
<code>and</code>	Verdadero si ambas lo son	<code>True and False → False</code>
<code>or</code>	Verdadero si una lo es	<code>True or False → True</code>
<code>not</code>	Invierte el valor lógico	<code>not True → False</code>

```
edad = 20
print(edad > 18 and edad < 30) # True
```

4. Operadores de asignación

Usados para **guardar valores** en variables.

Operador	Descripción	Ejemplo
<code>=</code>	Asignación básica	<code>x = 5</code>
<code>+=</code>	Sumar y asignar	<code>x += 1 → x = x + 1</code>
<code>-=</code>	Restar y asignar	<code>x -= 2</code>
<code>*=</code>	Multiplicar y asignar	<code>x *= 3</code>
<code>/=</code>	Dividir y asignar	<code>x /= 2</code>
<code>//=</code>	División entera y asignar	<code>x //= 2</code>
<code>%=</code>	Módulo y asignar	<code>x %= 2</code>
<code>**=</code>	Potencia y asignar	<code>x **= 2</code>

5. Operadores de pertenencia

Para comprobar si un valor está dentro de una lista, cadena, tupla, etc.

Operador	Descripción	Ejemplo
<code>in</code>	Está presente	<code>"a" in "casa" → True</code>
<code>not in</code>	No está presente	<code>"z" not in "casa" → True</code>

6. Operadores de identidad

Verifican si dos variables **son el mismo objeto** (misma ubicación en memoria).

Operador	Descripción	Ejemplo
<code>is</code>	Son el mismo objeto	<code>a is b</code>
<code>is not</code>	No son el mismo objeto	<code>a is not b</code>
<pre>a = [1, 2] b = a c = [1, 2] print(a is b) # True (apuntan al mismo objeto) print(a is c) # False (aunque tengan los mismos datos)</pre>		

Ejemplo práctico con operadores

```
edad = 25
tiene_licencia = True

if edad >= 18 and tiene_licencia:
    print("Puede conducir")
else:
    print("No puede conducir")
```

Resumen

Tipo	Ejemplos principales
Aritméticos	<code>+, -, *, /, //, %, **</code>
Comparación	<code>==, !=, >, <, >=, <=</code>
Lógicos	<code>and, or, not</code>
Asignación	<code>=, +=, -=, *=, /=, etc.</code>
Pertenencia	<code>in, not in</code>
Identidad	<code>is, is not</code>
Bit a bit	<code>&, ^</code>

3. ¿Qué es un string?

Un **string** es simplemente una **cadena de texto**: letras, palabras, frases o incluso números escritos como texto.

```
mensaje = "Hola mundo"  
nombre = 'Sofía'
```

Puedes usar **comillas dobles** " " o **simples** ' '.

◆ Crear strings

```
texto1 = "Hola"  
texto2 = 'Mundo'  
frase = "Me llamo 'Ana'" # Puedes mezclar comillas
```

◆ Caracteres especiales (escape)

Para usar comillas dentro del texto o saltos de línea, usamos \:

Símbolo Significado

\n	Salto de línea
\t	Tabulación
\\	Barra invertida
\"	Comilla doble
\'	Comilla simple

📌 Ejemplo:

```
print("Hola\nMundo") # Salta de línea
```

◆ Operaciones con strings

1. ◆ Concatenar (unir)

```
nombre = "Carlos"  
saludo = "Hola " + nombre  
print(saludo) # Hola Carlos
```

2. ◆ Repetir

```
print("Hola " * 3) # HolaHolaHola
```

3. ◆ Acceder a un carácter

Los strings son como listas de letras:

```
texto = "Python"  
print(texto[0]) # P (el primero)  
print(texto[-1]) # n (el último)
```

◆ Funciones útiles de strings

✚ `len(texto)`

Devuelve la **cantidad de caracteres**:

```
frase = "Hola"
print(len(frase)) # 4
```

✚ `.lower()` y `.upper()`

Convierte el texto a minúsculas o mayúsculas:

```
nombre = "Ana"
print(nombre.upper()) # ANA
print(nombre.lower()) # ana
```

✚ `.capitalize()` y `.title()`

- `capitalize()` → Pone la **primera letra en mayúscula**
- `title()` → Pone la **primera letra de cada palabra en mayúscula**

```
texto = "hola mundo"
print(texto.capitalize()) # Hola mundo
print(texto.title()) # Hola Mundo
```

✚ `.strip()`

Quita espacios en blanco al principio y al final:

```
entrada = " hola "
print(entrada.strip()) # "hola"
```

✚ `.replace()`

Reemplaza partes del texto:

```
frase = "me gusta el café"
nueva = frase.replace("café", "té")
print(nueva) # me gusta el té
```


✚ `.split()`

Divide el texto en partes (por defecto, en los espacios):

```
frase = "uno dos tres"
palabras = frase.split()
print(palabras) # ['uno', 'dos', 'tres']
```

También puedes dividir por comas u otros caracteres:

```
datos = "nombre,edad,correo"
print(datos.split(",")) # ['nombre', 'edad', 'correo']
```

 **.join()**

Une elementos de una lista en un string:

```
palabras = ['Hola', 'mundo']
print(" ".join(palabras)) # Hola mundo
```

◆ Verificar contenido

Función	Resultado
<code>.startswith("Hola")</code>	¿Empieza con...?
<code>.endswith("mundo")</code>	¿Termina con...?
<code>"a" in texto</code>	¿Contiene...?
<code>.isnumeric()</code>	¿Solo números?
<code>.isalpha()</code>	¿Solo letras?
<code>.isalnum()</code>	¿Letras y/o números?

 **Ejemplo:**

```
codigo = "12345"
print(codigo.isnumeric()) # True
```

◆ Formatear texto (f-strings)

Forma moderna de insertar variables dentro de un texto:

```
nombre = "Ana"
edad = 30
print(f"Me llamo {nombre} y tengo {edad} años")
```

◆ Resumen de operaciones útiles

Función	Ejemplo	Resultado
<code>+</code>	<code>"Hola" + " mundo"</code>	<code>"Hola mundo"</code>
<code>*</code>	<code>"ja" * 3</code>	<code>"jajaja"</code>
<code>texto[i]</code>	<code>"Hola"[1]</code>	<code>'o'</code>
<code>len(texto)</code>	<code>len("Hola")</code>	<code>4</code>
<code>texto.lower()</code>	<code>"HOLA".lower()</code>	<code>"hola"</code>
<code>texto.upper()</code>	<code>"hola".upper()</code>	<code>"HOLA"</code>
<code>texto.strip()</code>	<code>" hola ".strip()</code>	<code>"hola"</code>
<code>texto.split()</code>	<code>"a b c".split()</code>	<code>['a', 'b', 'c']</code>
<code>", ".join(lista)</code>	<code>", ".join(["a", "b"])</code>	<code>"a,b"</code>

4. ¿Qué es una lista?

Una **lista** es una colección **ordenada** y **mutable** (que se puede cambiar), que puede contener **valores de cualquier tipo**, incluso otras listas.

Ejemplo:

```
frutas = ["manzana", "banana", "naranja"]
numeros = [1, 2, 3, 4, 5]
mezcla = [1, "hola", True, 3.5]
```

Características clave

- ✓ **Ordenadas:** mantienen el orden en que agregas los elementos
- ✓ **Indexadas:** puedes acceder a los elementos por su posición
- ✓ **Mutables:** puedes modificarlas después de crearlas
- ✓ **Pueden contener duplicados**
- ✓ **Pueden tener distintos tipos de datos**

Acceso a elementos

```
colores = ["rojo", "verde", "azul"]

print(colores[0])    # rojo
print(colores[-1])   # azul (último)

colores[1] = "amarillo"
print(colores)       # ["rojo", "amarillo", "azul"]
```

Recorrer una lista

```
for fruta in ["manzana", "banana", "uva"]:
    print(fruta)
```

Operaciones con listas

✓ Añadir elementos

Método	Descripción	Ejemplo
<code>append(x)</code>	Añade al final	<code>lista.append("pera")</code>
<code>insert(i, x)</code>	Inserta en posición i	<code>lista.insert(1, "sandía")</code>
<code>extend([...])</code>	Añade otra lista	<code>lista.extend(["melón", "uva"])</code>
<pre>frutas = ["manzana"] frutas.append("pera") # ["manzana", "pera"]</pre>		

```
frutas.insert(1, "uva")    # ["manzana", "uva", "pera"]
```

✓ Eliminar elementos

Método	Descripción	Ejemplo
<code>remove(x)</code>	Elimina la primera coincidencia	<code>lista.remove("uva")</code>
<code>pop([i])</code>	Elimina y devuelve el elemento en índice <code>i</code>	<code>lista.pop(0)</code>
<code>clear()</code>	Vacía toda la lista	<code>lista.clear()</code>

✓ Ordenar y contar

```
numeros = [3, 1, 4, 2]

numeros.sort()      # [1, 2, 3, 4]
numeros.reverse()   # [4, 3, 2, 1]
print(numeros.count(3)) # 1
print(numeros.index(2)) # posición: 2
```

✓ Copiar listas

```
copia = lista.copy()
```

✎ **Ojo:** Si haces `otra = lista`, ambas apuntan a la misma lista.

◆ Slicing (rebanado)

Permite obtener partes de una lista:

```
numeros = [0, 1, 2, 3, 4, 5]

print(numeros[1:4])    # [1, 2, 3]
print(numeros[:3])     # [0, 1, 2]
print(numeros[::2])    # [0, 2, 4] (de 2 en 2)
print(numeros[::-1])  # [5, 4, 3, 2, 1, 0] (al revés)
```

◆ Comprobar si un valor está en la lista

```
frutas = ["manzana", "pera"]

if "manzana" in frutas:
    print("Sí hay manzana")
```

◆ Listas dentro de listas (listas anidadas)

```
matriz = [[1, 2], [3, 4], [5, 6]]
print(matriz[1][0])  # 3
```

◆ Funciones útiles con listas

Función	Descripción
<code>len(lista)</code>	Número de elementos
<code>sum(lista)</code>	Suma de los elementos numéricos
<code>min(lista)</code>	Mínimo valor
<code>max(lista)</code>	Máximo valor
<code>sorted(lista)</code>	Ordena sin modificar la original

□ Ejemplo práctico completo

```
nombres = ["Ana", "Luis", "Marta"]

# Agregar y mostrar
nombres.append("Carlos")
print(nombres)

# Eliminar
nombres.remove("Luis")

# Ordenar
nombres.sort()

# Ver cada uno
for nombre in nombres:
    print(f"Hola, {nombre}")
```

✂ Resumen rápido

- Las **listas** guardan múltiples valores.
- Se usan `[]` para crearlas.
- Son **mutables**: puedes cambiar su contenido.
- Puedes recorrerlas con `for`, modificarlas, ordenarlas y mucho más.

5. ¿Qué es una tupla?

Una **tupla** es una colección **ordenada** e **inmutable** de elementos. Se parece mucho a una lista, pero **no se puede modificar** una vez creada.

✂ Se escriben con paréntesis `()`:

```
coordenadas = (10, 20)
colores = ("rojo", "verde", "azul")
```

□ Diferencias entre lista y tupla

Característica	Lista ([])	Tupla (())
Mutable	✓ Sí	✗ No
Sintaxis	[]	()
Velocidad	Más lenta	Más rápida
Uso	Datos que cambian	Datos fijos

◆ Crear una tupla

```
tupla = (1, 2, 3)
tupla2 = ("Python", 3.9, True)

# También se puede crear sin paréntesis (no recomendable):
otra = 1, 2, 3

# Tupla de un solo elemento (¡ojo con la coma!)
tupla_unica = ("hola",)
```

◆ Acceder a elementos

Como en las listas:

```
colores = ("rojo", "verde", "azul")

print(colores[0])    # rojo
print(colores[-1])   # azul
```

◆ Recorrer una tupla

```
for color in colores:
    print(color)
```

◆ Métodos disponibles

Las tuplas tienen muy pocos métodos, ya que **no se pueden modificar**:

Método	Descripción
<code>.count(x)</code>	Cuenta cuántas veces aparece x
<code>.index(x)</code>	Devuelve el índice de la primera ocurrencia

Ejemplo:

```
tupla = (1, 2, 2, 3, 4)

print(tupla.count(2))    # 2
print(tupla.index(3))    # 3 (posición del número 3)
```

◆ Operaciones con tuplas

◆ Concatenar

```
a = (1, 2)
b = (3, 4)
c = a + b
print(c)    # (1, 2, 3, 4)
```

◆ Repetir

```
t = ("Hola",) * 3
print(t)    # ('Hola', 'Hola', 'Hola')
```

◆ Slicing (rebanado)

```
t = (10, 20, 30, 40, 50)
print(t[1:4])    # (20, 30, 40)
```

◆ ¿Para qué usar tuplas?

- ✓ Cuando necesitas **proteger datos** (que no se modifiquen)
- ✓ Para mejorar el **rendimiento** (son más rápidas que las listas)
- ✓ Como **claves de diccionarios**
- ✓ Para devolver **múltiples valores** de una función

◆ Desempaquetar tuplas (unpacking)

Puedes asignar los valores de una tupla a variables:

```
persona = ("Ana", 30, "Perú")

nombre, edad, pais = persona

print(nombre)    # Ana
print(edad)      # 30
print(pais)      # Perú
```

◆ Convertir entre listas y tuplas

```
# Lista a tupla
l = [1, 2, 3]
```

```
t = tuple(l)

# Tupla a lista
t = (4, 5, 6)
l = list(t)
```

□ Ejemplo práctico

```
dias = ("lunes", "martes", "miércoles", "jueves", "viernes")

# Imprimir solo días hábiles
for dia in dias:
    print(dia)

# Verificar si un día existe
if "domingo" in dias:
    print("Es un día válido")
else:
    print("Domingo no está en la tupla")
```

✓ Resumen final

Propiedad	Tupla
Ordenada	✓ Sí
Indexada	✓ Sí
Mutable	✗ No
Elementos	Cualquier tipo
Métodos útiles	<code>count()</code> , <code>index()</code>
Sintaxis	<code>tupla = (1, 2, 3)</code>

6. ¿Qué es un *set*?

Un **set** es una colección **no ordenada**, **sin elementos duplicados** y **mutable** (puedes cambiarlo).

✂ Se crea con llaves `{}` o con la función `set()`.

```
numeros = {1, 2, 3}
colores = set(["rojo", "verde", "azul"])
```

❑ Características de los sets

Propiedad	Valor
✓ No tienen duplicados	Elimina repetidos
✓ No están ordenados	No tiene índices
✓ Son mutables	Se pueden modificar
✗ No se puede acceder por posición	No hay índices

◆ Crear un set

```
frutas = {"manzana", "pera", "uva"}
numeros = set([1, 2, 2, 3, 4])
print(numeros)  # {1, 2, 3, 4} - elimina duplicados
```

⚠ Un set vacío se crea con `set()`, no con `{}`, porque `{}` crea un diccionario.

◆ Agregar elementos

```
colores = {"rojo", "verde"}
colores.add("azul")
print(colores)
```

◆ Eliminar elementos

Método	Descripción
<code>.remove(x)</code>	Elimina <code>x</code> (error si no existe)
<code>.discard(x)</code>	Elimina <code>x</code> (NO da error si no existe)
<code>.pop()</code>	Elimina un elemento aleatorio
<code>.clear()</code>	Vacía el set

```
colores.remove("verde")
colores.discard("amarillo")  # No da error
```

◆ Recorrer un set

```
animales = {"perro", "gato", "pez"}

for animal in animales:
    print(animal)
```

⚠ No garantiza el orden de los elementos.

◆ Comprobar si un valor está en el set

```
if "manzana" in frutas:  
    print("Sí hay manzana")
```

◆ Operaciones entre sets

Los sets permiten hacer **operaciones matemáticas de conjuntos**:

◆ Unión → `set1 | set2` o `.union()`

```
a = {1, 2, 3}  
b = {3, 4, 5}  
print(a | b)           # {1, 2, 3, 4, 5}  
print(a.union(b))
```

◆ Intersección → `set1 & set2` o `.intersection()`

```
print(a & b)           # {3}  
print(a.intersection(b))
```

◆ Diferencia → `set1 - set2` o `.difference()`

```
print(a - b)           # {1, 2}
```

◆ Diferencia simétrica → `set1 ^ set2` o `.symmetric_difference()`

```
print(a ^ b)           # {1, 2, 4, 5}
```

◆ Métodos útiles

Método	Descripción
<code>.add(elem)</code>	Agrega un elemento
<code>.remove(elem)</code> / <code>.discard()</code>	Elimina un elemento
<code>.pop()</code>	Elimina un elemento aleatorio
<code>.clear()</code>	Vacía el set
<code>.union(set2)</code>	Unión con otro set
<code>.intersection(set2)</code>	Elementos comunes
<code>.difference(set2)</code>	Elementos solo en el primero

Método	Descripción
<code>.symmetric_difference(set2)</code>	Elementos únicos en ambos
<code>.issubset(set2)</code>	¿Es subconjunto?
<code>.issuperset(set2)</code>	¿Es superconjunto?
<code>.copy()</code>	Copia del set

▣ Ejemplo práctico

```
lenguajes_backend = {"Python", "Java", "PHP"}
lenguajes_frontend = {"JavaScript", "HTML", "CSS", "Python"}

# Lenguajes usados en ambos
comunes = lenguajes_backend & lenguajes_frontend
print("Comunes:", comunes)

# Todos los lenguajes
todos = lenguajes_backend | lenguajes_frontend
print("Todos:", todos)

# Solo backend (que no estén en frontend)
solo_backend = lenguajes_backend - lenguajes_frontend
print("Solo backend:", solo_backend)
```

✓ Resumen rápido

Concepto	Explicación
Sintaxis	<code>set()</code> o <code>{1, 2, 3}</code>
Orden	✗ No ordenados
Duplicados	✗ No se permiten
Acceso directo	✗ No hay índices
Mutables	✓ Sí
Útiles para	Eliminar duplicados, operaciones de conjuntos

7. ¿Qué es un diccionario?

Un **diccionario** (`dict`) es una **colección desordenada, mutable y sin claves duplicadas** que almacena datos en forma de **clave: valor**.

Ejemplo:

```
persona = {  
    "nombre": "Ana",  
    "edad": 30,  
    "ciudad": "Madrid"  
}
```

◆ Características de los diccionarios

Propiedad	Valor
Ordenados (desde Python 3.7)	✓ Conservan el orden de inserción
Mutables	✓ Se pueden modificar
Indexación	Por clave, no por posición
Duplicados	✗ No se permiten claves duplicadas

◆ Sintaxis

```
diccionario = {  
    "clave1": valor1,  
    "clave2": valor2  
}
```

O usando `dict()`:

```
datos = dict(nombre="Pedro", edad=25)
```

◆ Acceder a valores

```
print(persona["nombre"]) # Ana
```

⚠ Si accedes a una clave que no existe, da **error**. Usa `.get()` si quieres evitarlo:

```
print(persona.get("profesion", "No definida"))
```

◆ Modificar valores

```
persona["edad"] = 31
```

◆ Agregar nuevas claves

```
persona["profesion"] = "Ingeniera"
```

◆ Eliminar claves

Método	Descripción
<code>del dic[key]</code>	Elimina una clave (da error si no existe)
<code>.pop(key)</code>	Elimina clave y devuelve su valor
<code>.popitem()</code>	Elimina y devuelve el último par
<code>.clear()</code>	Vacía todo el diccionario

```
del persona["ciudad"]
```

◆ Recorrer un diccionario

```
for clave in persona:  
    print(clave, "→", persona[clave])
```

También puedes usar:

```
# Solo claves  
for k in persona.keys():  
    print(k)  
  
# Solo valores  
for v in persona.values():  
    print(v)  
  
# Claves y valores  
for k, v in persona.items():  
    print(k, ":", v)
```

◆ Comprobar si una clave existe

```
if "edad" in persona:  
    print("La edad está registrada")
```

◆ Funciones y métodos útiles

Método	Descripción
<code>.get(clave, defecto)</code>	Devuelve el valor o un valor por defecto
<code>.keys()</code>	Devuelve todas las claves
<code>.values()</code>	Devuelve todos los valores

Método	Descripción
<code>.items()</code>	Devuelve pares (clave, valor)
<code>.update(dict2)</code>	Actualiza con otro diccionario
<code>.pop(clave)</code>	Elimina una clave y devuelve su valor
<code>.popitem()</code>	Elimina el último par añadido
<code>.clear()</code>	Elimina todo el contenido
<code>len(dic)</code>	Número de elementos

▣ Ejemplo práctico

```
producto = {
    "nombre": "Teclado",
    "precio": 20,
    "stock": 15
}

# Aumentar stock
producto["stock"] += 5

# Agregar descripción
producto["descripcion"] = "Teclado mecánico"

# Mostrar todo
for clave, valor in producto.items():
    print(clave, ":", valor)
```

✓ Resumen general

Concepto	Valor
Tipo	<code>dict</code>
Sintaxis	<code>{"clave": valor}</code>
Indexación	Por clave, no por índice numérico
Duplicados	✗ No se permiten claves repetidas
Orden	✓ Mantienen orden desde Python 3.7
Mutabilidad	✓ Puedes agregar, cambiar o borrar valores

8. ¿Qué es un condicional?

Un **condicional** permite que el programa **ejecute una parte del código u otra dependiendo de si una condición es verdadera (True) o falsa (False)**.

Por ejemplo:

```
if edad >= 18:
    print("Eres mayor de edad")
```

◆ Tipos de condicionales en Python

Palabra clave	Descripción
if	Si se cumple una condición
elif	Sino si (otra condición)
else	Sino (ninguna de las anteriores)

◆ Sintaxis básica

```
if condición:
    # Código si se cumple
elif otra_condición:
    # Código si se cumple esta otra
else:
    # Código si ninguna se cumplió
```

🔍 Ejemplo:

```
edad = 20

if edad >= 18:
    print("Puedes votar")
elif edad >= 16:
    print("Puedes votar con autorización")
else:
    print("No puedes votar")
```

◆ Características de los condicionales

Característica	Valor
Evaluación booleana	Solo se ejecuta si la condición es True

Característica	Valor
Soporte de múltiples ramas	Se puede usar <code>elif</code> y <code>else</code>
Se puede anidar	Puedes poner un <code>if</code> dentro de otro
Se combina con operadores	Aritméticos, lógicos, comparación, etc.

◆ Tipos de condiciones

Las condiciones se evalúan como **booleanas** (verdaderas o falsas). Puedes usar:

1. Comparaciones

```
x == 5      # igual
x != 3      # distinto
x > 2       # mayor
x <= 8      # menor o igual
```

2. Operadores lógicos

```
if edad > 18 and tiene_licencia:
    print("Puedes conducir")
```

Operador Significado

`and` `y`

`or` `o`

`not` `no`

◆ Anidamiento de condicionales (if dentro de otro if)

```
if edad > 18:
    if tiene_licencia:
        print("Puedes conducir")
    else:
        print("Necesitas una licencia")
else:
    print("Eres menor de edad")
```

◆ Condicional en una sola línea (condicional ternario)

```
mensaje = "Mayor" if edad >= 18 else "Menor"
print(mensaje)
```

◆ Ejemplo completo

```
usuario = input("Introduce tu nombre: ")
```

```
edad = int(input("Introduce tu edad: "))

if edad < 12:
    print(f"{usuario}, eres un niño")
elif edad < 18:
    print(f"{usuario}, eres un adolescente")
elif edad < 65:
    print(f"{usuario}, eres un adulto")
else:
    print(f"{usuario}, eres un adulto mayor")
```

✓ Resumen general

Elemento	Ejemplo	Significado
if	if x > 0:	Ejecuta si la condición es True
elif	elif x == 0:	Otra condición si la anterior falla
else	else:	Si ninguna condición fue verdadera
Lógicos	and, or, not	Combinar condiciones
Anidado	if dentro de otro if	Estructura más compleja
Ternario	"Mayor" if edad >= 18 else "Menor"	Condición rápida en una línea

9. ¿Qué es un bucle?

Un **bucle** (o ciclo) es una estructura que ejecuta una misma sección de código **repetidamente** mientras se cumpla una condición o se itere sobre una colección de elementos.

◆ Tipos de bucles en Python

Python tiene principalmente dos tipos de bucles:

Tipo	Descripción
for	Repite un bloque de código un número determinado de veces o sobre una colección
while	Repite un bloque mientras una condición sea verdadera

1. Bucle for

◆ ¿Para qué sirve?

Se usa para **recorrer una secuencia** (lista, tupla, cadena, rango, etc.) o repetir algo un número fijo de veces.

◆ Sintaxis

```
for variable in secuencia:  
    # código a repetir
```

◆ Ejemplo simple con lista

```
frutas = ["manzana", "banana", "cereza"]  
  
for fruta in frutas:  
    print("Me gusta la", fruta)
```

Salida:

```
Me gusta la manzana  
Me gusta la banana  
Me gusta la cereza
```

◆ Ejemplo con rango de números

```
for i in range(5): # i va de 0 a 4  
    print(i)
```

Salida:

```
CopiarEditar  
0  
1  
2  
3  
4
```

◆ Funciones útiles con `for`

- `range(start, stop, step)`: Genera una secuencia numérica (no incluye stop).

```
for i in range(2, 10, 2): # Números pares de 2 a 8  
    print(i)
```

Salida:

```
CopiarEditar  
2  
4  
6  
8
```

- Recorrer diccionarios:

```
persona = {"nombre": "Ana", "edad": 30}

for clave, valor in persona.items():
    print(clave, ":", valor)
```

2. Bucle while

◆ ¿Para qué sirve?

Repite el código **mientras una condición sea verdadera**.

◆ Sintaxis

```
while condición:
    # código a repetir
```

◆ Ejemplo simple

```
contador = 0

while contador < 5:
    print("Contador =", contador)
    contador += 1 # Es importante modificar la condición para evitar bucle infinito
```

Salida:

```
Contador = 0
Contador = 1
Contador = 2
Contador = 3
Contador = 4
```

◆ Evitar bucles infinitos

Si la condición nunca se vuelve `False`, el programa se quedará repitiendo el bloque para siempre. Siempre asegúrate de modificar variables dentro del bucle que afecten la condición.

🔗 Control del flujo dentro de bucles

1. `break`

Termina el bucle inmediatamente.

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

Salida:

```
0  
1  
2  
3  
4
```

2. continue

Salta la iteración actual y sigue con la siguiente.

```
for i in range(5):  
    if i == 2:  
        continue  
    print(i)
```

Salida:

```
0  
1  
3  
4
```

3. else en bucles

Puedes usar un `else` que se ejecuta **si el bucle termina normalmente**, es decir, sin encontrar un `break`.

```
for i in range(3):  
    print(i)  
else:  
    print("Bucle terminado sin interrupción")
```

□ Resumen

Bucle	Uso principal	Sintaxis básica
<code>for</code>	Iterar sobre secuencias o rangos	<code>for var in secuencia:</code>
<code>while</code>	Repetir mientras condición sea verdadera	<code>while condición:</code>
<code>break</code>	Terminar el bucle	Dentro del bucle
<code>continue</code>	Saltar a la siguiente iteración	Dentro del bucle
<code>else</code>	Código al terminar bucle sin interrupción	Después del bucle

Bucles y condicionales combinados en Python

1. ¿Por qué combinarlos?

Los bucles repiten código, y los condicionales deciden qué código ejecutar según ciertas condiciones. Combinados, permiten:

- Ejecutar código repetido que cambia según diferentes situaciones.
- Filtrar, modificar o actuar solo sobre ciertos elementos.
- Crear programas con lógica compleja y adaptativa.

2. Ejemplo básico: imprimir solo números pares en un rango

```
for i in range(10):
    if i % 2 == 0:      # Condición: si i es par
        print(i, "es par")
```

Qué hace:

- El bucle recorre los números del 0 al 9.
- El condicional dentro filtra solo los pares y los imprime.

3. Ejemplo con `while` y condicionales: pedir números positivos

```
while True:
    num = int(input("Introduce un número positivo (0 para salir): "))

    if num == 0:
        print("Saliendo del programa.")
        break

    if num < 0:
        print("Número inválido, intenta de nuevo.")
        continue

    print(f"Número válido: {num}")
```

Qué pasa aquí:

- El bucle `while True` crea un ciclo infinito.
- Si el usuario introduce 0, se rompe el ciclo con `break`.
- Si el número es negativo, muestra mensaje y usa `continue` para saltar al siguiente ciclo sin imprimir nada más.
- Solo si el número es positivo se muestra "Número válido".

4. Ejemplo práctico: contar cuántos números positivos y negativos ingresó el usuario

```
positivos = 0
negativos = 0

for _ in range(5):
    n = int(input("Introduce un número: "))

    if n > 0:
        positivos += 1
    elif n < 0:
        negativos += 1
    else:
        print("Has introducido cero, no se cuenta.")

print(f"Has introducido {positivos} números positivos y {negativos} negativos.")
```

5. Anidando condicionales dentro de bucles

```
for i in range(1, 6):
    if i % 2 == 0:
        if i > 3:
            print(f"{i} es par y mayor que 3")
        else:
            print(f"{i} es par y menor o igual a 3")
    else:
        print(f"{i} es impar")
```

6. Resumen rápido

Acción	Ejemplo
Bucle con condicional	Repetir y decidir dentro del bucle
Usar <code>break</code>	Salir del bucle si se cumple una condición
Usar <code>continue</code>	Saltar una iteración según condición
Anidar <code>if</code> dentro de <code>for</code> o <code>while</code>	Lógica más específica en cada vuelta

10. ¿Qué es una función?

Una **función** es un **bloque de código reutilizable** que realiza una tarea específica. Puedes **llamar** (usar) esa función tantas veces como quieras sin repetir el código.

◆ ¿Por qué usar funciones?

- **Organizar el código:** Hace que el programa sea más claro y modular.
- **Reutilización:** Evita repetir el mismo código varias veces.
- **Mantenimiento:** Facilita cambiar una sola vez la lógica y que afecte en todas partes.
- **Separación de responsabilidades:** Cada función tiene una tarea clara.

◆ Sintaxis básica para definir una función

```
def nombre_funcion(parámetros):
    # Bloque de código
    instrucciones
    return valor # opcional
```

- `def`: palabra clave para definir funciones.
- `nombre_funcion`: nombre que le das a la función (reglas de identificadores).
- `parámetros`: datos que la función recibe para trabajar (opcionales).
- `return`: devuelve un resultado (opcional).

◆ Ejemplo simple: función sin parámetros ni retorno

```
def saludar():  
    print("¡Hola, mundo!")
```

Llamar a la función:

```
saludar()
```

Salida:

```
¡Hola, mundo!
```

◆ Función con parámetros

```
def saludar(nombre):  
    print(f"Hola, {nombre}!")
```

Llamada:

```
saludar("Ana")
```

Salida:

```
Hola, Ana!
```

◆ Función con retorno

```
def sumar(a, b):  
    resultado = a + b  
    return resultado
```

Uso:

```
suma = sumar(3, 5)  
print(suma)  # 8
```

◆ Parámetros opcionales y valores por defecto

Puedes asignar valores por defecto a parámetros para que sean opcionales:

```
def saludar(nombre="amigo"):  
    print(f"Hola, {nombre}!")
```

Llamadas válidas:

```
saludar("Luis")  # Hola, Luis!  
saludar()        # Hola, amigo!
```

◆ Tipos de parámetros

- **Posicionales:** Se pasan en orden y se asignan según posición.
- **Nombrados (keywords):** Se especifican con nombre.
- **Argumentos variables:** Reciben cualquier número de parámetros.

Ejemplo con argumentos variables:

```
def suma_todos(*numeros):  
    total = 0  
    for n in numeros:  
        total += n  
    return total  
  
print(suma_todos(1, 2, 3, 4)) # 10
```

◆ Ámbito de variables (scope)

- Variables definidas dentro de la función son **locales** (solo existen allí).
- Variables definidas fuera son **globales**.
- Para modificar una variable global dentro de una función, usa `global`.

◆ Documentación (docstring)

Es buena práctica documentar funciones con triple comillas:

```
def sumar(a, b):  
    """Suma dos números y devuelve el resultado."""  
    return a + b
```

Puedes verlo con:

```
print(sumar.__doc__)
```

◆ Funciones anónimas (lambda)

Funciones pequeñas y sin nombre:

```
doblar = lambda x: x * 2  
print(doblar(5)) # 10
```

◆ Ejemplo completo

```
def calcular_area_rectangulo(base, altura):  
    """Calcula el área de un rectángulo."""  
    return base * altura  
  
b = float(input("Base: "))  
h = float(input("Altura: "))
```

```
area = calcular_area_rectangulo(b, h)
print("El área es:", area)
```

□ Resumen de funciones

Concepto	Descripción
Definición	<code>def nombre(parámetros):</code>
Parámetros	Datos de entrada
Retorno (<code>return</code>)	Devuelve resultado (opcional)
Parámetros por defecto	Permite llamar sin pasar todos los parámetros
Argumentos variables	<code>*args</code> para cantidad variable de parámetros
Documentación (docstring)	Explica qué hace la función
Variables locales	Solo visibles dentro de la función
Funciones lambda	Funciones anónimas para tareas simples

11. ¿Qué es una clase?

Una **clase** es una plantilla o molde para crear **objetos** (instancias). Define las **propiedades** (atributos) y **comportamientos** (métodos) que tendrán esos objetos.

- Piensa en la clase como el plano de una casa.
- El objeto es una casa construida según ese plano.

1. ¿Por qué usar clases?

- Modelar objetos del mundo real en el código.
- Organizar el programa de forma clara y modular.
- Reutilizar código creando múltiples objetos de una misma clase.
- Facilitar el mantenimiento y la extensión del código.
- Permitir la encapsulación (ocultar detalles internos).
- Permitir la herencia (crear nuevas clases basadas en otras).
- Facilitar el polimorfismo (métodos con el mismo nombre en distintas clases).

2. Sintaxis básica para definir una clase

```
class NombreClase:
    def __init__(self, parámetros):
        # Constructor: inicializa atributos del objeto
        self.atributo1 = valor1
        self.atributo2 = valor2

    def metodo(self):
        # Método: función dentro de la clase que realiza una acción
        pass
```

- `class`: palabra reservada para definir clases.
- `NombreClase`: nombre de la clase (por convención, empieza con mayúscula).
- `__init__`: método especial llamado **constructor**, se ejecuta al crear el objeto.

- `self`: referencia al propio objeto (como `this` en otros lenguajes).
- Los métodos siempre reciben `self` como primer parámetro.

3. Crear un objeto (instancia) de la clase

```
mi_objeto = NombreClase(parámetros)
```

Esto crea un objeto con sus propios atributos y métodos.

4. Ejemplo básico: clase **Persona**

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre # atributo nombre
        self.edad = edad     # atributo edad

    def saludar(self):
        print(f"Hola, soy {self.nombre} y tengo {self.edad} años.")
```

Crear objetos e invocar métodos:

```
personal = Persona("Ana", 30)
persona2 = Persona("Luis", 25)

personal.saludar() # Hola, soy Ana y tengo 30 años.
persona2.saludar() # Hola, soy Luis y tengo 25 años.
```

5. Métodos especiales

- `__init__(self, ...)`: constructor, se ejecuta al crear el objeto.
- `__str__(self)`: define lo que muestra `print(objeto)`.

Ejemplo con `__str__`:

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def __str__(self):
        return f"{self.nombre}, {self.edad} años"

p = Persona("Ana", 30)
print(p) # Ana, 30 años
```

6. Herencia (clases hijas)

Permite crear una clase que hereda atributos y métodos de otra clase (clase padre).

```
class Estudiante(Persona):
    def __init__(self, nombre, edad, carrera):
        super().__init__(nombre, edad) # llama al constructor de Persona
        self.carrera = carrera

    def estudiar(self):
```

```
print(f"{self.nombre} está estudiando {self.carrera}.")
```

Uso:

```
e = Estudiante("Luis", 22, "Matemáticas")
e.saludar()    # Método heredado de Persona
e.estudiar()   # Método propio de Estudiante
```

7. Encapsulación y propiedades

- Por convención, los atributos “privados” se nombran con un guion bajo: `_atributo`.
- Para controlar el acceso se usan **propiedades** con `@property` y `setters`.

Ejemplo:

```
class Cuenta:
    def __init__(self, saldo):
        self._saldo = saldo # atributo "privado"

    @property
    def saldo(self):
        return self._saldo

    @saldo.setter
    def saldo(self, valor):
        if valor >= 0:
            self._saldo = valor
        else:
            print("Saldo no puede ser negativo")
```

Uso:

```
c = Cuenta(100)
print(c.saldo)    # 100
c.saldo = -50     # Saldo no puede ser negativo
```

8. Ámbito de variables en clases

- Los atributos definidos con `self.` son **locales al objeto**.
- Variables definidas fuera son globales o de clase (con otras sintaxis).

9. Métodos comunes en clases

- Métodos que modifican atributos.
- Métodos que muestran información.
- Métodos que realizan cálculos o acciones.

10. Funciones lambda en clases (menos común pero posible)

```
class Calculadora:
    doble = lambda self, x: x * 2
```

11. Ejemplo completo con varias características

```
class Coche:
    def __init__(self, marca, modelo, año):
        self.marca = marca
        self.modelo = modelo
        self.año = año

    def mostrar_info(self):
        print(f"{self.marca} {self.modelo} - Año {self.año}")

    def actualizar_año(self, nuevo_año):
        self.año = nuevo_año

    def __str__(self):
        return f"{self.marca} {self.modelo}, año {self.año}"

# Crear objeto
mi_coche = Coche("Toyota", "Corolla", 2010)
print(mi_coche) # Toyota Corolla, año 2010
mi_coche.mostrar_info()
mi_coche.actualizar_año(2022)
print(mi_coche) # Toyota Corolla, año 2022
```

12. Resumen de conceptos

Concepto	Descripción
Clase	Molde para crear objetos
Objeto	Instancia concreta de una clase
Atributo	Propiedad o estado del objeto
Método	Función que realiza una acción del objeto
<code>__init__</code>	Constructor que inicializa atributos
<code>self</code>	Referencia al objeto actual
Herencia	Clases derivadas que reutilizan código
Encapsulación	Controlar acceso a datos con propiedades
Métodos especiales	Funciones con nombres especiales (<code>__str__</code> , etc.)
Propiedades	Getters y setters para controlar acceso

12. ¿Qué son las excepciones?

Las **excepciones** son errores que ocurren durante la ejecución de un programa y que interrumpen su flujo normal.

- Por ejemplo, dividir entre cero, acceder a un índice inexistente, abrir un archivo que no existe, etc.
- Python genera automáticamente estos errores como **excepciones**.
- Si no se gestionan, el programa termina con un error (traceback).

1. ¿Para qué sirven las excepciones?

- Detectar y manejar errores de forma controlada.
- Evitar que el programa se cierre abruptamente.
- Permitir que el programa tome acciones alternativas o muestre mensajes amigables.
- Facilitar la depuración y robustez del código.

2. Sintaxis básica para manejar excepciones: `try-except`

```
try:
    # Código que puede generar error
    instrucción(es)
except TipoDeError:
    # Código para manejar el error específico
    instrucción(es)
```

- El bloque `try` contiene código que se ejecuta normalmente.
- Si ocurre un error dentro del `try` y coincide con el `except`, se ejecuta el código del `except`.
- Luego el programa continúa normalmente.

3. Ejemplo básico

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("No se puede dividir entre cero.")
```

Salida:

No se puede dividir entre cero.

4. Capturar cualquier excepción

Puedes capturar cualquier error con `except Exception` o simplemente `except` (pero es mejor especificar el tipo):

```
try:
    x = int("hola")
except Exception as e:
    print("Error:", e)
```

5. Manejar múltiples excepciones

```
try:
    x = int(input("Número: "))
    y = 10 / x
except ValueError:
    print("Debes ingresar un número válido.")
except ZeroDivisionError:
    print("No puedes dividir entre cero.")
```

6. Bloque **else**

Se ejecuta si **no hay excepción** en el `try`:

```
try:
    x = int(input("Número: "))
except ValueError:
    print("Error: no es un número válido.")
else:
    print(f"El número ingresado es {x}")
```

7. Bloque **finally**

Se ejecuta **siempre**, ocurra o no excepción. Se usa para liberar recursos (archivos, conexiones, etc.):

```
try:
    archivo = open("datos.txt")
    datos = archivo.read()
except FileNotFoundError:
    print("El archivo no existe.")
finally:
    archivo.close()
    print("Archivo cerrado.")
```

8. Lanzar (generar) excepciones: **raise**

Puedes generar una excepción tú mismo para indicar un error:

```
def dividir(a, b):
    if b == 0:
        raise ValueError("No se puede dividir entre cero")
    return a / b

try:
    resultado = dividir(5, 0)
except ValueError as e:
    print(e)
```

9. Características importantes

Característica	Descripción
Excepción	Error detectado durante ejecución
<code>try</code>	Bloque donde puede ocurrir un error
<code>except</code>	Bloque que maneja el error
<code>else</code>	Bloque que se ejecuta si no hay error
<code>finally</code>	Bloque que se ejecuta siempre (liberar recursos)
<code>raise</code>	Generar una excepción explícitamente

Característica	Descripción
Captura de error	Guardar info del error con <code>except Exception as e</code>
Jerarquía	Excepciones tienen herencia y pueden ser específicas

10. Jerarquía y tipos comunes de excepciones

Algunos tipos comunes que puedes usar en `except`:

- `Exception` — clase base para todas las excepciones.
- `ValueError` — error al pasar un valor inválido.
- `ZeroDivisionError` — división por cero.
- `IndexError` — índice fuera de rango.
- `KeyError` — clave no encontrada en diccionario.
- `FileNotFoundError` — archivo no encontrado.
- `TypeError` — operación con tipo incorrecto.
- `ImportError` — fallo al importar un módulo.

11. Ejemplo combinando varios bloques

```
try:
    x = int(input("Ingresa un número: "))
    resultado = 10 / x
except ValueError:
    print("Debes ingresar un número válido.")
except ZeroDivisionError:
    print("No puedes dividir entre cero.")
else:
    print(f"El resultado es {resultado}")
finally:
    print("Operación finalizada.")
```

12. Buenas prácticas al usar excepciones

- Captura solo excepciones específicas que esperas.
- No uses excepciones para controlar la lógica normal.
- Usa mensajes claros al manejar errores.
- Usa `finally` para liberar recursos siempre.
- Puedes crear tus propias excepciones personalizadas heredando de `Exception`.

13. Crear excepciones personalizadas

```
class MiError(Exception):
    pass

def hacer_algo(valor):
    if valor < 0:
        raise MiError("Valor no puede ser negativo")

try:
    hacer_algo(-1)
except MiError as e:
```

```
print("Error personalizado:", e)
```

Resumen rápido

Bloque	¿Cuándo se ejecuta?
--------	---------------------

<code>try</code>	Siempre primero, aquí va código riesgoso
<code>except</code>	Solo si hay error que coincida con excepción
<code>else</code>	Solo si NO hay error
<code>finally</code>	Siempre, pase lo que pase

13. ¿Qué es un módulo?

Un **módulo** es un archivo que contiene definiciones y declaraciones de Python (funciones, clases, variables, código ejecutable) y que se puede reutilizar en otros programas.

- En esencia, un módulo es un archivo `.py`.
- Permite organizar el código en partes más pequeñas, legibles y reutilizables.
- Facilita la modularidad y el mantenimiento del código.

1. ¿Por qué usar módulos?

- Reutilizar funciones, clases y variables en varios programas.
- Evitar repetir código.
- Organizar proyectos grandes en archivos separados.
- Acceder a funcionalidades ya implementadas (bibliotecas estándar y externas).
- Facilitar la colaboración en equipo.

2. Sintaxis básica para usar módulos: `import`

Para usar un módulo en tu código, lo importas con `import`:

```
import nombre_modulo
```

Ejemplo:

```
import math  
  
print(math.sqrt(16)) # 4.0
```

Aquí, `math` es un módulo estándar que contiene funciones matemáticas.

3. Acceder a funciones, variables o clases del módulo

Usas la sintaxis:

```
nombre_modulo.funcion()  
nombre_modulo.variable
```

Por ejemplo:

```
import random

num = random.randint(1, 10)
print(num)
```

4. Importar funciones/clases específicas con `from ... import`

Si quieres importar solo partes concretas de un módulo, usas:

```
from nombre_modulo import funcion1, funcion2
```

Ejemplo:

```
from math import pi, sqrt

print(pi)           # 3.141592653589793
print(sqrt(25))     # 5.0
```

5. Importar todo el contenido con `from ... import *`

```
from math import *
```

Nota: No es recomendable en proyectos grandes, porque puede causar confusión por nombres repetidos.

6. Renombrar módulos o funciones con `as`

Puedes darle un alias para usar un nombre corto:

```
import numpy as np
print(np.array([1, 2, 3]))
```

O funciones:

```
from math import sqrt as raiz_cuadrada
print(raiz_cuadrada(16))
```

7. Crear tus propios módulos

Solo crea un archivo `.py` con funciones, clases, variables, por ejemplo:

```
mimodulo.py:

def saludar(nombre):
    print(f"Hola, {nombre}!")

PI = 3.14159
```

Luego úsalo en otro archivo:

```
import mimodulo
```

```
mimodulo.saludar("Ana")
print(mimodulo.PI)
```

8. El archivo `__init__.py`

- En carpetas, el archivo `__init__.py` indica que la carpeta es un paquete (colección de módulos).
- Puede estar vacío o con código para inicializar el paquete.

9. Algunas funciones útiles relacionadas con módulos

- `dir(modulo)`: muestra las funciones, variables y clases que contiene un módulo.
- `help(modulo)`: muestra documentación del módulo.
- `type(modulo)`: indica que es un módulo.
- `module.__name__`: nombre del módulo.

Ejemplo:

```
import math

print(dir(math))
print(help(math.sqrt))
```

10. Paquetes: módulos en carpetas

- Un paquete es una carpeta con varios módulos y un archivo `__init__.py`.
- Permite organizar módulos en jerarquías.

Estructura:

```
mi_paquete/
  __init__.py
  modulo1.py
  modulo2.py
```

Importar:

```
from mi_paquete import modulo1
modulo1.funcion()
```

11. Ejemplo completo: creación y uso de un módulo

Archivo: `operaciones.py`

```
def sumar(a, b):
    return a + b

def restar(a, b):
    return a - b

PI = 3.1416
```

Archivo: programa.py

```
import operaciones

print(operaciones.sumar(10, 5)) # 15
print(operaciones.restar(10, 5)) # 5
print(operaciones.PI)           # 3.1416
```

12. Características principales de los módulos

Característica	Descripción
Archivo .py	Cada módulo es un archivo Python
Reutilización	Permite reutilizar código entre programas
Encapsulamiento	Oculta detalles internos y expone funciones/clases
Alias (as)	Permite importar con otro nombre para evitar conflictos
Modularidad	Facilita organización y mantenimiento de proyectos
Biblioteca estándar	Python trae muchos módulos útiles integrados
Paquetes	Carpetas con varios módulos y <code>__init__.py</code>

13. Importancia del `if __name__ == "__main__":`

Dentro de un módulo, puedes poner código que solo se ejecute si se ejecuta el archivo directamente, no cuando se importa:

```
def funcion():
    print("Función llamada")

if __name__ == "__main__":
    print("Este código solo corre si ejecuto este archivo directamente")
    funcion()
```

Esto es útil para pruebas y evitar que cierto código se ejecute al importar el módulo.

14. Resumen rápido

Concepto	Descripción
Módulo	Archivo .py con código Python
import	Importar módulo completo
from ... import	Importar elementos específicos
Alias (as)	Renombrar módulos o funciones
Paquete	Carpeta con módulos y <code>__init__.py</code>
<code>__name__</code>	Variable para distinguir ejecución directa o importada

Sintaxis avanzada

01. Dates

02. Compresión de listas

03. Lambdas

04. Funciones de orden superior

05. Tipos de error

06. Manejo de archivos

07. Expresiones regulares

08. Paquetes de Python

Extra: Creación paquete real en Python

Ejercicios con soluciones detalladas en Python

01. Fechas (dates) en Python

◆ 1. ¿Qué son las fechas en Python?

En Python, las fechas se manejan principalmente con el **módulo `datetime`**, que forma parte de la **biblioteca estándar**. Este módulo permite trabajar con fechas y horas de manera precisa y flexible: crear fechas, modificarlas, hacer cálculos, formatearlas y más.

◆ 2. Módulos principales para trabajar con fechas

- `datetime`: el más usado, maneja fechas y horas.
- `time`: para funciones básicas de tiempo (segundos, reloj).
- `calendar`: útil para obtener calendarios y verificar años bisiestos.
- `dateutil`: módulo externo (más avanzado que `datetime`).

Para comenzar, lo más común y completo es usar:

```
from datetime import date, time, datetime, timedelta
```

◆ 3. Clases principales en `datetime`

Clase	Uso principal
<code>date</code>	Solo fecha: año, mes, día
<code>time</code>	Solo hora: hora, minuto, segundo
<code>datetime</code>	Fecha y hora combinadas
<code>timedelta</code>	Diferencias entre fechas/horas

◆ 4. Sintaxis para crear fechas y horas

Fecha (`date`)

```
from datetime import date

hoy = date.today()
cumple = date(1995, 12, 20) # Año, mes, día

print(hoy)      # 2025-06-08 (por ejemplo)
print(cumple)   # 1995-12-20
```

Hora (`time`)

```
from datetime import time
```

```
hora = time(14, 30, 45) # hora, minuto, segundo
print(hora)             # 14:30:45
```

Fecha y hora (**datetime**)

```
from datetime import datetime

ahora = datetime.now()
momento = datetime(2023, 5, 12, 10, 15, 30)

print(ahora)      # 2025-06-08 10:22:45.123456
print(momento)    # 2023-05-12 10:15:30
```

◆ 5. Acceder a componentes individuales

```
from datetime import datetime

hoy = datetime.now()

print(hoy.year)    # Año actual
print(hoy.month)   # Mes
print(hoy.day)     # Día
print(hoy.hour)    # Hora
print(hoy.minute)  # Minuto
print(hoy.second)  # Segundo
```

◆ 6. Formatear fechas (**strftime**)

Convierte una fecha a **cadena de texto con formato** personalizado:

```
from datetime import datetime

hoy = datetime.now()

formateada = hoy.strftime("%d/%m/%Y %H:%M:%S")
print(formateada) # "08/06/2025 10:30:00"
```

Códigos comunes en **strftime**

Código	Significado	Ejemplo
%d	Día (2 dígitos)	08
%m	Mes (2 dígitos)	06
%Y	Año (4 dígitos)	2025
%H	Hora (24h)	10
%I	Hora (12h)	10
%p	AM/PM	AM

Código	Significado	Ejemplo
%M	Minutos	45
%S	Segundos	30
%A	Día de la semana	Sunday
%B	Nombre del mes	June

◆ 7. Convertir string a fecha (`strptime`)

```
from datetime import datetime

cadena = "20/12/1995 14:30"
fecha = datetime.strptime(cadena, "%d/%m/%Y %H:%M")

print(fecha)  # 1995-12-20 14:30:00
```

◆ 8. Operaciones con fechas (`timedelta`)

La clase `timedelta` permite sumar o restar días, horas, etc.

```
from datetime import datetime, timedelta

hoy = datetime.now()
mañana = hoy + timedelta(days=1)
ayer = hoy - timedelta(days=1)

print("Hoy:", hoy)
print("Mañana:", mañana)
print("Ayer:", ayer)
```

También puedes usar:

```
una_semana = timedelta(weeks=1)
dos_horas = timedelta(hours=2)
quince_min = timedelta(minutes=15)
```

◆ 9. Diferencia entre fechas

```
from datetime import date

hoy = date.today()
cumple = date(1995, 12, 20)

diferencia = hoy - cumple
print(diferencia.days)  # Días desde el cumpleaños
```

◆ 10. Comparación de fechas

Puedes comparar fechas como si fueran números:

```
from datetime import date

hoy = date.today()
futuro = date(2030, 1, 1)

if futuro > hoy:
    print("La fecha es futura")
```

◆ 11. Uso del módulo `calendar`

```
import calendar

print(calendar.month(2025, 6))    # Calendario de junio 2025
print(calendar.isleap(2024))      # True (bisiesto)
```

◆ 12. Características generales

Característica	Descripción
Precisión	Maneja fechas y horas con precisión de microsegundos
Mutable	Los objetos <code>datetime</code> , <code>date</code> y <code>time</code> son inmutables
Comparables	Se pueden comparar con operadores <code>==</code> , <code><</code> , <code>></code> , etc.
Soporta formatos	Se puede convertir a y desde cadenas con formato personalizado
Compatibilidad	Es parte de la biblioteca estándar de Python (no se necesita instalar nada)
Operaciones	Puedes sumar, restar, comparar y formatear fechas fácilmente

◆ 13. Buenas prácticas

- Usa siempre `datetime` cuando necesites **fecha + hora**.
- Usa `date` si solo te interesa la fecha sin hora.
- Usa `timedelta` para calcular diferencias o desplazamientos de tiempo.
- Usa `strftime` y `strptime` para convertir entre fecha y texto.
- Usa `calendar` para tareas más complejas de calendario.

◆ 14. Ejemplo completo

```
from datetime import datetime, timedelta

# Obtener fecha actual
hoy = datetime.now()
```

```
# Formatearla
print("Hoy es:", hoy.strftime("%A %d de %B de %Y"))

# Sumar días
futuro = hoy + timedelta(days=10)
print("Dentro de 10 días será:", futuro.strftime("%d/%m/%Y"))

# Diferencia entre fechas
evento = datetime(2025, 12, 25)
dias_restantes = (evento - hoy).days
print(f"Faltan {dias_restantes} días para Navidad")
```

02. Compresión de listas en Python (List Comprehensions)

1. ¿Qué es una compresión de lista?


Es una **forma concisa y elegante de crear listas** en Python, usando una única línea de código. Permite transformar o filtrar elementos de una secuencia (como una lista, tupla o rango), de manera muy eficiente.

2. ¿Para qué sirve?

- Generar listas nuevas de forma rápida.
- Aplicar transformaciones a elementos.
- Filtrar elementos usando condiciones.
- Evitar bucles `for` largos o innecesarios.

3. Sintaxis básica

```
[expresión for elemento in iterable]
```

 Equivalente a:

```
resultado = []
for elemento in iterable:
    resultado.append(expresión)
```

Ejemplo simple

```
cuadrados = [x**2 for x in range(5)]
print(cuadrados)  # [0, 1, 4, 9, 16]
```

4. Con condicional (**if**)

Filtrar elementos que cumplan una condición:

```
[expresión for elemento in iterable if condición]
```

Ejemplo: Solo pares

```
pares = [x for x in range(10) if x % 2 == 0]
print(pares)  # [0, 2, 4, 6, 8]
```

5. Condicional ternario (**if else**) en la expresión

Cuando quieres aplicar una transformación diferente según una condición:

```
[valor_si if condición else valor_no for elemento in iterable]
```

Ejemplo: "par" o "impar"

```
paridad = ["par" if x % 2 == 0 else "impar" for x in range(5)]
print(paridad)  # ['par', 'impar', 'par', 'impar', 'par']
```

6. Con múltiples bucles **for** (anidados)

```
[expresión for i in iterable1 for j in iterable2]
```

Ejemplo: Producto cartesiano

```
combinaciones = [(x, y) for x in [1, 2] for y in ['a', 'b']]
print(combinaciones)  # [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
```

7. Usos frecuentes

Tarea	Ejemplo
Crear una lista de números	<code>[x for x in range(10)]</code>
Transformar valores	<code>[x**2 for x in lista]</code>
Filtrar valores	<code>[x for x in lista if x > 5]</code>
Convertir texto a mayúsculas	<code>[palabra.upper() for palabra in lista]</code>
Eliminar espacios	<code>[s.strip() for s in lista_de_strings]</code>
Convertir tipos	<code>[int(x) for x in lista_de_strings]</code>


8. Comparación entre bucle `for` y comprensión de lista

Código tradicional:

```
cuadrados = []  
for x in range(5):  
    cuadrados.append(x**2)
```

Compresión:

```
cuadrados = [x**2 for x in range(5)]
```

 **Resultado idéntico**, pero más compacto y legible.

9. Características clave

Característica	Descripción
Compacta	Menos líneas de código
Legible	Si no es muy compleja, es más fácil de leer
Funcional	Muy útil para mapeo y filtrado de elementos
Rápida	Generalmente más rápida que los bucles tradicionales
No mutable	Crea una nueva lista sin modificar la original
Expresiva	Puede anidar bucles y condiciones

10. Buenas prácticas

 Usa comprensión de lista cuando:

- La lógica es simple y clara.
- Estás transformando o filtrando elementos.

 Evítala cuando:

- Hay mucha lógica o condiciones complejas.
- La línea se vuelve demasiado larga o difícil de leer.

□ 11. Ejemplos avanzados

◆ Eliminar duplicados y convertir a enteros

```
lista = ['1', '2', '2', '3']
resultado = list(set([int(x) for x in lista]))
print(resultado) # [1, 2, 3]
```

◆ Convertir texto a lista de caracteres en mayúscula

```
texto = "python"
letras = [letra.upper() for letra in texto]
print(letras) # ['P', 'Y', 'T', 'H', 'O', 'N']
matriz = [[x * y for x in range(3)] for y in range(3)]
print(matriz)
# [[0, 0, 0], [0, 1, 2], [0, 2, 4]]
```



12. Resumen

Elemento	Ejemplo
Básico	<code>[x for x in iterable]</code>
Con condición	<code>[x for x in iterable if condición]</code>
Condición ternaria	<code>[a if cond else b for x in iterable]</code>
Varios bucles	<code>[f(x, y) for x in A for y in B]</code>
Múltiples usos	Conversión, filtrado, limpieza de datos

03. Funciones Lambda en Python

✓ 1. ¿Qué es una función `lambda`?

Una función `lambda` es una **función anónima, pequeña y de una sola línea** que se usa para realizar operaciones simples sin tener que definir una función completa con `def`.

✧ Se usa comúnmente con funciones como `map()`, `filter()`, `sorted()`, y `reduce()`.

📦 2. Sintaxis

```
lambda argumentos: expresión
```

✧ **Importante:**

- Puede tener **cualquier número de argumentos**, pero **solo una expresión**.
- No necesita `return`: la expresión se evalúa y se devuelve automáticamente.

◆ Ejemplo simple:

```
suma = lambda x, y: x + y  
print(suma(3, 5)) # 8
```

Equivalente a:

```
def suma(x, y):  
    return x + y
```

📦 3. ¿Para qué se usa?

Las `lambda` son útiles para:

- Crear funciones simples de una sola línea.
- Usarlas como **argumentos de funciones de orden superior**.
- Evitar definiciones largas para funciones que se usan una sola vez.

📦 4. Ejemplos comunes

◆ Doblar un número:

```
doble = lambda x: x * 2  
print(doble(4)) # 8
```

Elemento

Ejemplo

◆ Usar con `map()` : aplicar a cada elemento

```
numeros = [1, 2, 3, 4]
cuadrados = list(map(lambda x: x**2, numeros))
print(cuadrados) # [1, 4, 9, 16]
```

◆ Usar con `filter()` : filtrar elementos

```
pares = list(filter(lambda x: x % 2 == 0, numeros))
print(pares) # [2, 4]
```

◆ Usar con `sorted()` : ordenar con clave personalizada

```
nombres = ['Ana', 'Luis', 'Pedro', 'Carlos']
ordenado = sorted(nombres, key=lambda x: len(x))
print(ordenado) # ['Ana', 'Luis', 'Pedro', 'Carlos']
```



5. Características de `lambda`

Característica	Descripción
Anónima	No tiene nombre (a menos que la guardes en una variable).
Función ligera	Ideal para funciones cortas y de una sola operación.
Retorna automáticamente	No necesita la palabra <code>return</code> .
Solo una expresión	No se pueden escribir bloques complejos ni varias líneas.
Reemplazo temporal	Se usa generalmente en funciones de paso rápido (como argumentos).

⊘ 6. Limitaciones

- No puedes usar instrucciones múltiples como `if-else` en bloques (solo ternario).
- No se recomienda para funciones complejas.
- No tiene nombre, lo que dificulta su depuración si da errores.
- No puedes usar `try-except`, ni bucles `for`, ni `while`.

Elemento Ejemplo

7. Diferencia entre `lambda` y `def`

Aspecto	<code>def</code>	<code>lambda</code>
Nombre	Necesita un nombre	Puede ser anónima
Longitud	Puede tener varias líneas	Solo una expresión
Funcionalidad	Soporta lógica compleja	Solo lógica simple
Claridad	Más clara y mantenible	Más compacta, pero menos legible
Uso	General y estructurado	Casos rápidos y puntuales

8. Ejemplo con `if-else` en `lambda` (condición ternaria)

```
mayor = lambda x, y: x if x > y else y
print(mayor(5, 8)) # 8
```

9. Ejemplo con `lambda` + `map`, `filter`, `reduce`

```
from functools import reduce

lista = [1, 2, 3, 4, 5]

# map: elevar al cuadrado
print(list(map(lambda x: x**2, lista))) # [1, 4, 9, 16, 25]

# filter: solo impares
print(list(filter(lambda x: x % 2 != 0, lista))) # [1, 3, 5]

# reduce: multiplicar todos
print(reduce(lambda x, y: x * y, lista)) # 120
```

10. Ejercicios propuestos

¿Te gustaría intentar estos?

1. Usa `lambda` para convertir una lista de temperaturas en °C a °F.
2. Usa `lambda` con `sorted()` para ordenar una lista de tuplas por el segundo valor.
3. Filtra una lista de palabras y deja solo las que empiecen con "a".

Elemento

Ejemplo

□ 11. Resumen

Tema	Detalles
Sintaxis	<code>lambda args: expresión</code>
Número de expresiones	Solo una
Usos comunes	<code>map()</code> , <code>filter()</code> , <code>sorted()</code> , <code>reduce()</code>
Ventajas	Rápida, compacta, útil en una sola línea
Desventajas	No clara en funciones complejas

04. Funciones de Orden Superior en Python

✓ 1. ¿Qué es una función de orden superior?

Una **función de orden superior** es una función que cumple **al menos una de las siguientes características**:

1. **Recibe una o más funciones como argumentos.**
2. **Devuelve una función** como resultado.

En otras palabras, las funciones de orden superior son aquellas que permiten trabajar con funciones dentro de otras funciones.

🌀 2. ¿Para qué se usan?

- Permiten escribir **código más abstracto, flexible y reutilizable**.
- Son muy útiles para funciones como `map()`, `filter()`, `reduce()`, y otras funciones en Python que toman otras funciones como argumento.
- Facilitan la creación de **comportamientos dinámicos**, como callbacks o manejadores de eventos.

📦 3. Sintaxis

🌐 Función de orden superior que recibe funciones

Una función que **recibe funciones como parámetros** es comúnmente usada con una o más funciones anidadas.

```
def orden_superior(funcion, valor):
    return funcion(valor)
```

Ejemplo:

```
def aplicar_funcion(func, x):  
    return func(x)  
  
# Definimos una función simple  
def cuadrado(x):  
    return x ** 2  
  
resultado = aplicar_funcion(cuadrado, 5)  
print(resultado) # 25
```

Aquí, `aplicar_funcion` es una **función de orden superior** que toma `cuadrado` como argumento y lo ejecuta con el valor 5.

4. Función que devuelve otra función

A veces, una función de orden superior **devuelve otra función** como resultado. Esto es útil cuando se quieren crear funciones con comportamiento dinámico o personalizado.

```
def multiplicar_por(n):  
    def multiplicar(x):  
        return x * n  
    return multiplicar
```

Ejemplo:

```
# Creamos una función que multiplica por 2  
multiplica_por_2 = multiplicar_por(2)  
  
# Usamos la función devuelta  
resultado = multiplica_por_2(5)  
print(resultado) # 10
```

En este caso, la función `multiplicar_por` es una función de orden superior, ya que devuelve la función `multiplicar` con un comportamiento específico.

5. Ejemplos comunes de funciones de orden superior

`map()`:

Aplica una función a todos los elementos de un iterable (como una lista).

```
numeros = [1, 2, 3, 4]  
cuadrados = list(map(lambda x: x**2, numeros))  
print(cuadrados) # [1, 4, 9, 16]
```

`filter()`:

Filtra los elementos de un iterable según una función condicional (devuelve los que cumplen la condición).

```
numeros = [1, 2, 3, 4, 5]
pares = list(filter(lambda x: x % 2 == 0, numeros))
print(pares)  # [2, 4]
```

◆ **reduce():**

Aplica una función acumulativa sobre los elementos de un iterable, reduciendo el iterable a un único valor.

```
from functools import reduce

numeros = [1, 2, 3, 4]
producto = reduce(lambda x, y: x * y, numeros)
print(producto)  # 24
```

📐 6. Características de las funciones de orden superior

Característica	Descripción
Reciben funciones como argumentos	Permiten pasar funciones como parámetros.
Devuelven funciones	Pueden devolver funciones para su uso posterior.
Abstracción	Ayudan a crear código más general y reutilizable.
Flexibilidad	Permiten dinámicamente cambiar el comportamiento de un programa.

📐 7. Uso de funciones anónimas (`lambda`) en funciones de orden superior

Las funciones de orden superior a menudo se combinan con **funciones anónimas** (`lambda`) para realizar operaciones rápidas de transformación o filtrado de elementos.

✂ Ejemplo de `map()` con `lambda`:

```
numeros = [1, 2, 3, 4]
dobles = list(map(lambda x: x * 2, numeros))
print(dobles)  # [2, 4, 6, 8]
```

🤖 8. Ejemplos de combinaciones útiles

◆ Usar `map()` con una función de orden superior

```
def aplicar_a_lista(func, lista):
    return list(map(func, lista))

# Definimos una función para elevar al cuadrado
```

```
def cuadrado(x):  
    return x ** 2  
  
resultado = aplicar_a_lista(cuadrado, [1, 2, 3, 4])  
print(resultado)  # [1, 4, 9, 16]
```

◆ Usar `filter()` con una función de orden superior

```
def filtrar_pares(lista):  
    return list(filter(lambda x: x % 2 == 0, lista))  
  
resultado = filtrar_pares([1, 2, 3, 4, 5, 6])  
print(resultado)  # [2, 4, 6]
```

🔗 9. Funciones de orden superior en bibliotecas

Las funciones de orden superior son **comunes en muchas bibliotecas estándar** de Python y en programación funcional.

◆ `sorted()` con una función de orden superior:

```
nombres = ['Ana', 'Luis', 'Pedro', 'Carlos']  
ordenado = sorted(nombres, key=lambda x: len(x))  
print(ordenado)  # ['Ana', 'Luis', 'Pedro', 'Carlos']
```

🚀 10. Usos avanzados: Creación de funciones dinámicas

Las funciones de orden superior son muy útiles cuando se desea crear **funciones personalizadas** de manera dinámica.

🔗 Ejemplo: Crear funciones para multiplicar por diferentes valores

```
def generador_multiplicador(n):  
    return lambda x: x * n  
  
# Crear funciones multiplicadoras  
multiplica_por_2 = generador_multiplicador(2)  
multiplica_por_3 = generador_multiplicador(3)  
  
print(multiplica_por_2(5))  # 10  
print(multiplica_por_3(5))  # 15
```

□ 11. Resumen

Las **funciones de orden superior** en Python son una poderosa herramienta para:

- **Recibir funciones como argumentos.**
- **Devolver funciones** para su uso posterior.
- Proporcionan **flexibilidad, abstracción y reutilización** en tu código.

Función de orden superior	Ejemplo
<code>map()</code>	<code>list(map(func, lista))</code>
<code>filter()</code>	<code>list(filter(func, lista))</code>
<code>reduce()</code>	<code>reduce(func, lista)</code>
Función que recibe funciones	<code>def orden_superior(func, x): return func(x)</code>
Función que devuelve función	<code>def generador(func): return lambda x: func(x)</code>

05. Tipos de Errores en Python

✓ ¿Qué es un error?

En programación, un **error** es cualquier condición inesperada que hace que el programa se detenga, funcione incorrectamente o produzca un resultado no deseado.

En Python, los errores se dividen principalmente en dos grandes categorías:

- ◆ 1. Errores de sintaxis (Syntax Errors)
- ◆ 2. Excepciones o errores en tiempo de ejecución (Runtime Errors)

□ 1. Errores de Sintaxis (`SyntaxError`)

✂ ¿Qué son?

Errores que ocurren cuando el **código no sigue las reglas del lenguaje Python**.

🔑 Características:

- Detectados **antes de ejecutar** el programa.
- Siempre generan una interrupción.
- Impiden la ejecución completa.

□ Ejemplo:

```
if True
    print("Hola")
```

Error:

```
SyntaxError: expected ':'
```

⚠ 2. Excepciones o Errores en Tiempo de Ejecución

🔗 ¿Qué son?

Errores que ocurren **durante la ejecución del programa**, una vez que la sintaxis ha sido verificada como válida.

📦 Tipos de errores más comunes (con ejemplos)

Error	Significado	Ejemplo
<code>SyntaxError</code>	Error de sintaxis	<code>if x = 5:</code>
<code>NameError</code>	Variable o función no definida	<code>print(valor)</code>
<code>TypeError</code>	Tipo incorrecto de dato u operación	<code>"5" + 2</code>
<code>ValueError</code>	Valor inválido para la operación	<code>int("abc")</code>
<code>IndexError</code>	Índice fuera de rango	<code>lista[10]</code> en una lista con 3 elementos
<code>KeyError</code>	Clave no existente en un diccionario	<code>dic["clave_no_existente"]</code>
<code>AttributeError</code>	Objeto sin atributo especificado	<code>"hola".append("mundo")</code>
<code>ZeroDivisionError</code>	División por cero	<code>5 / 0</code>
<code>ImportError / ModuleNotFoundError</code>	Fallo al importar un módulo	<code>import modulo_que_no_existe</code>
<code>IndentationError</code>	Mala indentación del código	Código mal tabulado

📦 3. Cómo manejar errores: `try — except`

Python permite **atrapar errores** y manejarlos para que el programa no se detenga bruscamente.

📦 Sintaxis básica:

```
try:
    # Código que puede causar un error
```

```
resultado = 10 / 0
except ZeroDivisionError:
    print(";No puedes dividir por cero!")
```

❏ Múltiples excepciones:

```
try:
    numero = int("abc")
except ValueError:
    print("Ese valor no es un número válido.")
except TypeError:
    print("Tipo incorrecto.")
```

🔗 4. Bloques adicionales: `else` y `finally`

◆ `else`: se ejecuta si no hay error

```
try:
    print("Todo bien")
except:
    print("Algo falló")
else:
    print("No hubo errores")
```

◆ `finally`: se ejecuta siempre, haya error o no

```
try:
    resultado = 5 / 0
except ZeroDivisionError:
    print("Error detectado.")
finally:
    print("Esto se ejecuta siempre.")
```

🔗 5. Personalizar errores: `raise`

Puedes **lanzar errores manualmente** si detectas condiciones inválidas en tu propio código.

❏ Ejemplo:

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

❏ 6. Crear tus propias excepciones (clases personalizadas)

```
class MiErrorPersonal(Exception):
    pass

try:
    raise MiErrorPersonal("Esto es un error personalizado")
except MiErrorPersonal as e:
    print(f"Error capturado: {e}")
```


▣ 7. Características de los errores en Python

Característica	Descripción
Tipados	Cada error es una clase (objeto) que hereda de <code>BaseException</code>
Jerárquicos	Todos los errores forman una jerarquía de clases
Controlables	Puedes atraparlos y manejar su comportamiento
Personalizables	Puedes crear tus propias excepciones

🔺 8. Jerarquía simplificada de errores

```
BaseException
├── SystemExit
├── KeyboardInterrupt
├── Exception
│   ├── ArithmeticError
│   │   ├── ZeroDivisionError
│   │   └── OverflowError
│   ├── LookupError
│   │   ├── IndexError
│   │   └── KeyError
│   ├── ValueError
│   ├── TypeError
│   ├── NameError
│   └── ...
```

🔗 9. Buenas prácticas

- Usa bloques `try-except` solo donde lo necesites.
- Siempre maneja errores específicos (evita `except:` sin tipo).
- Usa `finally` para cerrar archivos, conexiones o liberar recursos.
- Documenta qué errores pueden aparecer en tus funciones.
- No ocultes errores importantes sin `log` o mensaje.

🔑 10. Ejemplo completo

```
def dividir(a, b):
    try:
        resultado = a / b
    except ZeroDivisionError:
        return "No se puede dividir por cero"
    except TypeError:
        return "Ambos valores deben ser números"
    else:
        return resultado
    finally:
        print("Intento de división completado.")
```

```
print(dividir(10, 2))    # 5.0
print(dividir(10, 0))    # No se puede dividir por cero
print(dividir("10", 2))  # Ambos valores deben ser números
```

□ 11. Práctica recomendada

Puedes practicar errores provocándolos intencionalmente y capturándolos:

```
errores = [
    lambda: 1/0,
    lambda: int("no-numero"),
    lambda: [1,2][10],
    lambda: {"a":1}["b"],
    lambda: "texto".append("otro")
]

for prueba in errores:
    try:
        prueba()
    except Exception as e:
        print(f"Error capturado: {e}")
```

□ Resumen

Tema	Descripción
Errores de sintaxis	Detectados antes de ejecutar, como <code>SyntaxError</code> , <code>IndentationError</code>
Errores de ejecución	Ocurren durante la ejecución, como <code>TypeError</code> , <code>ZeroDivisionError</code>
Manejarlos	Se usa <code>try-except</code> para prevenir que el programa falle
Personalización	Puedes lanzar (<code>raise</code>) y crear (<code>class</code>) tus propios errores

06. Manejo de Archivos en Python

□ ¿Qué es el manejo de archivos?

El manejo de archivos permite a un programa **interactuar con archivos externos**, ya sea para:

- **Leer** contenido.
- **Escribir** nuevo contenido.
- **Actualizar** información.
- **Guardar resultados**.

1. Abrir archivos con `open()`

✦ **Sintaxis general:**

```
archivo = open("ruta_del_archivo", "modo")
```

Parámetro

Significado

"ruta_del_archivo" Ruta del archivo (puede ser relativa o absoluta)

"modo" Modo de apertura (ver tabla abajo)

2. Modos de apertura de archivos

Modo	Significado	Descripción
'r'	Read	Solo lectura (default). Error si el archivo no existe.
'w'	Write	Escritura. Crea archivo nuevo o sobrescribe el existente.
'a'	Append	Añadir al final del archivo. Lo crea si no existe.
'x'	Exclusive creation	Crea un archivo nuevo, error si ya existe.
'b'	Binary	Usado junto con otros (rb, wb) para archivos binarios.
't'	Text	(Default) abre en modo texto.

💡 Ejemplo:

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

3. Leer archivos

📖 Métodos comunes de lectura:

```
archivo.read()          # Lee todo el contenido
archivo.readline()      # Lee una línea
archivo.readlines()     # Devuelve una lista con todas las líneas
```

📖 Ejemplo:


```
archivo = open("datos.txt", "r")
contenido = archivo.read()
print(contenido)
archivo.close()
```

4. Escribir en archivos

💎 **write():** escribe una cadena

writelines(): escribe una lista de cadenas

```
archivo = open("nuevo.txt", "w")
archivo.write("Hola, mundo\n")
archivo.writelines(["Línea 1\n", "Línea 2\n"])
archivo.close()
```

 Si el archivo ya existía y se abre con "w", se **borra todo su contenido**.

5. Añadir al final (append)

```
archivo = open("log.txt", "a")
archivo.write("Nueva línea agregada\n")
archivo.close()
```

6. Buenas prácticas con with

Usar with gestiona automáticamente el **cerrado del archivo**, incluso si hay errores:

```
with open("datos.txt", "r") as archivo:
    contenido = archivo.read()
    print(contenido)
```

Es equivalente a:

```
archivo = open("datos.txt", "r")
try:
    contenido = archivo.read()
finally:
    archivo.close()
```

7. Operaciones comunes

Comprobar si un archivo existe

```
import os

if os.path.exists("archivo.txt"):
    print("Existe")
else:
    print("No existe")
```

Eliminar archivos

```
import os
os.remove("archivo.txt")
```

8. Leer línea por línea (ideal para archivos grandes)

```
with open("grande.txt", "r") as archivo:
```

```
for linea in archivo:
    print(linea.strip())
```

9. Modificar un archivo

No se puede **editar directamente** el contenido en una posición específica. Se debe:

1. Leer el archivo.
2. Modificar el contenido en memoria.
3. Volver a escribir el archivo.

```
# Leer
with open("datos.txt", "r") as archivo:
    lineas = archivo.readlines()

# Modificar
lineas[1] = "Línea modificada\n"

# Escribir
with open("datos.txt", "w") as archivo:
    archivo.writelines(lineas)
```

10. Leer/escribir archivos CSV

```
import csv

# Escribir
with open("personas.csv", "w", newline="") as archivo:
    escritor = csv.writer(archivo)
    escritor.writerow(["Nombre", "Edad"])
    escritor.writerow(["Ana", 25])
    escritor.writerow(["Luis", 30])

# Leer
with open("personas.csv", "r") as archivo:
    lector = csv.reader(archivo)
    for fila in lector:
        print(fila)
```

11. Archivos binarios (imágenes, PDF, etc.)

```
# Copiar una imagen
with open("foto.jpg", "rb") as origen:
    contenido = origen.read()

with open("copia.jpg", "wb") as destino:
    destino.write(contenido)
```

Características generales del manejo de archivos

Característica	Descripción
Persistencia	Permite guardar información entre ejecuciones.
Flujo	Acceso secuencial o controlado al contenido.
Versatilidad	Admite texto, binarios, CSV, JSON, etc.
Modularidad	Puedes usar módulos como <code>csv</code> , <code>json</code> , <code>os</code> , <code>pathlib</code>

□ Ejemplo completo

```
def registrar_usuario(nombre):
    with open("usuarios.txt", "a") as archivo:
        archivo.write(nombre + "\n")

def mostrar_usuarios():
    with open("usuarios.txt", "r") as archivo:
        for linea in archivo:
            print("Usuario:", linea.strip())

registrar_usuario("Juan")
registrar_usuario("Lucía")
mostrar_usuarios()
```

□ Resumen final

Acción	Código
Abrir archivo	<code>open("archivo.txt", "r")</code>
Leer todo	<code>.read()</code>
Leer línea	<code>.readline()</code>
Leer lista de líneas	<code>.readlines()</code>
Escribir texto	<code>.write("texto")</code>
Añadir al final	abrir con <code>"a"</code>
Cerrar archivo	<code>.close()</code> o <code>with</code>
Leer CSV	módulo <code>csv.reader()</code>
Leer binario	<code>open(..., "rb")</code>

07. Expresiones Regulares en Python (RegEx)

□ ¿Qué son?

Son **patrones de texto** que se usan para hacer coincidencias dentro de cadenas. Se basan en una **sintaxis especial** para identificar estructuras complejas como:

- Correos electrónicos
- Teléfonos
- Fechas
- Palabras específicas
- Etiquetas HTML, etc.

Se usan con el módulo incorporado `re`.

□ Importación del módulo

```
import re
```

□ 1. Funciones principales del módulo `re`

Función	Descripción
<code>re.match()</code>	Busca al inicio del texto
<code>re.search()</code>	Busca en cualquier parte del texto
<code>re.findall()</code>	Devuelve una lista de todas las coincidencias
<code>re.finditer()</code>	Devuelve un iterador con todos los objetos coincidentes
<code>re.sub()</code>	Sustituye coincidencias por otro texto
<code>re.split()</code>	Divide una cadena por el patrón dado
<code>re.compile()</code>	Compila una expresión para reutilizarla

2. Sintaxis básica de expresiones regulares

Patrón	Significado	Ejemplo
<code>.</code>	Cualquier carácter excepto nueva línea	<code>a.c</code> → "abc", "a-c"
<code>^</code>	Comienza con	<code>^Hola</code> → "Hola mundo"
<code>\$</code>	Termina con	<code>mundo\$</code> → "Hola mundo"

Patrón	Significado	Ejemplo
*	0 o más repeticiones	<code>a*</code> → "", "a", "aaa"
+	1 o más repeticiones	<code>a+</code> → "a", "aa"
?	0 o 1 repetición	<code>colou?r</code> → "color", "colour"
{n}	Exactamente n repeticiones	<code>a{3}</code> → "aaa"
{n, }	Al menos n repeticiones	<code>a{2, }</code> → "aa", "aaa"
{n, m}	Entre n y m repeticiones	<code>a{2, 4}</code> → "aa", "aaa", "aaaa"
[]	Cualquier carácter dentro	<code>[aeiou]</code> → "a", "e", etc.
[^]	Cualquier carácter excepto los listados	<code>[^0-9]</code> → no números
\d	Dígito ([0-9])	
\D	No dígito ([^0-9])	
\w	Alfanumérico ([a-zA-Z0-9_])	
\W	No alfanumérico	
\s	Espacio en blanco	
\S	No espacio en blanco	
,	,	"O" lógico
(...)	Agrupar patrones	<code>(\d{3})-(\d{2})</code>

□ 3. Ejemplos detallados

match() VS search()

```

texto = "Hola mundo"

re.match("Hola", texto)      # Coincide (inicio)
re.match("mundo", texto)     # None

re.search("mundo", texto)    # Coincide (en cualquier parte)

```

findall()

```

re.findall(r'\d+', "Hay 12 manzanas y 7 peras")
# ['12', '7']

```

finditer()

```

for match in re.finditer(r'\d+', "12 y 34 y 56"):

```



```
print(match.group(), match.start(), match.end())
```

sub() (reemplazo)

```
texto = "Mi teléfono es 123-456-7890"
nuevo = re.sub(r'\d', 'X', texto)
print(nuevo)
# Mi teléfono es XXX-XXX-XXXX
```

split() (división)

```
texto = "uno,dos,tres|cuatro"
re.split(r'[;|,]', texto)
# ['uno', 'dos', 'tres', 'cuatro']
```

□ 4. Uso de re.compile()

Permite **reutilizar** patrones y mejorar el rendimiento:

```
patron = re.compile(r"\d{4}-\d{2}-\d{2}")
fecha = "La fecha es 2025-06-08"
print(patron.search(fecha).group())
```

□ 5. Validaciones comunes

✓ Validar email

```
regex = r'^[\w\.-]+@[\w\.-]+\.\w+$'
email = "ejemplo@gmail.com"
print(re.match(regex, email)) # Coincide si es válido
```

✓ Validar número de teléfono

```
regex = r'^\d{3}-\d{3}-\d{4}$'
telefono = "123-456-7890"
```

✓ Validar contraseña fuerte (ejemplo básico)

```
regex = r'^(?=.*[A-Z])(?=.*\d)[A-Za-z\d]{8,}$'
# Mínimo 8 caracteres, al menos una mayúscula y un número
```

💡 Características de las expresiones regulares

Característica	Detalle
Potentes	Permiten búsquedas y transformaciones avanzadas
Crípticas	A veces difíciles de leer para principiantes

Característica	Detalle
Flexibles	Útiles para validar, dividir, buscar, limpiar texto
Universales	Se usan en muchos lenguajes de programación
Lentas si mal usadas	Evita patrones muy generales o complejos

❑ Bonus: Bandera `re.IGNORECASE`

Hace que la búsqueda **no distinga mayúsculas/minúsculas**:

```
re.search("hola", "HOLA", re.IGNORECASE)
```

También existen:

- `re.MULTILINE`: `^` y `$` funcionan por línea.
- `re.DOTALL`: `.` también coincide con `\n`.

❑ Ejemplo completo

```
import re

def extraer_emails(texto):
    patron = r'[\w\.-]+@[\w\.-]+\.\w+'
    return re.findall(patron, texto)

mensaje = "Contáctanos: soporte@empresa.com o ventas@empresa.com"
emails = extraer_emails(mensaje)
print("Correos encontrados:", emails)
```

✓ Resumen rápido

Acción	Función
Buscar al inicio	<code>re.match()</code>
Buscar en cualquier parte	<code>re.search()</code>
Todas las coincidencias	<code>re.findall()</code>
Iterar coincidencias	<code>re.finditer()</code>
Reemplazar	<code>re.sub()</code>
Dividir texto	<code>re.split()</code>
Compilar patrón	<code>re.compile()</code>

08. Paquetes en Python

□ ¿Qué es un paquete?

Un **paquete** en Python es una **colección de módulos organizados dentro de directorios** que permite **estructurar y reutilizar código**.

- Un **módulo** es un archivo `.py` que contiene funciones, clases, variables, etc.
- Un **paquete** es un **directorio que contiene un archivo especial llamado `__init__.py`** (aunque en versiones modernas es opcional) y uno o más módulos.

📁 Estructura básica de un paquete

Supongamos que tienes este paquete llamado `mimatematica`:

```
mimatematica/
├── __init__.py
├── suma.py
├── resta.py
├── algebra/
│   ├── __init__.py
│   └── ecuaciones.py
```

□ Función de `__init__.py`

- Indica que ese directorio es un **paquete importable**.
- Puede estar vacío o contener código que **se ejecuta al importar** el paquete.
- Sirve también para **exponer funciones/módulos al exterior**.

✓ Características de los paquetes

Característica	Detalle
Reutilización	Permite usar el código en múltiples proyectos
Organización	Agrupar módulos por funcionalidad
Escalabilidad	Ideal para proyectos grandes
Modularidad	Cada módulo puede trabajar de forma independiente
Distribución	Fácil de compartir como biblioteca en PyPI u otros proyectos

Sintaxis para importar paquetes y módulos

1. Importar todo un módulo


```
import mimatematica.suma
mimatematica.suma.sumar(3, 4)
```

2. Importar una función específica

```
from mimatematica.suma import sumar
sumar(3, 4)
```

3. Importar todo desde un módulo

```
from mimatematica.suma import *
```

 *Esto puede causar conflictos si hay funciones con el mismo nombre en otros módulos.*

□ Ejemplo práctico

Estructura del paquete:

```
calculadora/
├── __init__.py
├── suma.py
└── resta.py
```

suma.py

```
def sumar(a, b):
    return a + b
```

resta.py

```
def restar(a, b):
    return a - b
```

__init__.py

```
from .suma import sumar
from .resta import restar
```

□ Uso:

```
from calculadora import sumar, restar

print(sumar(5, 3)) # 8
print(restar(5, 3)) # 2
```

Importaciones relativas vs absolutas

Importación relativa

- Se usa **dentro de un paquete** para importar otros módulos del mismo nivel o subniveles.

```
from .resta import restar      # mismo nivel
from ..trigonometria import seno # un nivel arriba
```

Importación absoluta

```
from calculadora.resta import restar
```

- ◆ La relativa es útil en librerías empaquetadas.
- ◆ La absoluta es más clara en proyectos grandes.

Crear tu propio paquete para PyPI

1. Estructura del directorio:

```
mipaquete/
├── mipaquete/
│   ├── __init__.py
│   └── funciones.py
├── setup.py
└── README.md
```

2. setup.py (simplificado):

```
from setuptools import setup, find_packages

setup(
    name="mipaquete",
    version="0.1",
    packages=find_packages(),
    install_requires=[],
)
```

3. Instalación local:

```
pip install .
```

4. Publicación en PyPI:

- Crear cuenta en <https://pypi.org>
- Usar herramientas como `twine` para subir el paquete.

□ Operaciones comunes con paquetes

Operación	Código
Crear paquete	Carpeta con <code>__init__.py</code>
Importar módulo	<code>import paquete.modulo</code>
Importar función	<code>from paquete.modulo import funcion</code>
Usar alias	<code>import paquete as pk</code>
Crear subpaquetes	Carpetas dentro de paquetes con <code>__init__.py</code>

✂ Buenas prácticas

- Usa nombres **cortos y descriptivos** para paquetes y módulos.
- Evita usar `*` en importaciones (`from X import *`).
- Organiza funciones relacionadas en un mismo módulo.
- Usa `__init__.py` para definir una API clara y sencilla.

📌 Resumen

Concepto	Explicación
Módulo	Archivo <code>.py</code> con código reutilizable
Paquete	Carpeta con módulos y <code>__init__.py</code>
Subpaquete	Paquete dentro de otro paquete
<code>__init__.py</code>	Marca el directorio como paquete
Importaciones	<code>import</code> , <code>from</code> , <code>alias</code> , relativas o absolutas

Extra: Creación paquete real en Python.

✓ PASO 1: Estructura del Proyecto

Crea la siguiente estructura de carpetas:

```
mi_paquete/  
├── mi_paquete/  
│   ├── __init__.py  
│   ├── operaciones.py  
│   └── saludos.py  
├── tests/  
│   └── test_operaciones.py  
├── setup.py  
├── README.md  
└── requirements.txt
```

✓ PASO 2: Código de ejemplo

operaciones.py

```
def sumar(a, b):  
    return a + b
```

```
def restar(a, b):  
    return a - b
```

saludos.py

```
def saludar(nombre):  
    return f"Hola, {nombre}!"
```

__init__.py

Este archivo convierte el directorio `mi_paquete/` en un paquete:

```
from .operaciones import sumar, restar  
from .saludos import saludar
```

✓ PASO 3: Archivo **setup.py**

Este es el archivo que define los metadatos del paquete:

```
from setuptools import setup, find_packages  
  
setup(  
    name='mi_paquete',  
    version='0.1',  
    packages=find_packages(),  
    install_requires=[],  
    author='Tu Nombre',  
    author_email='tuemail@example.com',
```

```
description='Un paquete de ejemplo con operaciones matemáticas y saludos',
long_description=open('README.md').read(),
long_description_content_type='text/markdown',
url='https://github.com/tuusuario/mi_paquete',
classifiers=[
    'Programming Language :: Python :: 3',
    'License :: OSI Approved :: MIT License'
],
)
```

✓ PASO 4: Archivo `README.md`

```
# mi_paquete
```

Un paquete de ejemplo con funciones matemáticas y saludos.

```
## Instalación
```

```
pip install .
```

Uso

```
from mi_paquete import sumar, saludar
```

```
print(sumar(2, 3))
print(saludar("Juan"))
```

```
---
```

```
## ✓ PASO 5: Pruebas básicas (opcional)
```

Crea un archivo de pruebas:

```
### `tests/test_operaciones.py`
```

```
```python
from mi_paquete import sumar, restar
```

```
def test_sumar():
 assert sumar(2, 3) == 5
```

```
def test_restar():
 assert restar(5, 2) == 3
```

Ejecuta con `pytest`:

```
pip install pytest
pytest tests/
```

### ✓ PASO 6: Instalar y probar el paquete localmente

Desde el directorio raíz:

```
pip install .
```



Y luego puedes usarlo en cualquier otro archivo Python:

```
from mi_paquete import sumar, saludar

print(sumar(5, 5))
print(saludar("Ana"))
```

## ✓ PASO 7: Publicar en PyPI (opcional)

1. Regístrate en <https://pypi.org>
2. Instala herramientas:

```
pip install twine setuptools wheel
```

3. Genera los archivos:

```
python setup.py sdist bdist_wheel
```

4. Publica:

```
twine upload dist/*
```

## □ EJERCICIOS CON SOLUCIONES DETALLADAS EN PYTHON

### 1. VARIABLES

#### Ejercicio 1: Asignación básica

Crea tres variables:

- nombre con tu nombre
- edad con tu edad
- altura con tu altura en metros

Muestra un mensaje como:

Me llamo Juan, tengo 25 años y mido 1.75 metros.

#### ✓ Solución:

```
nombre = "Juan"
edad = 25
altura = 1.75

print(f"Me llamo {nombre}, tengo {edad} años y mido {altura} metros.")
```

### Explicación:

- Se usan **strings** para el nombre.
- Se usa un **f-string** para formatear el mensaje de forma clara y legible.

### Ejercicio 2: Intercambiar valores

Intercambia el valor de dos variables  $a = 10$  y  $b = 20$ .

### Solución:

```
a = 10
b = 20

a, b = b, a # Intercambio

print("a =", a) # 20
print("b =", b) # 10
```

### Explicación:

- La sintaxis `a, b = b, a` permite intercambiar valores de forma directa sin usar variables auxiliares.

---

## 2. + OPERADORES

### Ejercicio 2: Operadores aritméticos

Calcula el área de un triángulo con base 8 y altura 5.

### Solución:

```
base = 8
altura = 5
area = (base * altura) / 2
print("El área del triángulo es:", area)
```

### Explicación:

- Se usa `*` para multiplicar.
- Se usa `/` para dividir.
- La fórmula del área es  $(base * altura) / 2$ .

### Ejercicio 3: Operadores de comparación

Verifica si el número  $n = 18$  es mayor de edad (mayor o igual a 18).

### ✓ Solución:

```
n = 18
es_mayor = n >= 18
print("¿Es mayor de edad?", es_mayor)
```

### 💡 Explicación:

- Se usa >= para comparar si el valor cumple una condición.
- 

## 3. STRINGS

### Ejercicio 3: Manipulación de strings

Dado el string "python es genial", transforma:

- A mayúsculas
- A título (cada palabra en mayúscula)
- Cuenta cuántas veces aparece la letra "e"

### ✓ Solución:

```
texto = "python es genial"
print(texto.upper()) # MAYÚSCULAS
print(texto.title()) # Título
print(texto.count('e')) # Contar 'e'
```

### Ejercicio 4: Reemplazo y división

Dado el string "Hola mundo cruel", reemplaza "cruel" por "bonito" y separa las palabras en una lista.

### ✓ Solución:

```
texto = "Hola mundo cruel"
texto = texto.replace("cruel", "bonito")
palabras = texto.split()
print(texto)
print(palabras)
```

### 💡 Explicación:

- `replace()` cambia partes del texto.
  - `split()` divide el string por espacios (por defecto).
-

## 4. LISTAS

### Ejercicio 4: Listas y métodos

Crea una lista de 5 frutas, añade una más, ordena la lista y elimina la segunda.

#### Solución:

```
frutas = ["manzana", "naranja", "banana", "kiwi", "uva"]
frutas.append("sandía") # Añadir
frutas.sort() # Ordenar alfabéticamente
frutas.pop(1) # Eliminar segunda fruta (índice 1)
print(frutas)
```

### Ejercicio 5: Suma de elementos

Crea una lista de números y muestra la suma total.

```
numeros = [3, 7, 2, 9, 4]
```

#### Solución:

```
total = sum(numeros)
print("Suma total:", total)
```

#### Explicación:

- `sum()` recorre automáticamente la lista y suma todos los valores.

---

## 5. TUPLAS

### Ejercicio 5: Tuplas

Crea una tupla con 3 colores, accede al segundo elemento y verifica si "verde" está en la tupla.

#### Solución:

```
colores = ("rojo", "azul", "verde")
print(colores[1]) # Accede a "azul"
print("verde" in colores) # Verifica si existe
```

## Ejercicio 6: Iterar una tupla

Dada la tupla `dias = ("lunes", "martes", "miércoles")`, muestra todos los días con un bucle.

### Solución:

```
dias = ("lunes", "martes", "miércoles")
for dia in dias:
 print(dia)
```

### Explicación:

- Aunque las tuplas son inmutables, se pueden recorrer con bucles igual que listas.

---

## 6. SETS

### Ejercicio 6: Conjuntos

Dadas dos listas con elementos repetidos, crea sets y muestra la intersección.

```
a = [1, 2, 3, 4, 5, 5]
b = [4, 5, 6, 7, 5]
```

### Solución:

```
set_a = set(a)
set_b = set(b)
print(set_a & set_b) # Intersección
```

### Ejercicio 7: Diferencias y unión

Dado:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
```

Haz:

- Unión
- Diferencia

### Solución:

```
print("Unión:", set1 | set2)
print("Diferencia (set1 - set2):", set1 - set2)
```

### 💡 Explicación:

- | es la unión: todos los elementos únicos.
  - - es la diferencia: elementos de `set1` que no están en `set2`.
- 

## 7. 📖 DICCIONARIOS

### 📝 Ejercicio 7: Diccionario básico

Crea un diccionario con nombre, edad y ciudad. Luego:

- Muestra la edad
- Añade un campo "email"
- Cambia la ciudad

### ✅ Solución:

```
persona = {"nombre": "Lucía", "edad": 30, "ciudad": "Madrid"}
print(persona["edad"]) # Mostrar edad
persona["email"] = "lucia@mail.com"
persona["ciudad"] = "Barcelona"
print(persona)
```

### 📝 Ejercicio 8: Iterar claves y valores

Dado:

```
persona = {"nombre": "Ana", "edad": 22}
```

Recorre el diccionario y muestra cada par clave:valor.

### ✅ Solución:

```
for clave, valor in persona.items():
 print(f"{clave} -> {valor}")
```

### 💡 Explicación:

- `.items()` devuelve pares (clave, valor) para iterar fácilmente.
-

## 8. CONDICIONALES

### Ejercicio 1: Número positivo, negativo o cero

Pide al usuario un número y di si es positivo, negativo o cero.

#### Solución:

```
num = int(input("Introduce un número: "))

if num > 0:
 print("Es positivo")
elif num < 0:
 print("Es negativo")
else:
 print("Es cero")
```

#### Explicación:

- `if`, `elif`, y `else` permiten evaluar múltiples condiciones de forma ordenada.

### Ejercicio 2: Año bisiesto

Verifica si un año es bisiesto:

- Es divisible por 4 y no por 100, o divisible por 400.

#### Solución:

```
anio = int(input("Introduce un año: "))

if (anio % 4 == 0 and anio % 100 != 0) or (anio % 400 == 0):
 print("Es bisiesto")
else:
 print("No es bisiesto")
```

#### Explicación:

- Se usan operadores lógicos `and`, `or` para verificar múltiples condiciones.

---

## 9. BUCLES

### Ejercicio 1: Imprimir números del 1 al 10

Usa un bucle `for` para mostrar los números del 1 al 10.

### ✓ Solución:

```
for i in range(1, 11):
 print(i)
```

### 💡 Explicación:

- `range(1, 11)` genera números del 1 al 10 (el 11 no se incluye).

### 📝 Ejercicio 2: Bucle con `while`

Suma los números ingresados por el usuario hasta que escriba "fin".

### ✓ Solución:

```
total = 0
entrada = input("Introduce un número o 'fin' para salir: ")

while entrada != "fin":
 total += int(entrada)
 entrada = input("Introduce otro número o 'fin': ")

print("Suma total:", total)
```

### 💡 Explicación:

- `while` repite mientras la condición sea verdadera.
- Se convierte cada entrada a `int` para sumarla.

---

## 10. 🔗 FUNCIONES

### 📝 Ejercicio 1: Función que devuelve el cuadrado

Crea una función `cuadrado(num)` que devuelva el cuadrado de un número.

### ✓ Solución:

```
def cuadrado(num):
 return num ** 2

print(cuadrado(5)) # 25
```

### 💡 Explicación:

- Las funciones se definen con `def`.
- `return` devuelve un valor.



## Ejercicio 2: Función con parámetros y condicionales

Crea una función `es_par(num)` que devuelva `True` si el número es par, y `False` si es impar.

### Solución:

```
def es_par(num):
 return num % 2 == 0

print(es_par(4)) # True
print(es_par(7)) # False
```

### Explicación:

- Se usa el operador `%` para saber si el número tiene resto 0 al dividir entre 2.

---

## 11. □ CLASES

### Ejercicio 1: Clase `Persona`

Crea una clase con atributos `nombre` y `edad`. Incluye un método que salude.

### Solución:

```
class Persona:
 def __init__(self, nombre, edad):
 self.nombre = nombre
 self.edad = edad

 def saludar(self):
 print(f"Hola, me llamo {self.nombre} y tengo {self.edad} años.")

p1 = Persona("Luis", 30)
p1.saludar()
```

### Explicación:

- `__init__` es el constructor.
- `self` representa la instancia actual.

### Ejercicio 2: Clase `Rectángulo`

Crea una clase con atributos `base` y `altura`, y un método que calcule el área.

### Solución:

```
class Rectangulo:
 def __init__(self, base, altura):
```

```
 self.base = base
 self.altura = altura

 def area(self):
 return self.base * self.altura

r = Rectangulo(5, 3)
print("Área:", r.area())
```

### Explicación:

- Se define un método `area()` que usa los atributos de la clase.

---

## 12. EXCEPCIONES

### Ejercicio 1: Capturar división por cero

Pide dos números y divide, capturando errores si el divisor es cero.

### Solución:

```
try:
 a = int(input("Número 1: "))
 b = int(input("Número 2: "))
 print("Resultado:", a / b)
except ZeroDivisionError:
 print("Error: No se puede dividir por cero.")
```

### Explicación:

- `try` intenta ejecutar el bloque.
- `except` captura errores y evita que el programa se detenga.

### Ejercicio 2: Capturar múltiples errores

Modifica el código anterior para capturar errores si se ingresa texto en lugar de números.

### Solución:

```
try:
 a = int(input("Número 1: "))
 b = int(input("Número 2: "))
 print("Resultado:", a / b)
except ZeroDivisionError:
 print("Error: División por cero.")
except ValueError:
 print("Error: Debes ingresar un número entero.")
```

### 💡 Explicación:

- `ValueError` ocurre si `int()` falla por entrada no numérica.
- 

## 13. 📁 MÓDULOS

### 📝 Ejercicio 1: Usar el módulo `math`

Importa el módulo `math` y muestra la raíz cuadrada de 25.

### ✅ Solución:

```
import math

resultado = math.sqrt(25)
print("Raíz cuadrada de 25:", resultado)
```

### 💡 Explicación:

- `import math` te permite usar funciones matemáticas como `sqrt()`.
- Se accede con `math.función()`.

### 📝 Ejercicio 2: Crear y usar un módulo propio

Crea un archivo llamado `operaciones.py` con esta función:

```
def suma(a, b):
 return a + b
```

Y úsalo desde otro archivo:

```
import operaciones

print(operaciones.suma(4, 5))
```

### 💡 Explicación:

- Se puede crear un archivo `.py` con funciones reutilizables.
  - Luego se importa como un módulo.
-

## 14. 📅 DATES (Fechas)

### 📝 Ejercicio 1: Mostrar fecha actual

Usa el módulo `datetime` para mostrar la fecha y hora actual.

#### ✅ Solución:

```
from datetime import datetime

ahora = datetime.now()
print("Fecha y hora actual:", ahora)
```

#### 💡 Explicación:

- `datetime.now()` devuelve la fecha y hora actual.

### 📝 Ejercicio 2: Crear una fecha específica

Crea un objeto fecha para el 4 de julio de 2025 y muestra su día de la semana.

#### ✅ Solución:

```
import datetime

fecha = datetime.date(2025, 7, 4)
print("Día de la semana:", fecha.strftime("%A"))
```

#### 💡 Explicación:

- `%A` devuelve el nombre completo del día.

---

## 15. □ COMPRESIÓN DE LISTAS

### 📝 Ejercicio 1: Crear una lista de cuadrados

Usa comprensión de listas para obtener los cuadrados del 1 al 5.

#### ✅ Solución:

```
cuadrados = [x**2 for x in range(1, 6)]
print(cuadrados)
```

#### 💡 Explicación:

- La comprensión de listas genera una lista de forma compacta y legible.

## Ejercicio 2: Filtrar números pares

Crea una lista de los números pares entre 1 y 10.

### Solución:

```
pares = [x for x in range(1, 11) if x % 2 == 0]
print(pares)
```

### Explicación:

- Puedes añadir condiciones dentro de la comprensión (if).
- 

## 16. LAMBDA

### Ejercicio 1: Lambda para sumar

Crea una función lambda que sume dos números.

### Solución:

```
suma = lambda x, y: x + y
print(suma(3, 4)) # 7
```

### Explicación:

- `lambda` crea funciones anónimas. Útil para funciones simples.

### Ejercicio 2: Ordenar una lista de tuplas

Ordena una lista de tuplas por el segundo valor usando `lambda`.

```
pares = [(1, 3), (2, 2), (3, 1)]
```

### Solución:

```
ordenado = sorted(pares, key=lambda x: x[1])
print(ordenado)
```

### Explicación:

- `key=lambda x: x[1]` indica que ordene por el segundo elemento de cada tupla.
-

## 17. □ FUNCIONES DE ORDEN SUPERIOR

### Ejercicio 1: Usar `map` para duplicar

Duplica todos los valores de la lista `[1, 2, 3]` usando `map`.

#### Solución:

```
numeros = [1, 2, 3]
resultado = list(map(lambda x: x * 2, numeros))
print(resultado)
```

#### Explicación:

- `map(función, iterable)` aplica la función a cada elemento.

### Ejercicio 2: Usar `filter` para filtrar impares

Filtra los números impares en `[1, 2, 3, 4, 5]`.

#### Solución:

```
numeros = [1, 2, 3, 4, 5]
impares = list(filter(lambda x: x % 2 != 0, numeros))
print(impares)
```

#### Explicación:

- `filter()` selecciona elementos que cumplen una condición.

---

## 18. ✕ TIPOS DE ERROR

### Ejercicio 1: Manejar `TypeError`

Corrige este error y captura la excepción:

```
print("Número: " + 5)
```

#### Solución:

```
try:
 print("Número: " + str(5))
except TypeError:
 print("Error de tipo")
```

### 💡 Explicación:

- No se puede concatenar string con entero directamente. `str(5)` lo convierte a texto.

### 📝 Ejercicio 2: Manejar `IndexError`

Evita errores al acceder a un índice inexistente.

```
lista = [10, 20, 30]
print(lista[5])
```

### ✅ Solución:

```
lista = [10, 20, 30]

try:
 print(lista[5])
except IndexError:
 print("Índice fuera de rango")
```

### 💡 Explicación:

- `IndexError` ocurre cuando se accede a una posición que no existe.

---

## 19. 📁 MANEJO DE ARCHIVOS

### 📝 Ejercicio 1: Crear y escribir en un archivo

Escribe el texto "Hola, mundo!" en un archivo llamado `saludo.txt`.

### ✅ Solución:

```
with open("saludo.txt", "w", encoding="utf-8") as archivo:
 archivo.write("Hola, mundo!")
```

### 💡 Explicación:

- "w" significa **modo escritura**.
- `with` asegura que el archivo se cierre automáticamente.
- `archivo.write()` escribe texto.

### 📝 Ejercicio 2: Leer un archivo línea por línea

Lee e imprime las líneas del archivo `saludo.txt`.

### ✓ Solución:

```
with open("saludo.txt", "r", encoding="utf-8") as archivo:
 for linea in archivo:
 print(linea.strip())
```

### 💡 Explicación:

- "r" es **modo lectura**.
- `.strip()` elimina espacios y saltos de línea.

---

## 20. 🔍 EXPRESIONES REGULARES

### 📝 Ejercicio 1: Validar una dirección de correo

Verifica si el texto "usuario@correo.com" es un correo válido.

### ✓ Solución:

```
import re

correo = "usuario@correo.com"
patron = r"^[\\w\\.-]+@[\\w\\.-]+\\.\\w+$"

if re.match(patron, correo):
 print("Correo válido")
else:
 print("Correo inválido")
```

### 💡 Explicación:

- `re.match()` compara el patrón desde el inicio del texto.
- `\\w` equivale a letras, números o guiones bajos.
- `^` y `$` marcan inicio y final del texto.

### 📝 Ejercicio 2: Buscar números en un texto

Encuentra todos los números en "Tengo 2 perros y 3 gatos".

### ✓ Solución:

```
import re

texto = "Tengo 2 perros y 3 gatos"
numeros = re.findall(r"\d+", texto)
print(numeros) # ['2', '3']
```



### Explicación:

- `\d+` busca uno o más dígitos seguidos.
- `findall()` devuelve una lista de todas las coincidencias.

---

## 21. PAQUETES DE PYTHON

### Ejercicio 1: Crear un paquete simple

1. Estructura del paquete:

```
mi_paquete/
├── __init__.py
└── operaciones.py
```

2. Contenido de `operaciones.py`:

```
def sumar(a, b):
 return a + b
```

3. Uso desde otro archivo:

```
from mi_paquete import operaciones

print(operaciones.sumar(2, 3)) # 5
```

### Explicación:

- Un paquete es una carpeta con `__init__.py`.
- Puedes importar módulos del paquete desde otros scripts.

### Ejercicio 2: Instalar y usar un paquete externo

Instala `requests` y úsalo para obtener una página web:

```
pip install requests
```

Código:

```
import requests

respuesta = requests.get("https://www.example.com")
print("Código de estado:", respuesta.status_code)
```

### Explicación:

- `requests.get()` hace una petición HTTP GET.
- `.status_code` muestra el código de respuesta (200, 404, etc.).