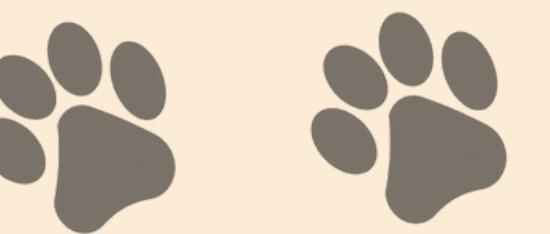
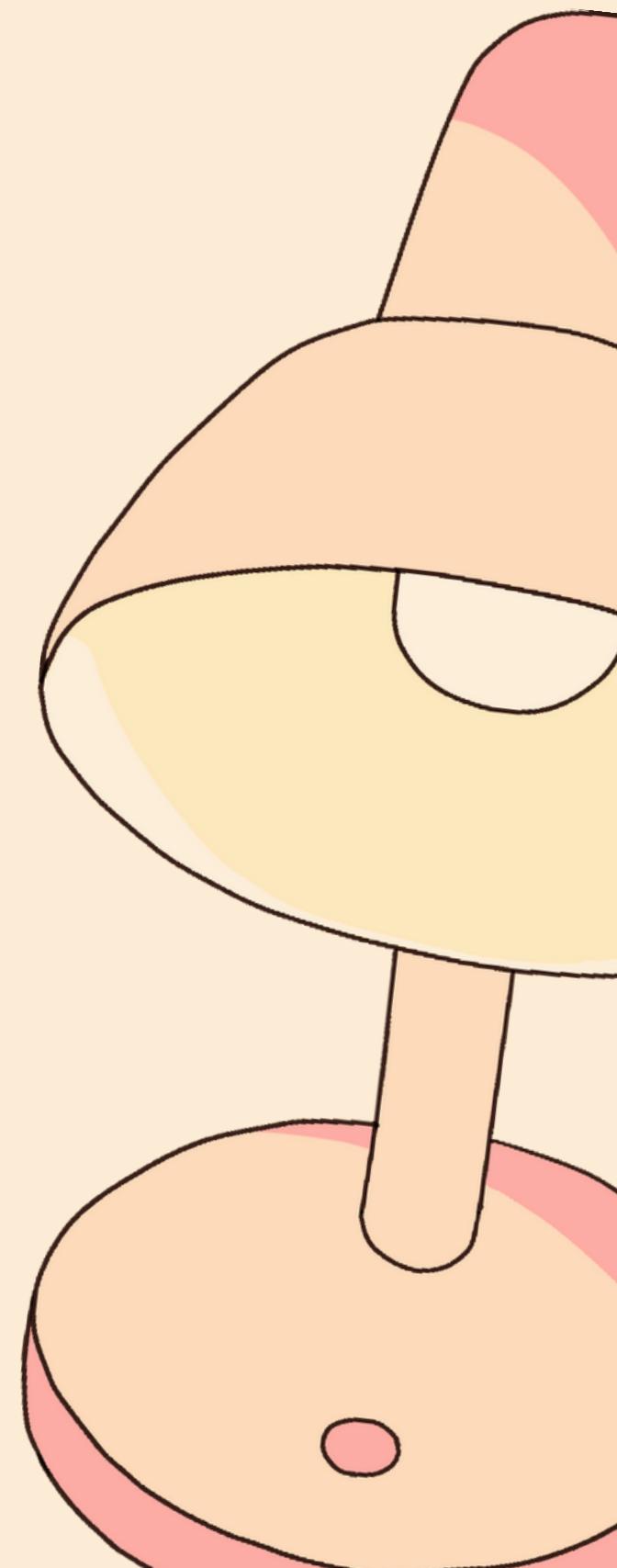


PYTHON



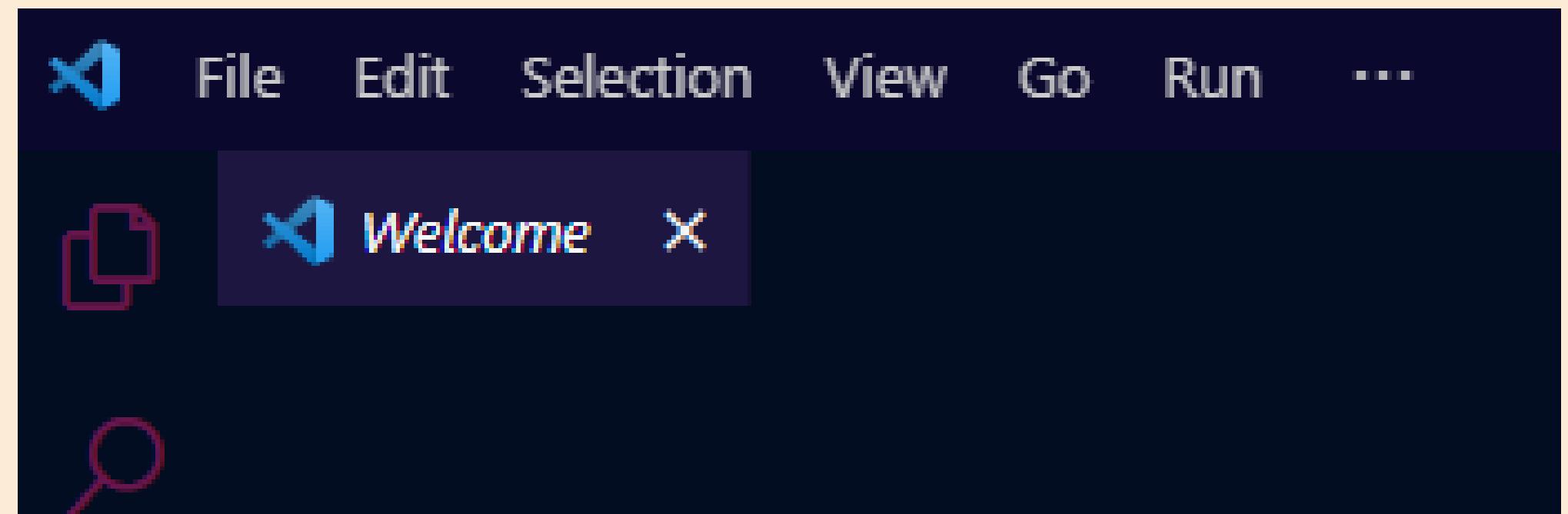
- 00 - Vs code
- 01 - Variables
- 02 - Operadores
- 03 - Strings
- 04 - Lista
- 05 - Tuplas
- 06 - Sets
- 07 - Diccionarios
- 08 - Condicionales
- 09 -
Bucles/Loops/Ciclos
- 10 - Funciones
- 11 - Clases
- 12 - Excepciones
- 13 - Módulos



VS Code

Visual Estudio Code es un editor de código, aquí es donde nosotros vamos a aprender python(tu puedes usar el editor que mas como te sientas).

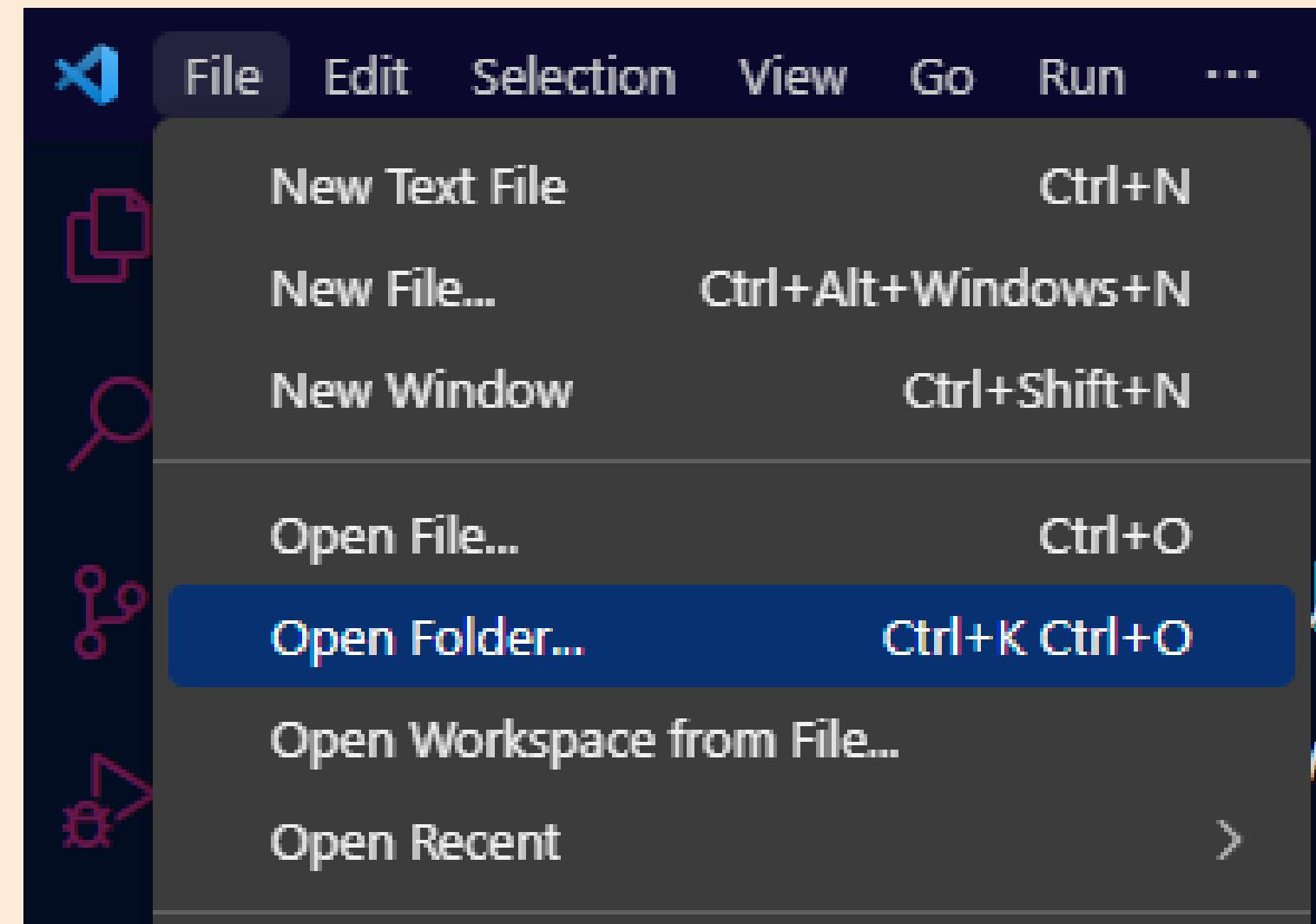
Vamos a ver como crear un archivo .py para poder empezar a aprender



Lo importante una vez instalado es ir a file

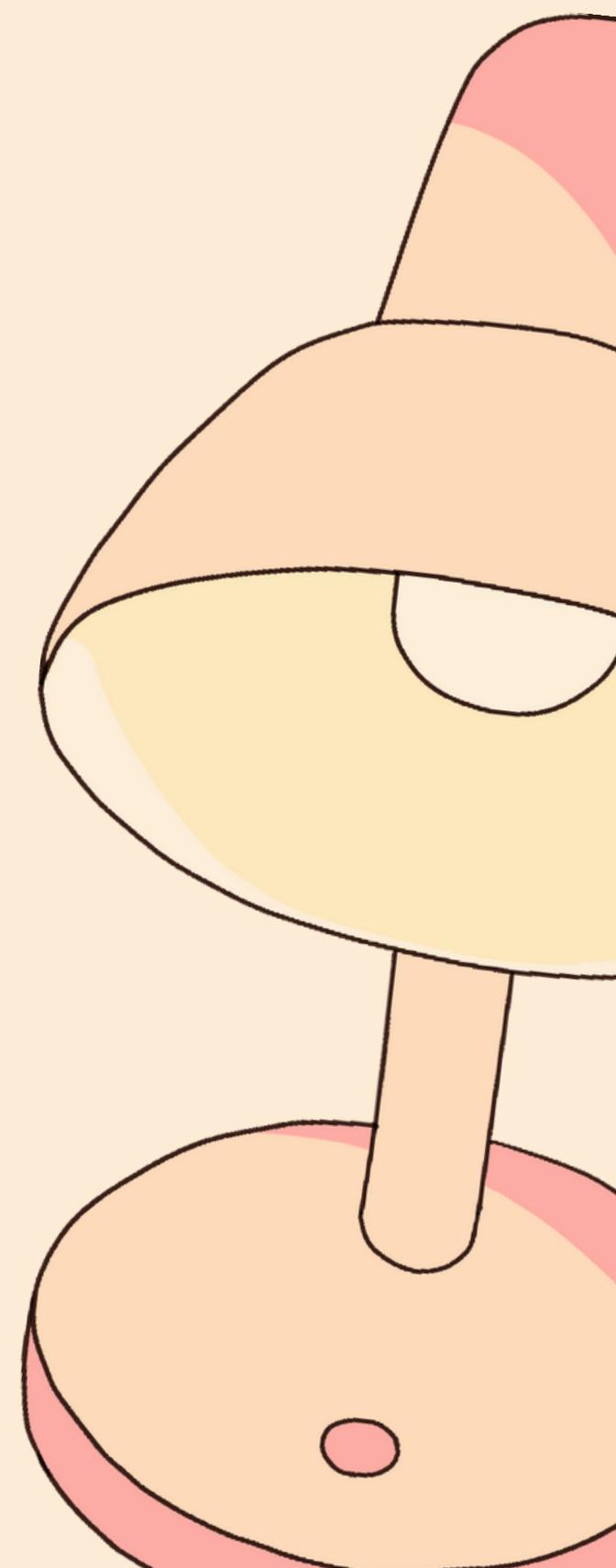
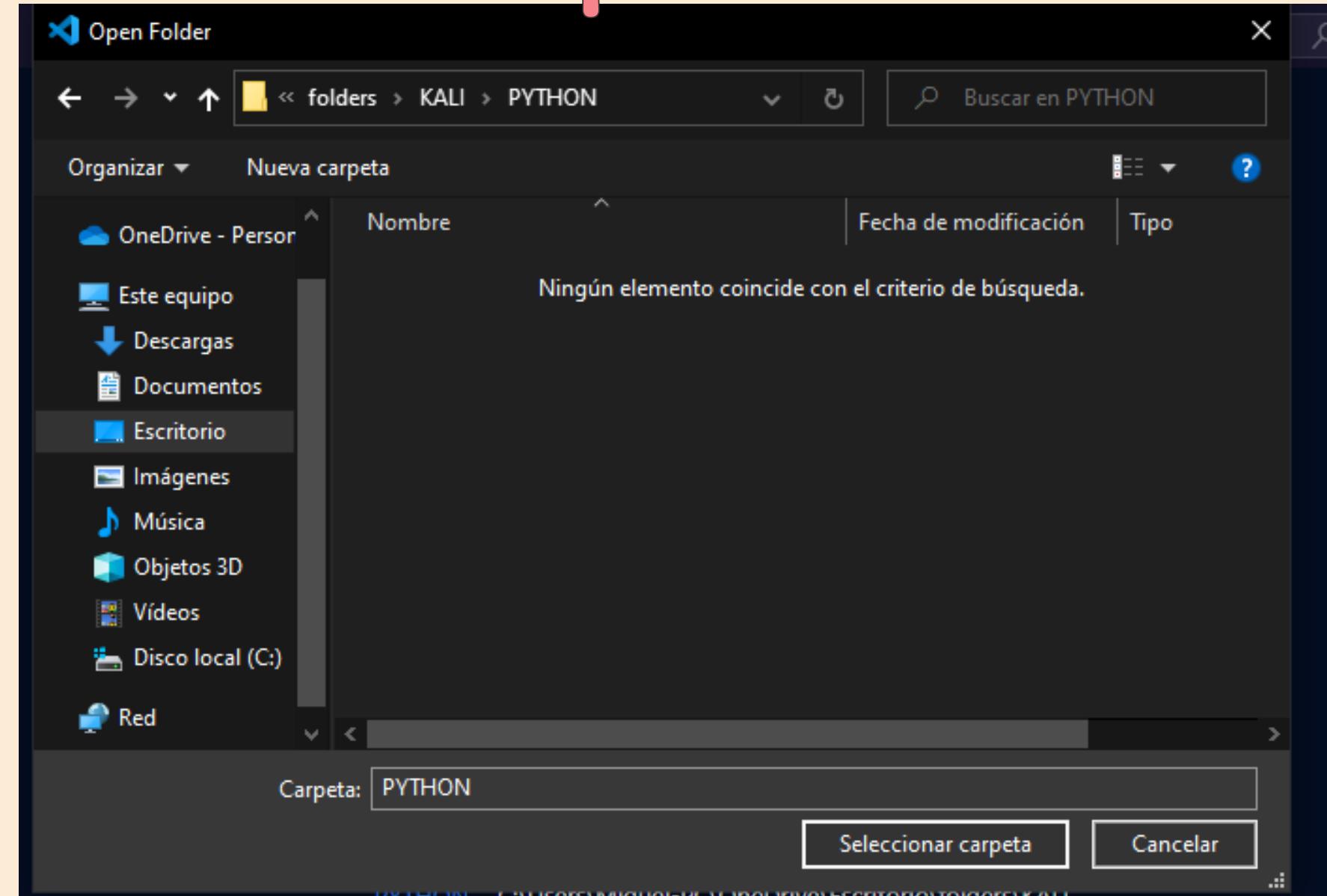
VS Code

necesitamos abrir una carpeta a si que le damos a open folder...



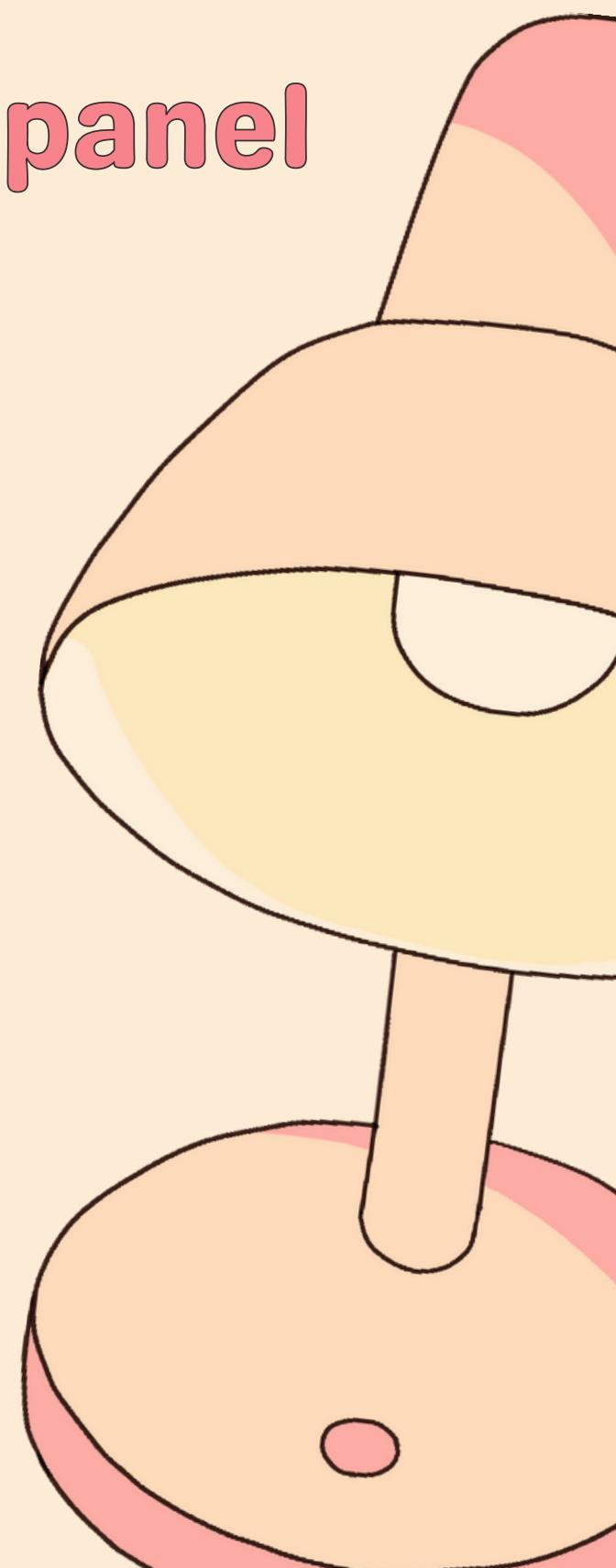
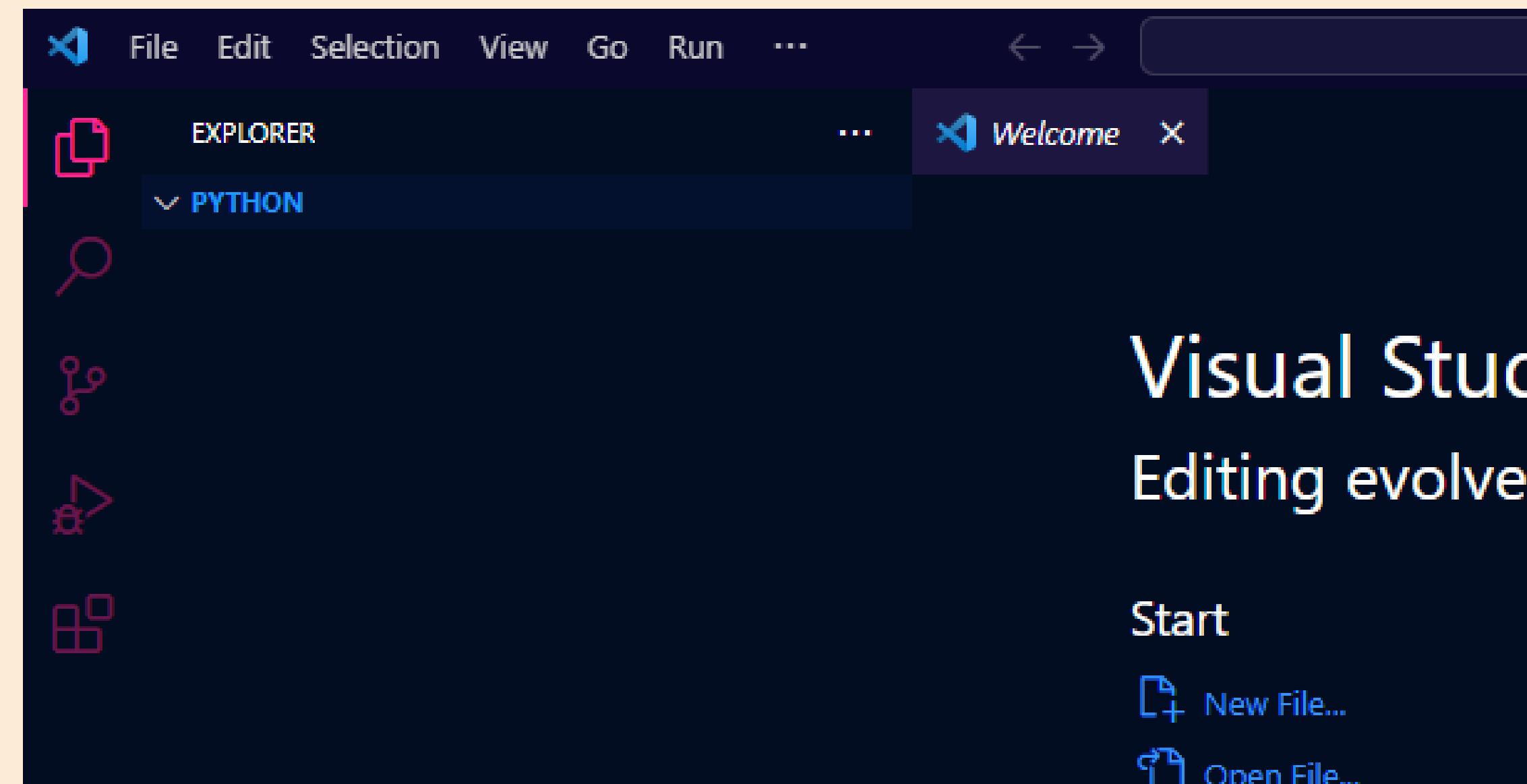
VS Code

elegimos una carpeta , esta carpeta sera donde almacenara los archivos.py que creamos.
en mi caso sera en la carpeta PYTHON



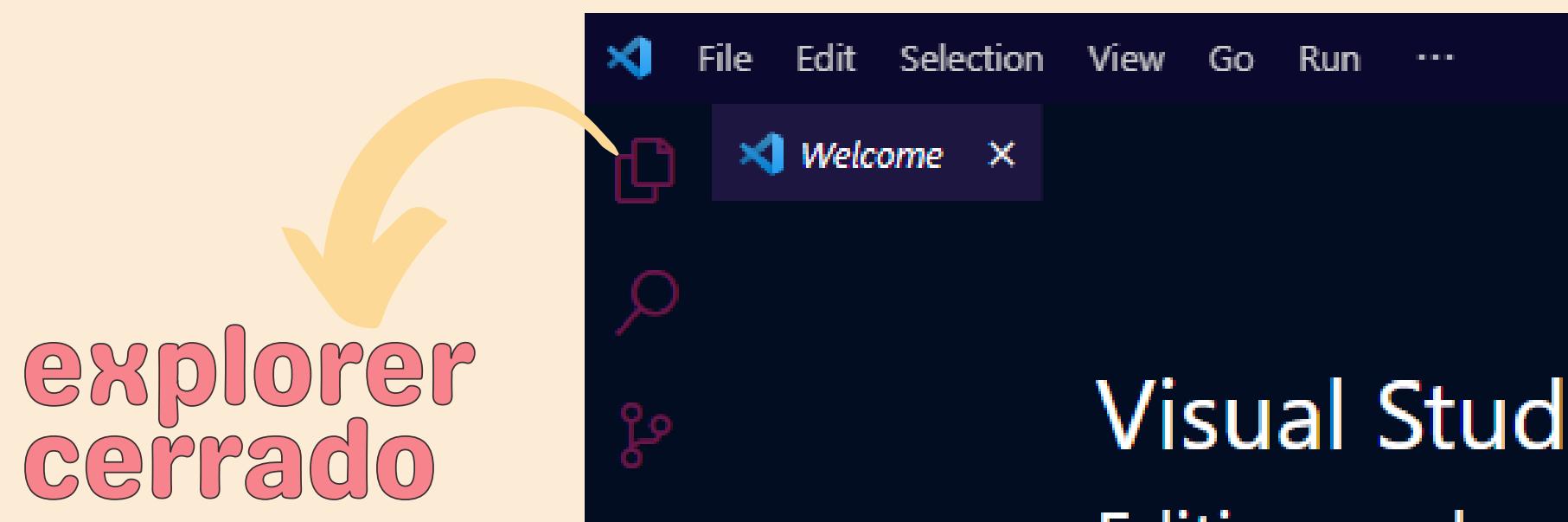
VS Code

Cuando le demos en seleccionar carpeta, nos aparecera el nombre de nuestra carpeta en ese panel EXPLORER

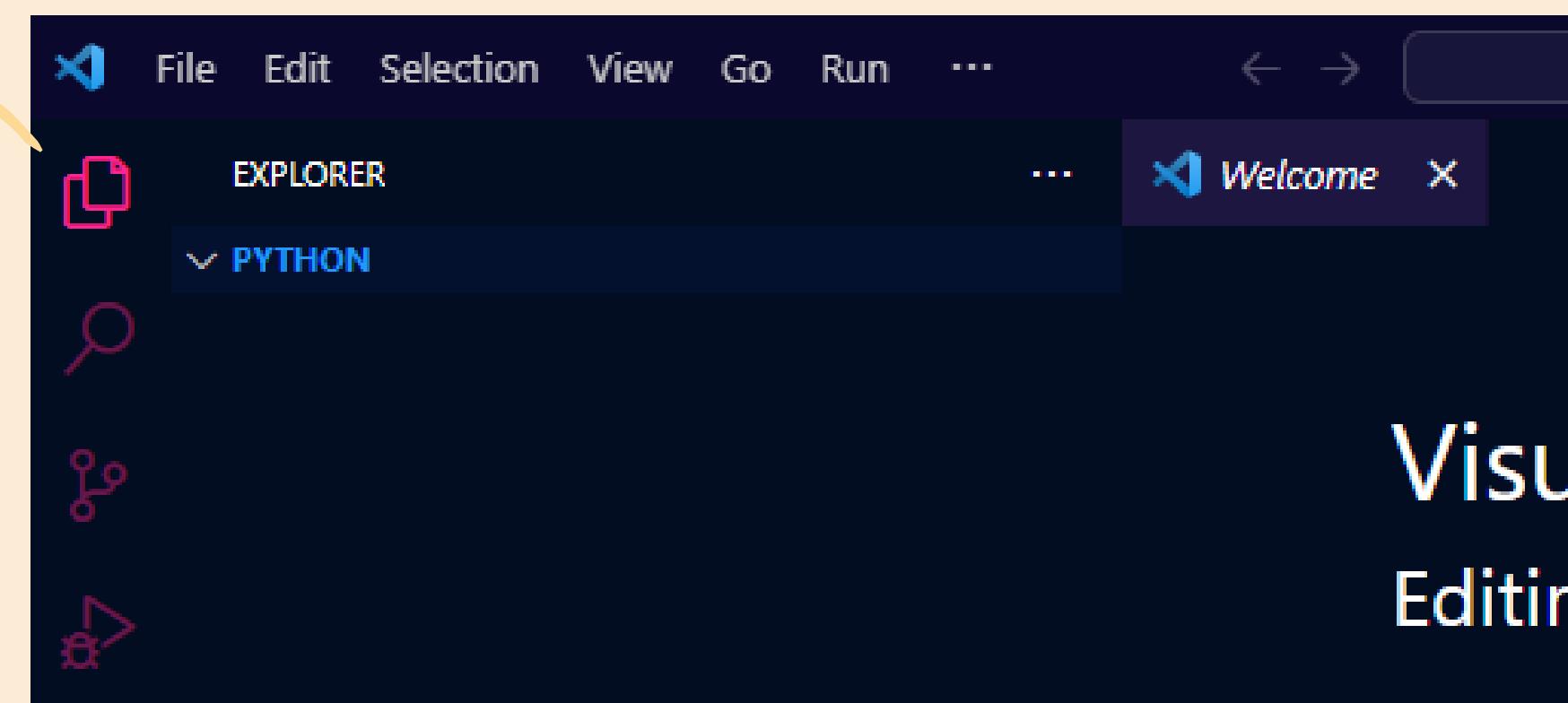


VS Code

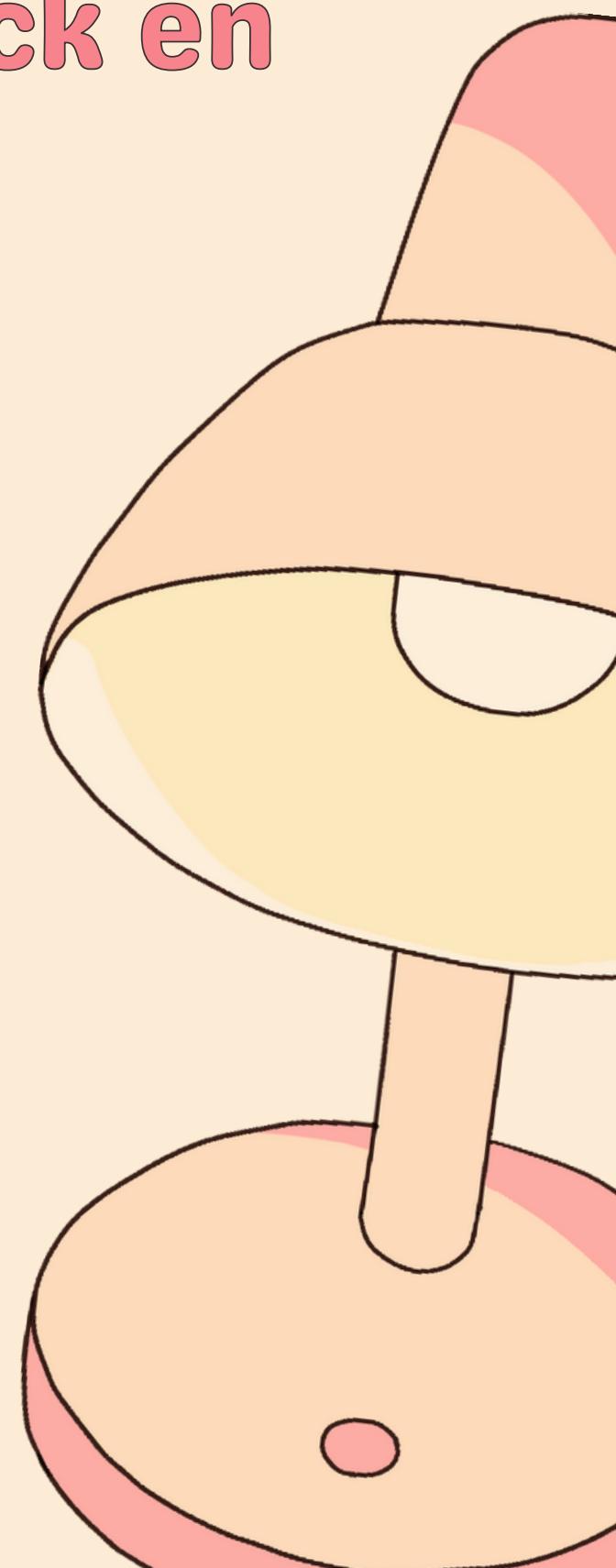
Este panel se puede abriro o cerrar haciendo click en este simbolo.



explorer
cerrado

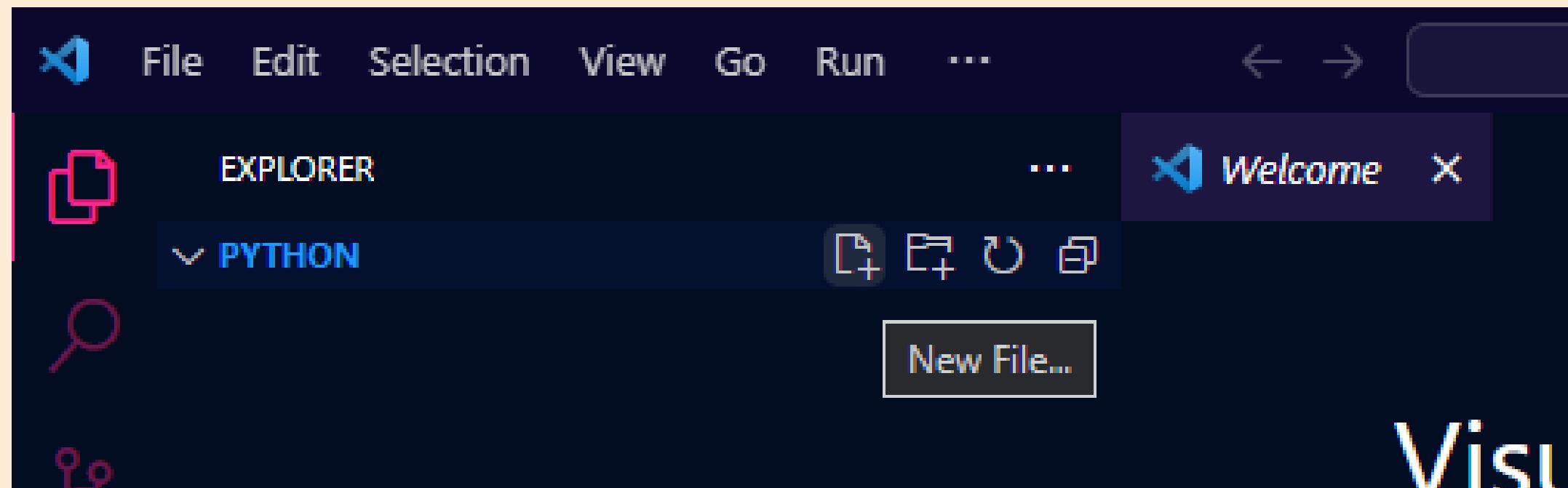
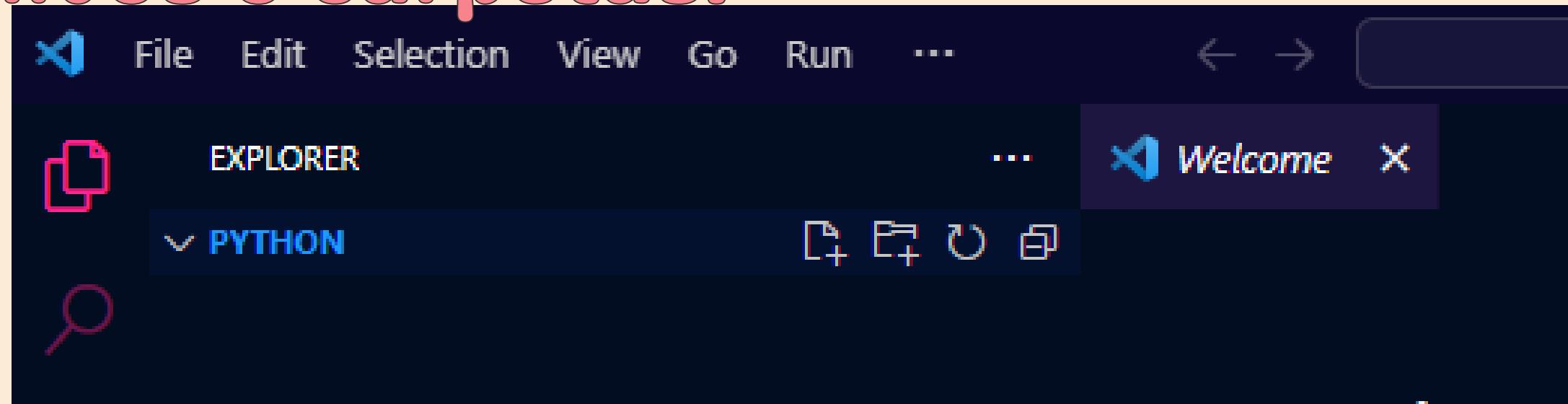


explorer
abierto



VS Code

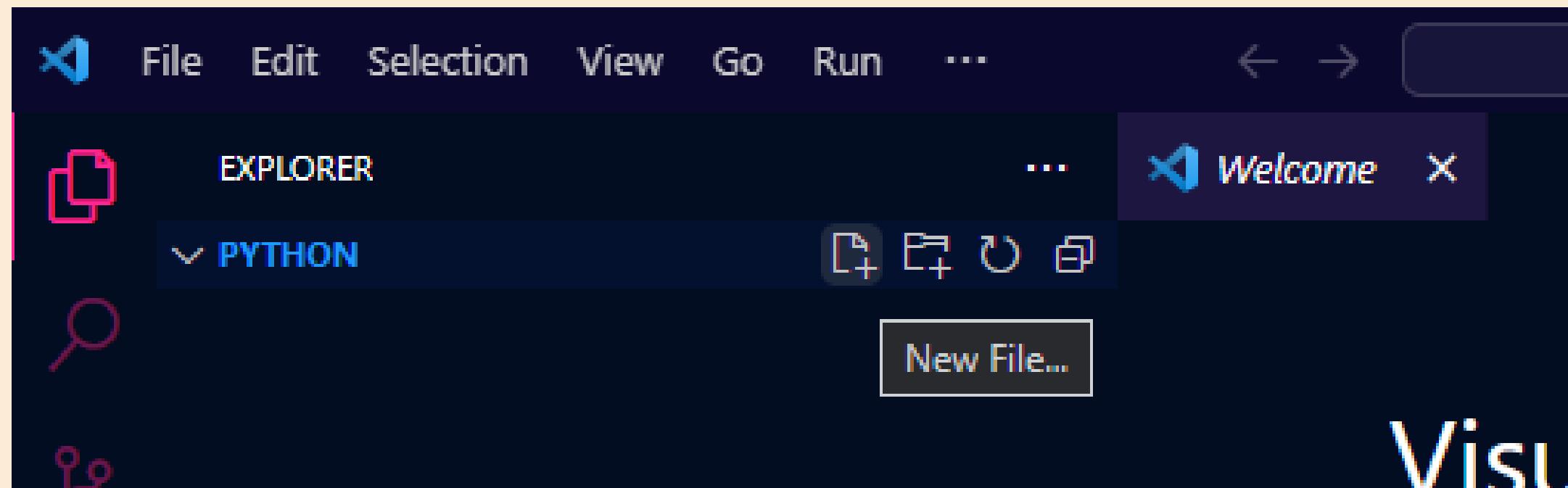
Si acercamos el mouse al panel de EXPLORER podemos ver que aparecen opciones donde podremos crear archivos o carpetas.



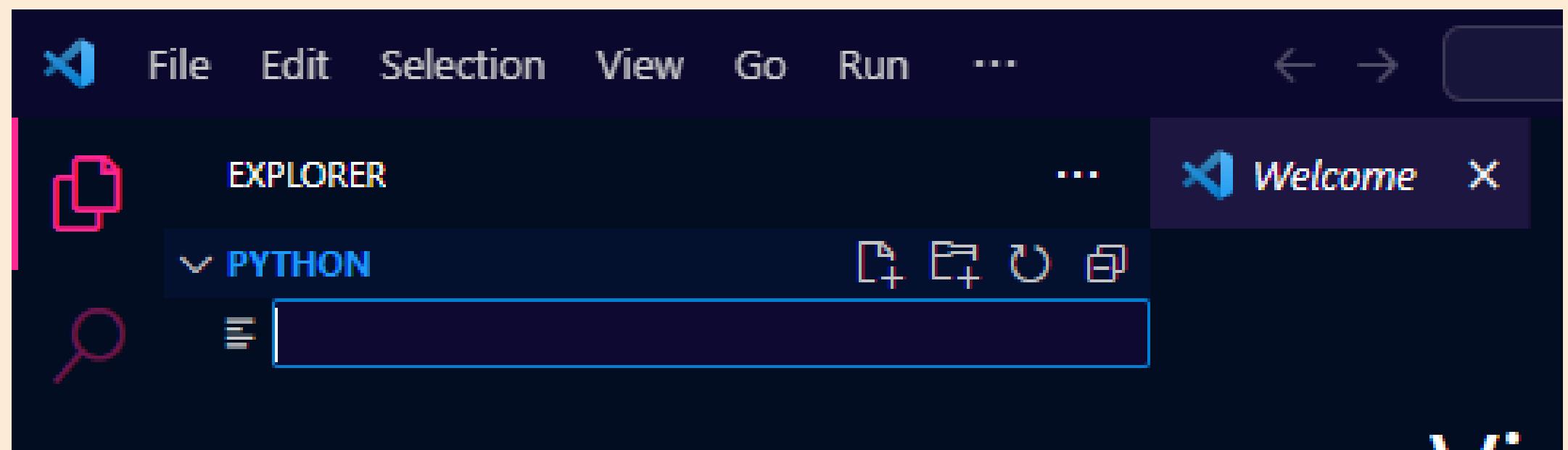
Visu

VS Code

Haremos click en new file , para crear un archivo

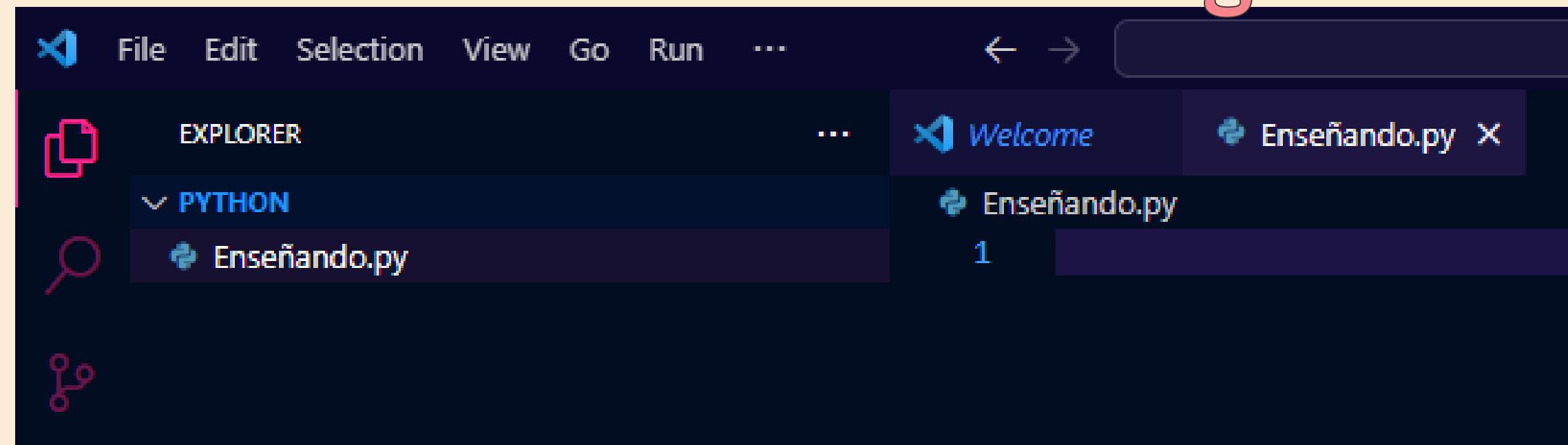


Tendremos que ponerle el nombre que queramos y al final .py



VS Code

poniendo al final del nombre.py estamos indicando al archivo que contendrá código python (si no hicieramos esto a la hora de escribir código no nos lo reconocería)



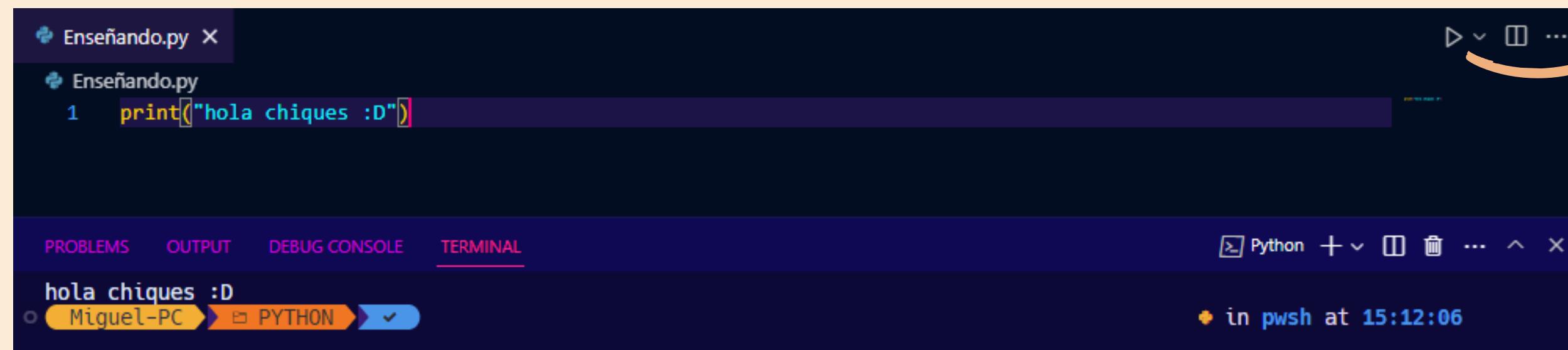
Borremos la ventana de Welcome para que no moleste



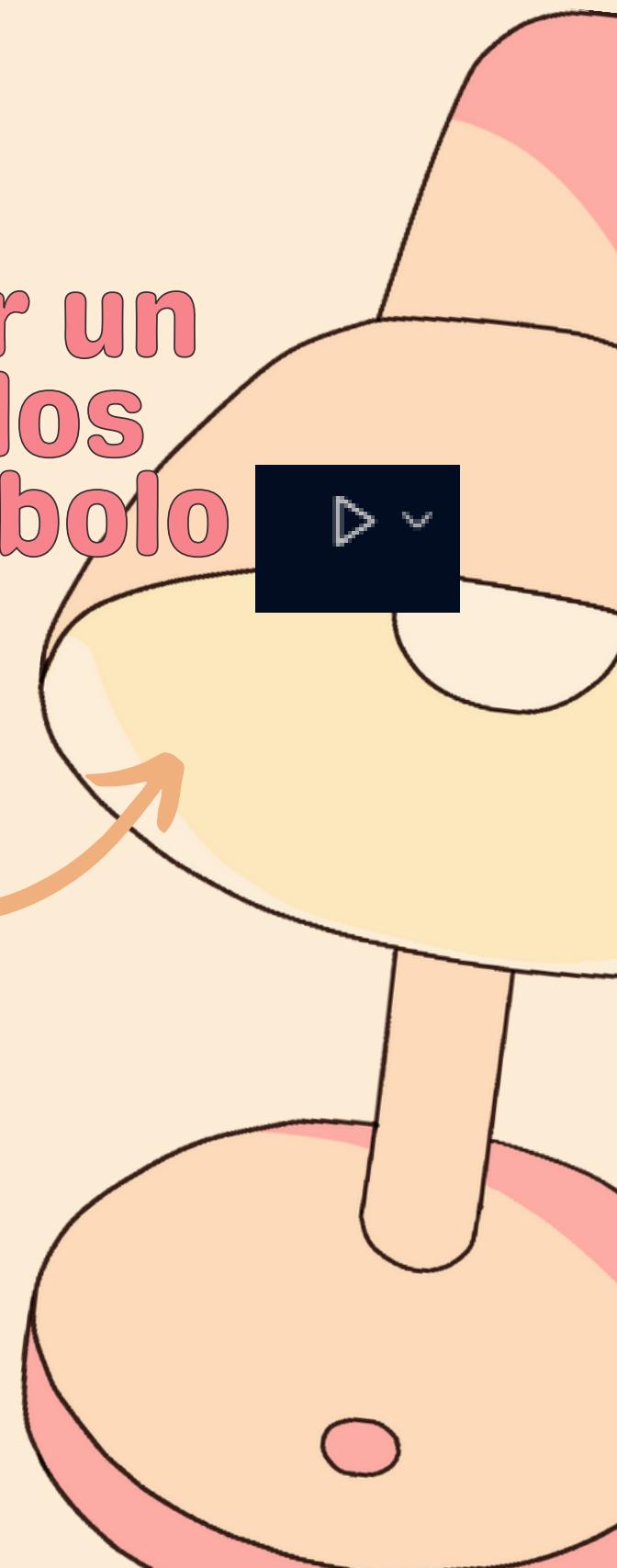
VS Code

una vez ya todo listo podemos escribir codigo
tranquilamente y ejecutarlos.
por ejemplo:

escribiremos print("hola chiques") para mostrar un
mensaje , ahora para ejecutar esto para que no los
muestre en la terminal hay que darle a este simbolo



A screenshot of the Visual Studio Code (VS Code) interface. The top bar shows two tabs: 'Enseñando.py' and another 'Enseñando.py' tab. The main editor area contains the Python code: 'print("hola chiques :D")'. Below the editor is a dark blue navigation bar with tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The 'TERMINAL' tab is currently selected, showing the output 'hola chiques :D'. At the bottom of the screen, there's a status bar with 'Python' and 'in pwsh at 15:12:06'.



variables

En Python, puedes crear una variable simplemente asignando un valor a un nombre. No necesitas declarar explícitamente el tipo de variable

EJEMPLO:

variable

```
python
mensaje = "Hola mundo"
print(mensaje)
```

asignacion

valor

variables

veamos como esta estructurado ese codigo por que aun que no lo parezca tiene muchas cosas.

mensaje

Estamos escribiendo una variable la cual la llamamos como nosotros queremos, en este caso mensaje esta bien para el ejemplo

```
python
mensaje = "Hola mundo"
print(mensaje)
```

=
Con este simbolo estamos diciendo que le asignamos un dato a nuestra variable mensaje

"hola mundo"
El valor asignado es una cadena de texto por que empieza y termina con "
esto significa que es un dato tipo string(texto)

variables



```
python
mensaje = "Hola mundo"
print(mensaje)
```

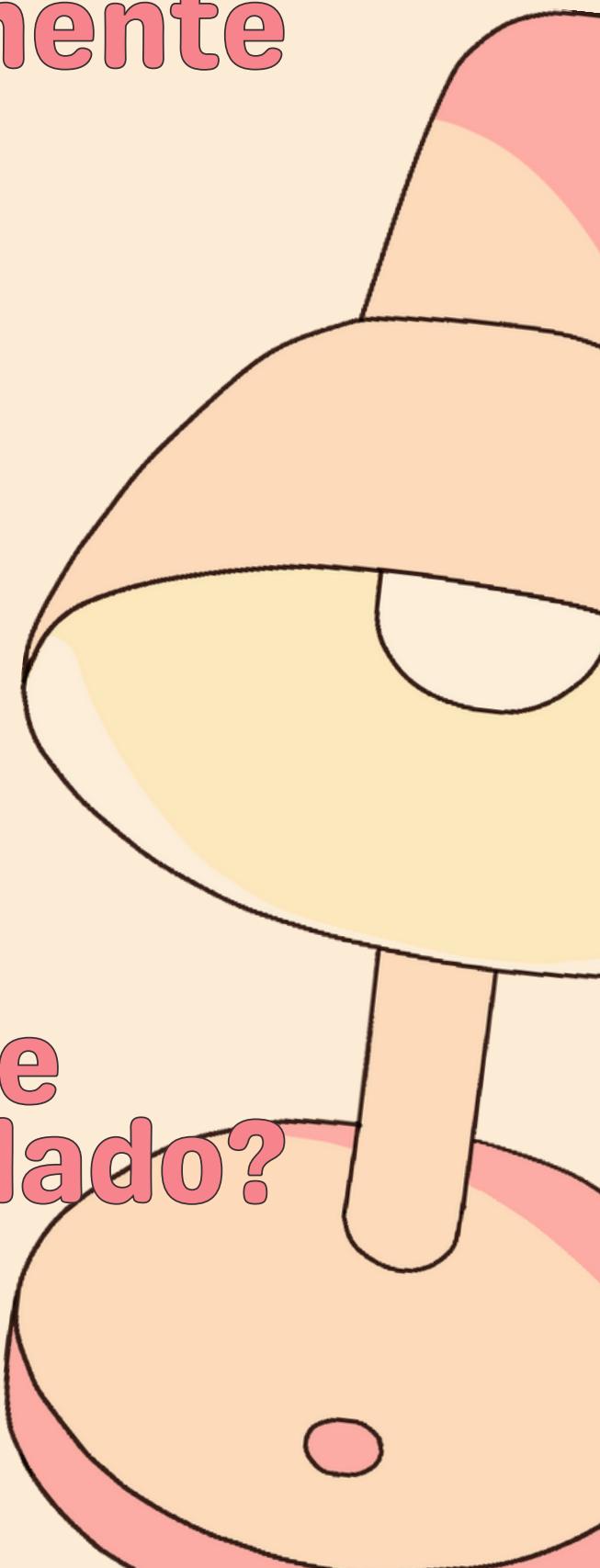
¿Que conclusion llegamos con esto?-
que creamos algo con un nombre y ese algo le
decimos que con este simbolo = quiero que contenga
ese dato que yo quiera.

variables

¿Como puedo ver que mi variable efectivamente tiene ese dato vinculado?

print()

Con print podemos mostrar nuestras variables o mensajes que nosotros queramos y esto se pone dentro de sus ()



```
python
mensaje = "Hola mundo"
print(mensaje)
```

¿Como puedo ver que mi variable efectivamente tiene ese dato vinculado?

variables

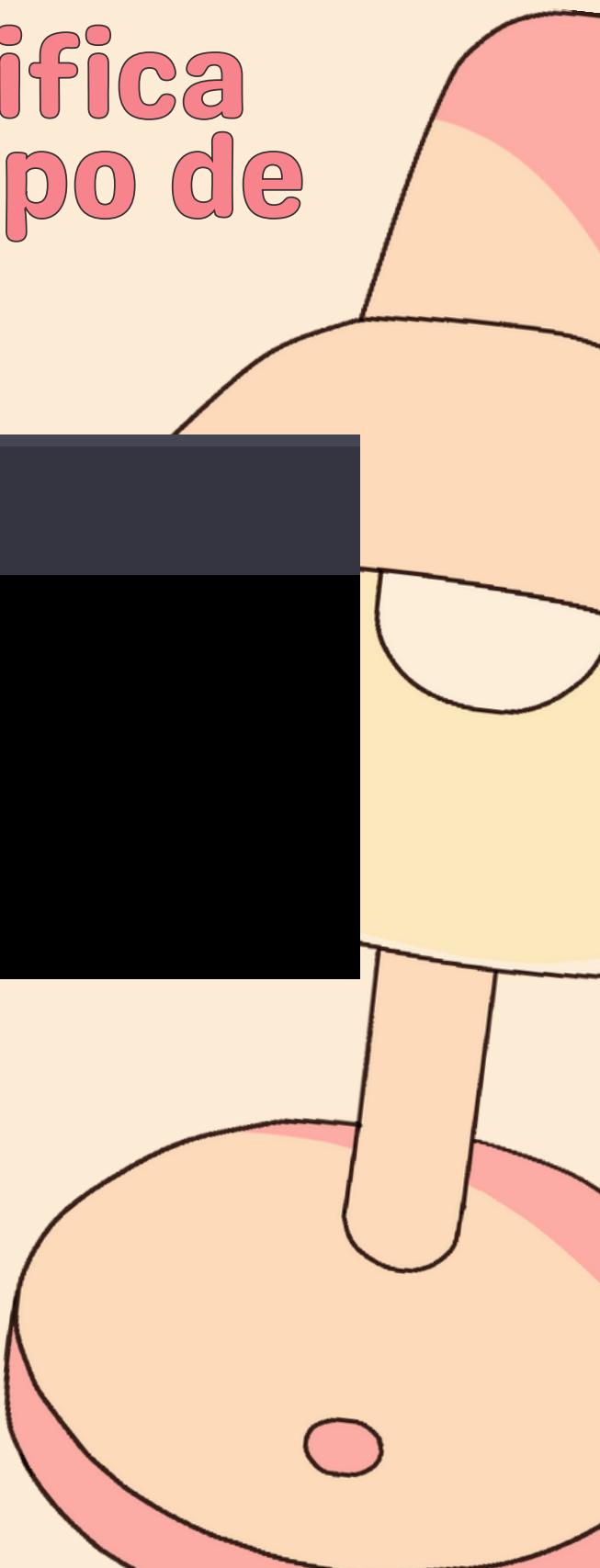
Como python es un lenguaje de alto nivel(significa que No necesitas declarar explícitamente el tipo de variable). por ejemplo:

ATENCION

No hace falta escribir por ejempl edad para que contenga un numero , el nombre puede ser cualquiera , solo le ponemos el nombre descriptivo para recordarlo con facilidad.
inicial = 25 el nombre da igual esto no influye en nuestro valor, el dato seguira siendo 25.



```
python
edad = 25
altura = 1.75
inicial = 'J'
```



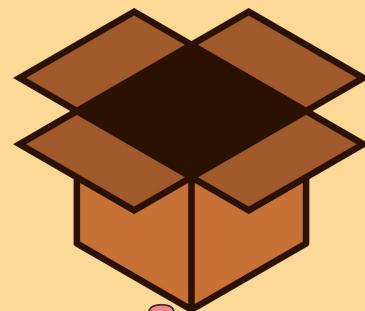
variables

Como python es un lenguaje de alto nivel(significa que No necesitas declarar explícitamente el tipo de variable). por ejemplo:

ATENCION

una variable para crearla necesitamos elegir un nombre luego con = podemos hacer que la variable se vincule(contenga un valor).

ejemplo grafico:



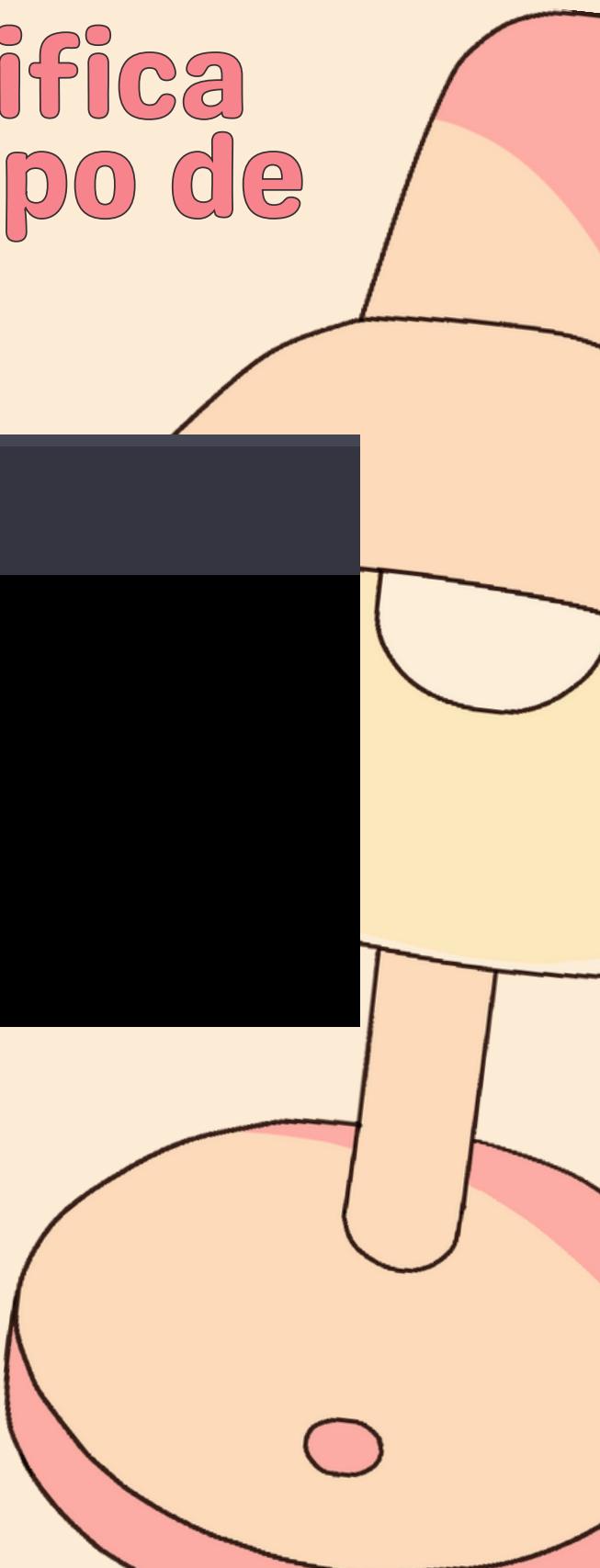
variable



valor



```
python
edad = 25
altura = 1.75
inicial = 'J'
```



variables

Como python es un lenguaje de alto nivel(significa que No necesitas declarar explícitamente el tipo de variable). por ejemplo:

ATENCION

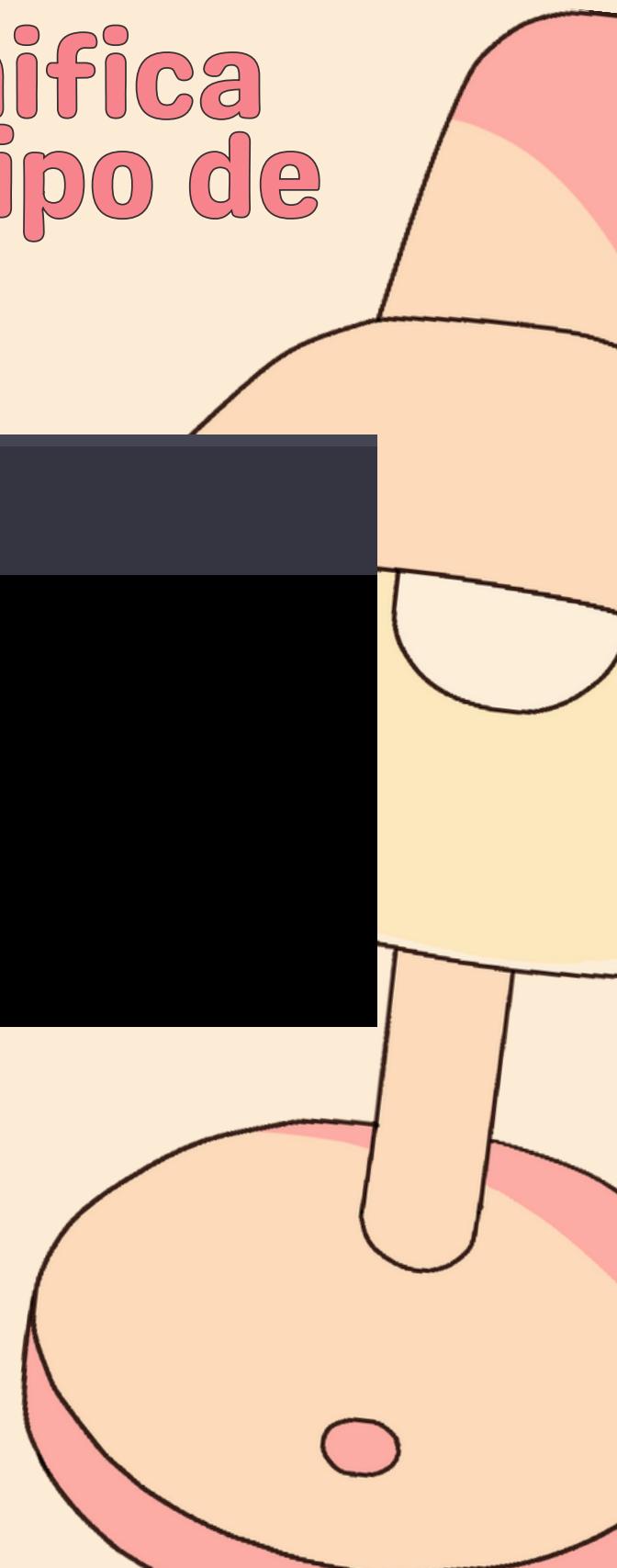


variable = **valor**

Ahora podemos ver como hay un valor dentro de la variable



```
python
edad = 25
altura = 1.75
inicial = 'J'
```



variables

podemos ver con el ejemplo grafico anterior como un lenguaje de alto nivel como python no hace falta indicar que tipo de dato es tu variable , lo detecta solo.

Pero encambio en un lenguaje de bajo nivel como C tienes que indicar que tipo dedato es la variable.

en un lenjuage de bajo nivel es requisito obligatorio poner al principio que tipo de dato almacenara tu variable, en este caso int permite que la variable almacene datos numericos enteros

float es lo mismo que int pero con la diferencia que permite que una variable almacene datos de numeros decimales

se utiliza principalmente para representar y almacenar caracteres individuales, como letras y simbolos

int

float

char

```
c  
#include <stdio.h>  
  
int main() {  
    int edad = 25;  
    float altura = 1.75;  
    char inicial = 'J';
```

variables

Pero no quiero que te marees con int,char,float ,no te preocupes python se encarga de marearse con esos datos.

Ahora quiero enseñarte el print().

print()

El print() te permitira mostrar por la terminal ¿que es la terminal? es el lugar donde se ejecutara tu codigo (en el siguiente pagina te mostare un ejemplo).

Con print() puedes escribir mensajes o mostrar variables ¿como? dentro de los parentesis puedes mostrar tu variable poniendo el nombre que le diste y asi una vez se ejecute se pueda ver que valor tiene esa variable.

O tambien puedes escribir un mensaje entre comillas "holis" y veras que cuando se ejecuta se vera lo que escribiste.

python

```
mensaje = "Hola mundo"  
print(mensaje)
```

variables

ejecucion
del codigo



```
Enseñando.py X
Enseñando.py
1 print("holis")
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
• PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
holis
○ Miguel-PC ➔ □ PYTHON ➔ ✓ |
```

```
Enseñando.py X
Enseñando.py
1 print(24)
2
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
• PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
24
○ Miguel-PC ➔ □ PYTHON ➔ ✓ |
```

Está bien aprendimos que dentro de un print podemos crear mensajes o numeros.

¿pero y como es eso de poner el nombre de una variable y ejecutarlo y que te muestre el valor? •

variables



```
Enseñando.py X
Enseñando.py > ...
1   mensaje = "Hola mundo"
2   print(mensaje)
3

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
● PYTHON > & C:/Users/Miguel-PC/AppData/Local/Po/folders/KALI/PYTHON/Enseñando.py
Hola mundo
○ Miguel-PC > □ PYTHON > ✓
```

```
python
mensaje = "Hola mundo"
print(mensaje)
```

Efectivamente una vez que le damos a ejecutar claramente se puede ver en la terminal el valor que contiene la variable mensaje!

variables

nombres de las variables

En Python, tanto CamelCase como snake_case son convenciones utilizadas para nombrar variables, funciones y clases en el código.

1. CamelCase:

- Es una convención de escritura donde las palabras se escriben juntas sin espacios y cada palabra adicional comienza con una letra mayúscula (excepto la primera palabra).
- Ejemplo: nombreCompleto, sumaTotal, obtenerEdad.
-

2. snake_case:

- Es una convención de escritura donde las palabras se separan con guiones bajos (underscores) y todas las letras son minúsculas.
- Ejemplo: nombre_completo, suma_total, obtener_edad.

```
# Ejemplo de variable en CamelCase  
nombreCompleto = "John Doe"
```

```
# Ejemplo de variable en snake_case  
nombre_completo = "John Doe"
```

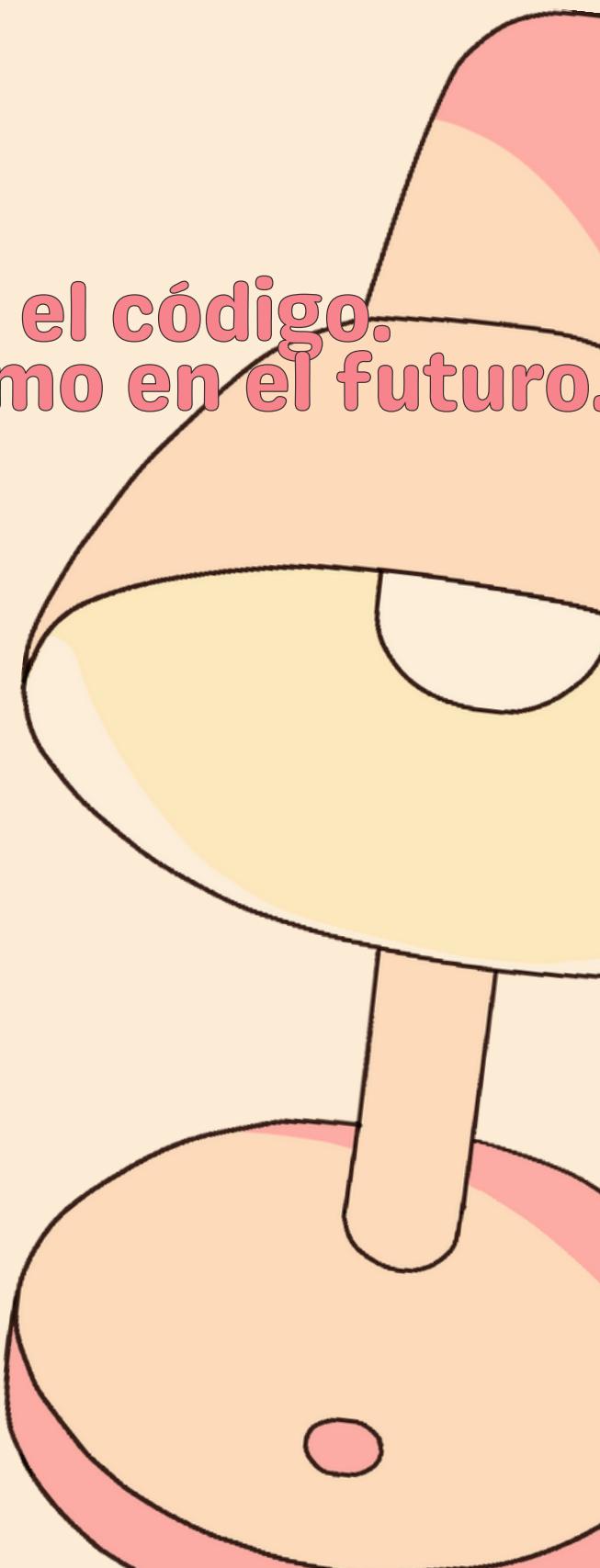
variables

nombres de las variables

cómo usar y por qué se usan los comentarios en Python:

- Los comentarios en Python se crean con el símbolo #.
- Los comentarios son útiles para agregar notas o explicaciones en el código.
- Ayudan a otros programadores a comprender tu código y a ti mismo en el futuro.
- Los comentarios no afectan la ejecución del programa.

Ejemplo:



A cartoon illustration of a yellow-skinned character with a large head, wearing a white lab coat and a stethoscope around their neck, looking thoughtful with one hand near their chin.

```
python
# Este es un comentario en una sola linea

"""
Este es un comentario
de múltiples líneas.
"""

Copy code
```

variables

EJERCICIOS:

- 1 - crea una variable llamado gatito y asignale un string con el nombre que quieras y crea un print que muestre el nombre de tu gatito
- 2- crea una variable que se llame edad_gatito y ponle la edad que quieras y muestralos con un print

variables

EJERCICIOS:solucion

1-

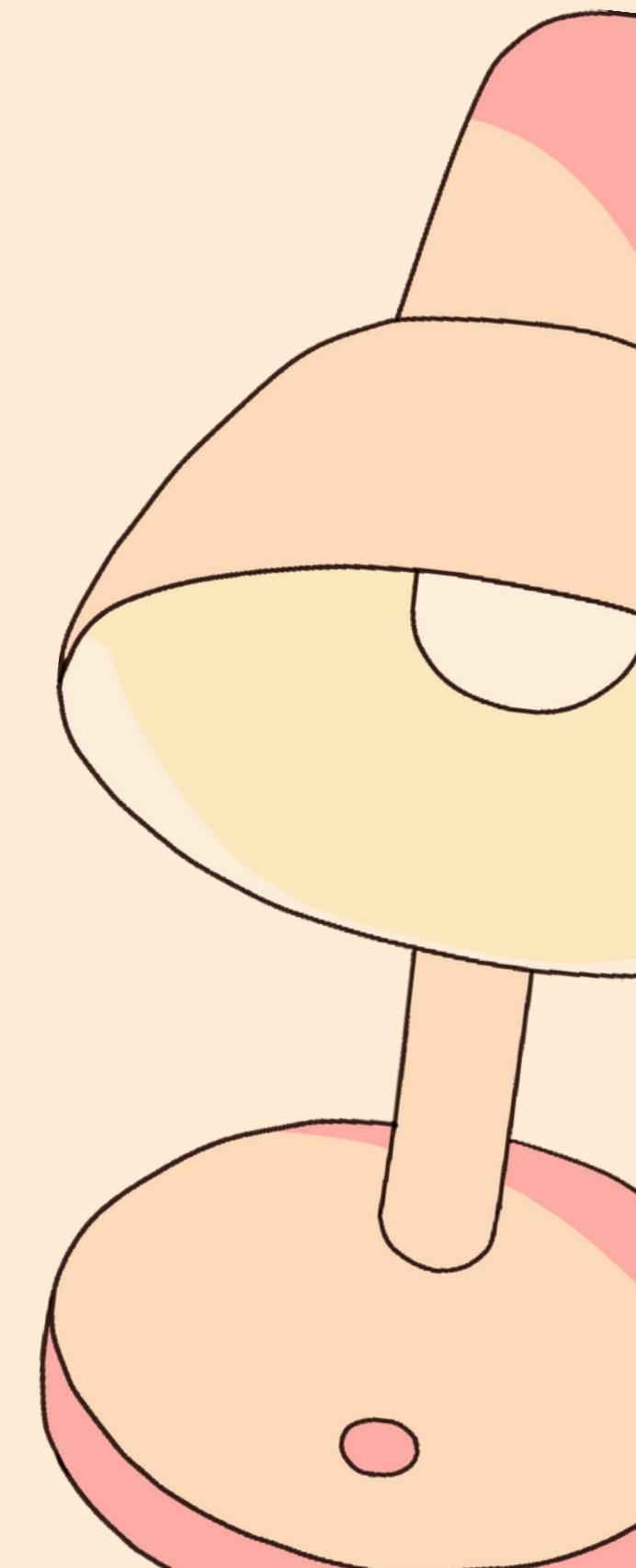
```
python
# Crear una variable llamada 'gatito' y asignarle un nombre
gatito = "Mochi"

# Imprimir el nombre del gatito
print("El nombre de mi gatito es:", gatito)
```

2-

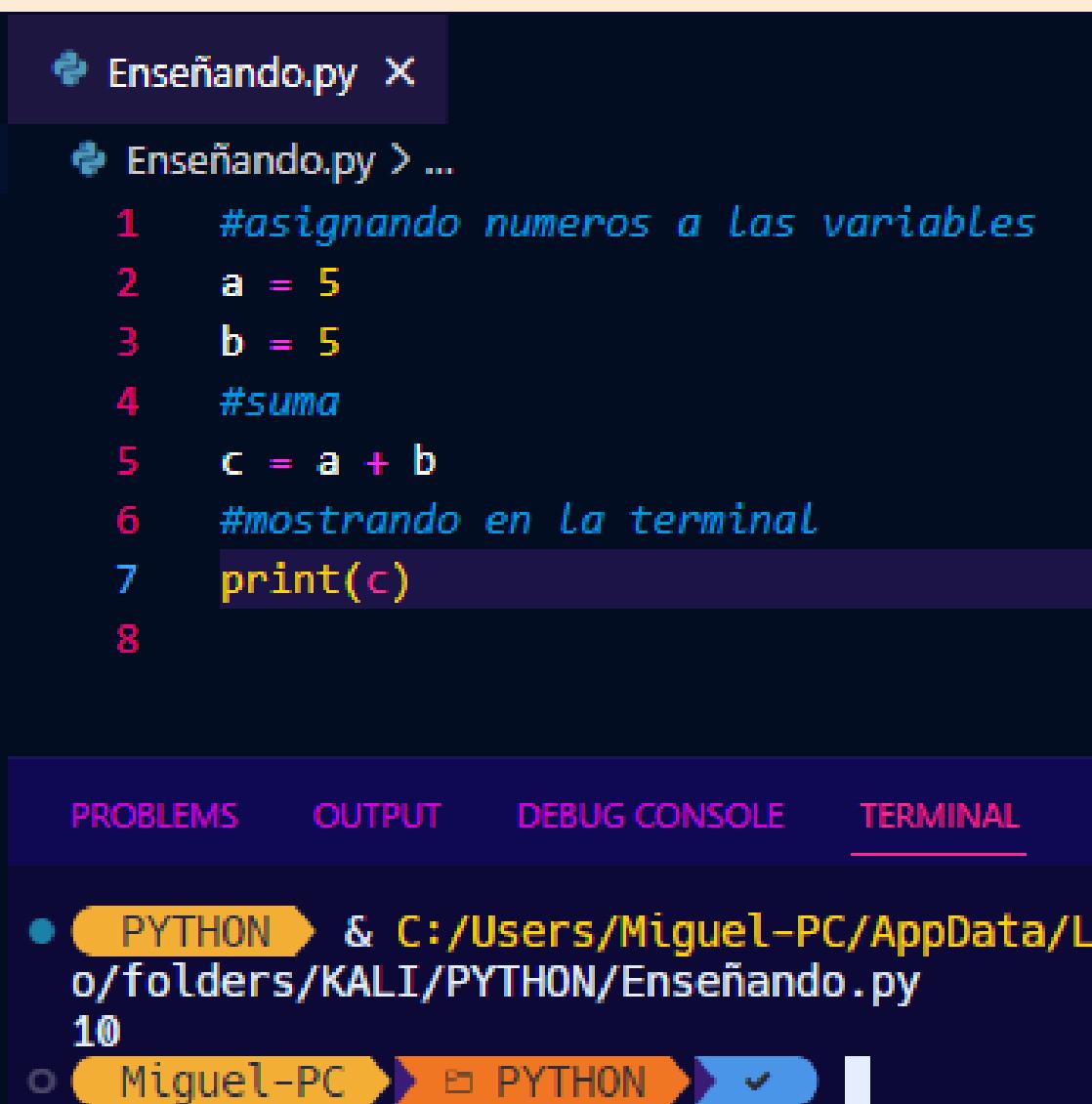
```
python
# Crear una variable llamada 'edad_gatito' y asignarle una edad
edad_gatito = 3

# Mostrar la edad del gatito
print("La edad de mi gatito es:", edad_gatito, "años")
```



operadores

Operadores aritméticos: suma
Hay muchas formas de sumar



```
Enseñando.py X
Enseñando.py > ...
1 #asignando numeros a las variables
2 a = 5
3 b = 5
4 #suma
5 c = a + b
6 #mostrando en La terminal
7 print(c)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py 10
- Miguel-PC ▶ PYTHON

sumando
usando
variables



```
Enseñando.py X
Enseñando.py > ...
1 #asignando numeros a las variables
2 a = 5
3 b = 5
4 #suma
5
6 #mostrando en La terminal
7 print(a + b)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py 10
- Miguel-PC ▶ PYTHON

sumando
en la
terminal



```
Enseñando.py X
Enseñando.py > ...
1 #asignando numeros a Las variables
2 a = 5 + 5
3
4 #suma
5
6 #mostrando en La terminal
7 print(a)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py 10
- Miguel-PC ▶ PYTHON

sumando
directamente
dentro de una
variable

operadores

Operadores aritméticos: resta
Hay muchas formas de restar



```
Enseñando.py X
Enseñando.py > ...
1 #asignando numeros a las variables
2 a = 5
3 b = 5
4 #resta
5 c = a - b
6 #mostrando en la terminal
7 print(c)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py 0
- Miguel-PC ▶ PYTHON ▶ ✓

restando
usando
variables

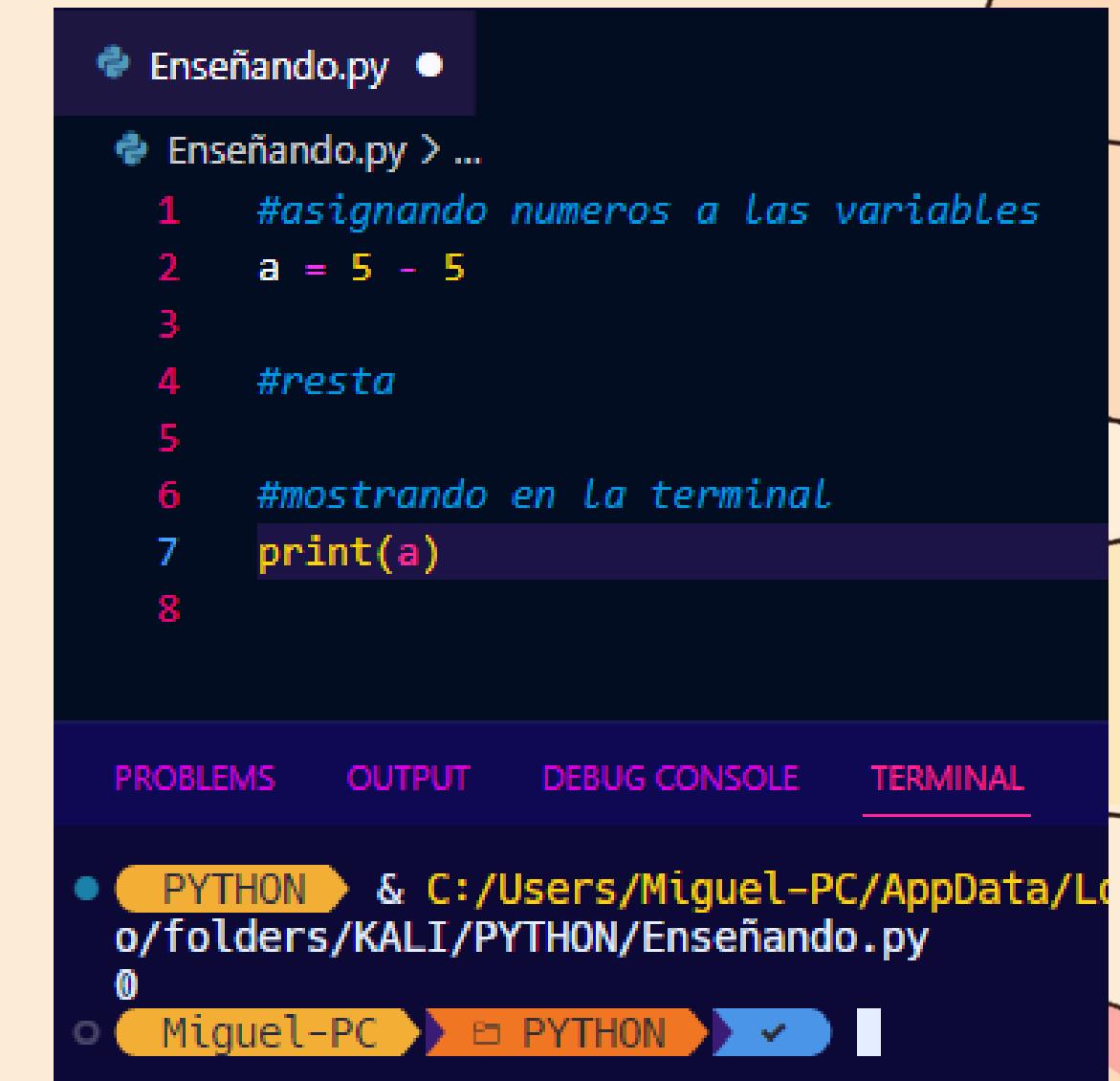


```
Enseñando.py ●
Enseñando.py > ...
1 #asignando numeros a las variables
2 a = 5
3 b = 5
4 #resta
5
6 #mostrando en la terminal
7 print(a - b)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py 0
- Miguel-PC ▶ PYTHON ▶ ✓

restando
en la
terminal



```
Enseñando.py ●
Enseñando.py > ...
1 #asignando numeros a las variables
2 a = 5 - 5
3
4 #resta
5
6 #mostrando en la terminal
7 print(a)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py 0
- Miguel-PC ▶ PYTHON ▶ ✓

restando
directamente
dentro de una
variable

operadores

Operadores aritméticos: multiplicar Hay muchas formas de multiplicar

Enseñando.py

```
1 #asignando numeros a las variables
2 a = 5
3 b = 5
4 #multiplicar
5 c = a * b
6 #mostrando en la terminal
7 print(c)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py 25
- Miguel-PC ▶ PYTHON

multiplicar
usando
variables

Enseñando.py

```
1 #asignando numeros a las variables
2 a = 5
3 b = 5
4 #multiplicar
5
6 #mostrando en la terminal
7 print(a * b)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py 25
- Miguel-PC ▶ PYTHON

multiplicar
en el print

Enseñando.py

```
1 #asignando numeros a las variables
2 a = 5 * 5
3
4 #multiplicar
5
6 #mostrando en la terminal
7 print(a)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py 25
- Miguel-PC ▶ PYTHON

multiplicar
directamente
dentro de una
variable

operadores

Operadores aritméticos: dividir Hay muchas formas de dividir

```
Enseñando.py X
Enseñando.py > ...
1 #asignando numeros a las variables
2 a = 5
3 b = 5
4 #dividir
5 c = a / b
6 #mostrando en la terminal
7 print(c)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py 1.0
- Miguel-PC ▶ PYTHON

dividir
usando
variables

```
Enseñando.py •
Enseñando.py > ...
1 #asignando numeros a las variables
2 a = 5
3 b = 5
4 #dividir
5
6 #mostrando en la terminal
7 print(a / b)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py 1.0
- Miguel-PC ▶ PYTHON

dividir en el
print

```
Enseñando.py X
Enseñando.py > ...
1 #asignando numeros a las variables
2 a = 5 / 5
3
4 #dividir
5
6 #mostrando en la terminal
7 print(a)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py 1.0
- Miguel-PC ▶ PYTHON

dividir
directamente
dentro de una
variable

operadores

Operadores aritméticos: division entera
(devuelve el cociente de una división sin decimales)
Hay muchas formas

```
Enseñando.py X
Enseñando.py > ...
1 #asignando numeros a las variables
2 a = 5
3 b = 5
4 #dividir entero
5 c = a // b
6 #mostrando en la terminal
7 print(c)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
- Miguel-PC ▶ PYTHON

dividir
usando
variables

```
Enseñando.py •
Enseñando.py > ...
1 #asignando numeros a las variables
2 a = 5
3 b = 5
4 #dividir entero
5
6 #mostrando en la terminal
7 print(a // b)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
- Miguel-PC ▶ PYTHON

dividir en el
print

```
Enseñando.py X
Enseñando.py > ...
1 #asignando numeros a las variables
2 a = 5 // 5
3
4 #dividir entero
5
6 #mostrando en la terminal
7 print(a)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
- Miguel-PC ▶ PYTHON

dividir
directamente
dentro de una
variable

operadores

Operadores aritméticos: Módulo
(devuelve el resto de una división)
Hay muchas formas

```
Enseñando.py X
Enseñando.py > ...
1 #asignando numeros a las variables
2 a = 5
3 b = 5
4 #modulo
5 c = a % b
6 #mostrando en la terminal
7 print(c)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py 0
- Miguel-PC ▶ PYTHON ▶ ↴

modulo
usando
variables

```
Enseñando.py X
Enseñando.py > ...
1 #asignando numeros a las variables
2 a = 5
3 b = 5
4 #modulo
5
6 #mostrando en La terminal
7 print(a % b)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py 0
- Miguel-PC ▶ PYTHON ▶ ↴

modulo en
el print

```
Enseñando.py X
Enseñando.py > ...
1 #asignando numeros a Las variables
2 a = 5 % 5
3
4 #modulo
5
6 #mostrando en La terminal
7 print(a)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py 0
- Miguel-PC ▶ PYTHON ▶ ↴

modulo
directamente
dentro de una
variable

operadores

Operadores aritméticos: potencia

Hay muchas formas

```
Enseñando.py X
Enseñando.py > ...
1 #asignando numeros a las variables
2 a = 5
3 b = 5
4 #potencia
5 c = 5 ** 5
6 #mostrando en la terminal
7 print(c)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
3125
- Miguel-PC ▶ PYTHON ➔ ✓

potencia
usando
variables

```
Enseñando.py X
Enseñando.py > ...
1 #asignando numeros a las variables
2 a = 5
3 b = 5
4 #potencia
5
6 #mostrando en la terminal
7 print(a ** b)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
3125
- Miguel-PC ▶ PYTHON ➔ ✓

potencia en
el print

```
Enseñando.py ●
Enseñando.py > ...
1 #asignando numeros a las variables
2 a = 5 ** 5
3
4 #potencia
5
6 #mostrando en la terminal
7 print(a)
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
3125
- Miguel-PC ▶ PYTHON ➔ ✓

potencia
directamente
dentro de una
variable

operadores

Operadores aritméticos: potencia ejercicios :

Problema: Supongamos que tienes una tienda de gatitos y quieres llevar un registro de la cantidad de gatitos disponibles en diferentes colores. Cada gatito tiene un color específico, y quieres saber cuántos gatitos tienes en total y cuántos tienes de cada color.

Tareas:

1. Crea variables para representar la cantidad de gatitos disponibles en cada color. Por ejemplo, puedes tener una variable `gatitos_blanco` para los gatitos blancos, `gatitos_negro` para los gatitos negros, etc. Asigna valores aleatorios a estas variables para simular la cantidad de gatitos en cada color.
2. Crea una variable `total_gatitos` y asígnale el valor de la suma de todas las variables de cantidad de gatitos por color.
3. Imprime en pantalla la cantidad total de gatitos y la cantidad de gatitos en cada color.

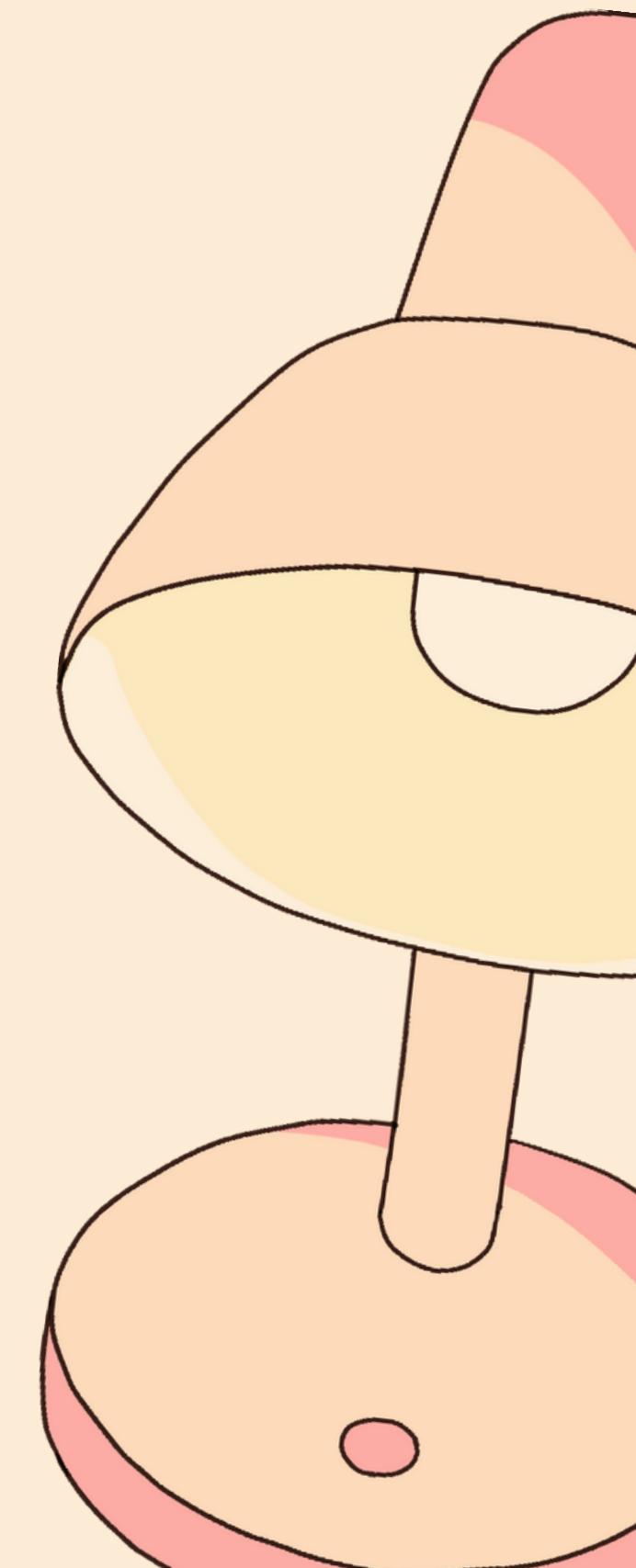


operadores

Operadores aritméticos: potencia

ejercicios :

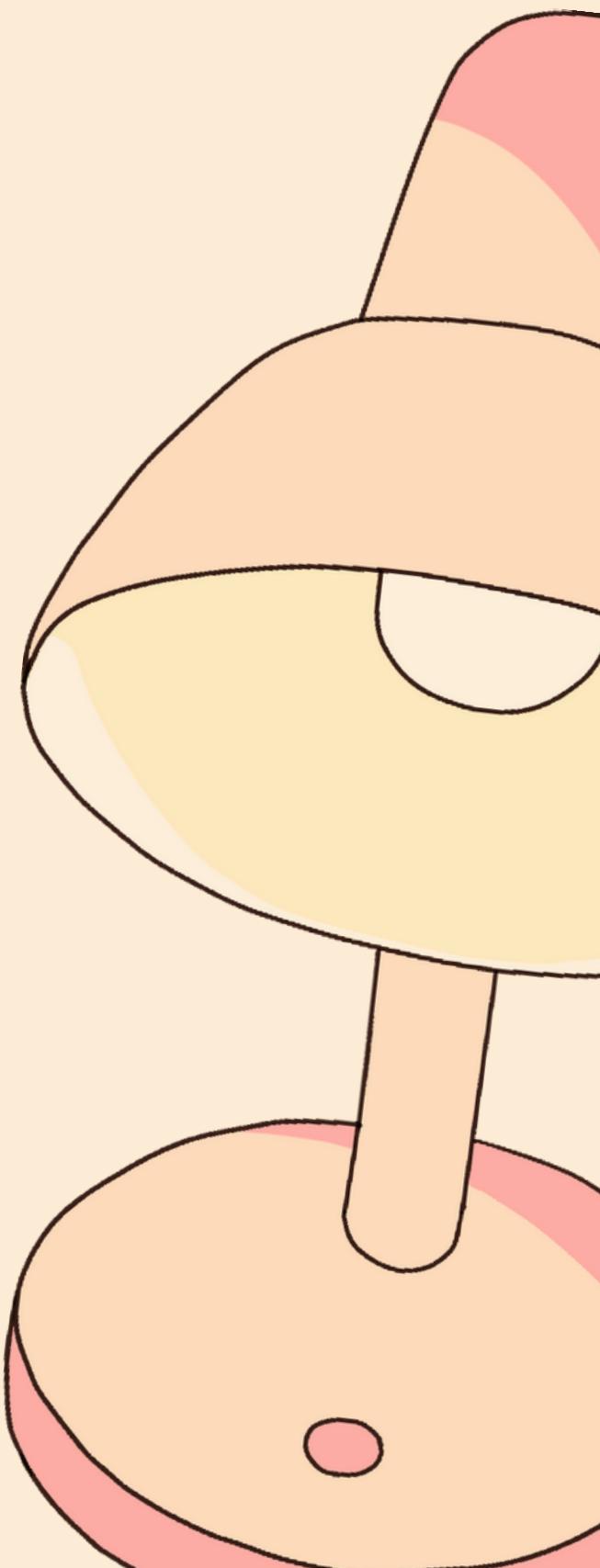
```
python Copy code  
  
# Variables para representar la cantidad de gatitos por color  
gatitos_blancos = 5  
gatitos_negros = 3  
gatitos_marrones = 7  
  
# Cálculo de la cantidad total de gatitos  
total_gatitos = gatitos_blancos + gatitos_negros + gatitos_marrones  
  
# Imprimir los resultados  
print("Cantidad total de gatitos:", total_gatitos)  
print("Cantidad de gatitos blancos:", gatitos_blancos)  
print("Cantidad de gatitos negros:", gatitos_negros)  
print("Cantidad de gatitos marrones:", gatitos_marrones)
```



operadores

Operadores de asignación

Asignación: "="



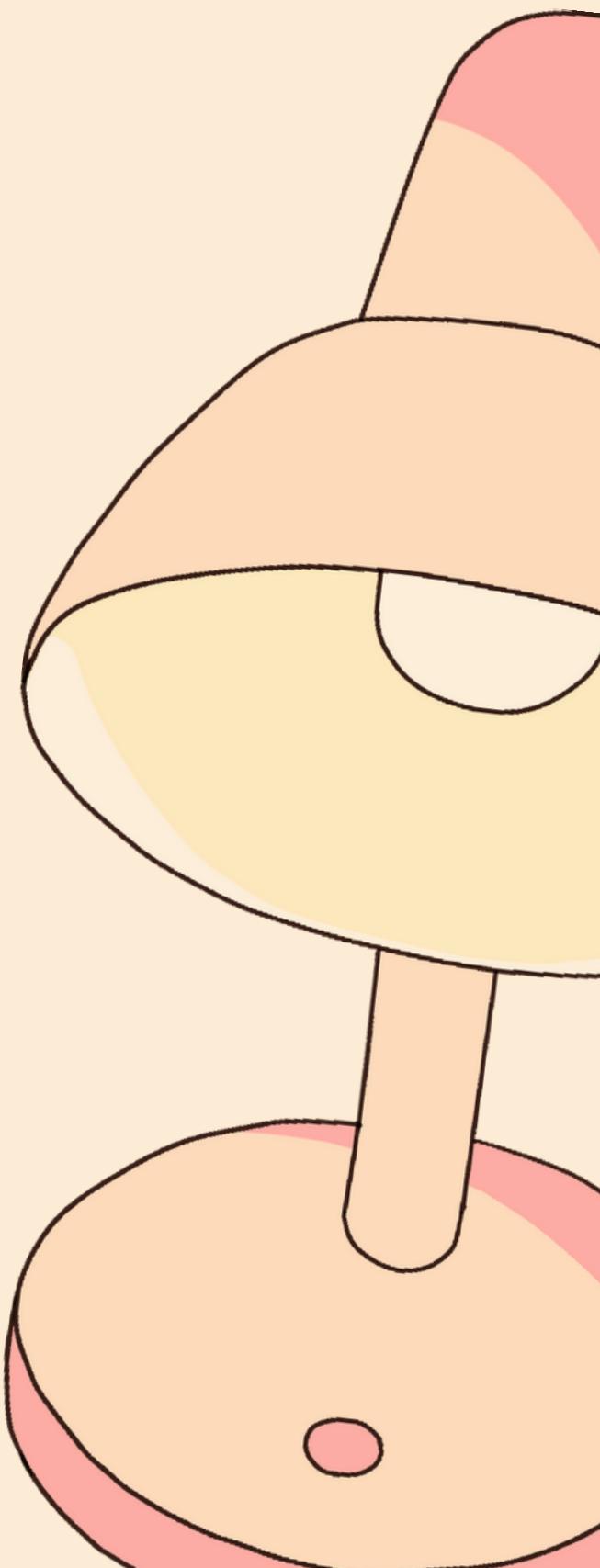
```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a la variable
3 b = 1
4 #asignamos operador
5 a = b
6
7 print(b)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON > & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
1
○ Miguel-PC > □ PYTHON ➤ ✓
```

operadores

Operadores de asignación

Suma y
asignación: "+="



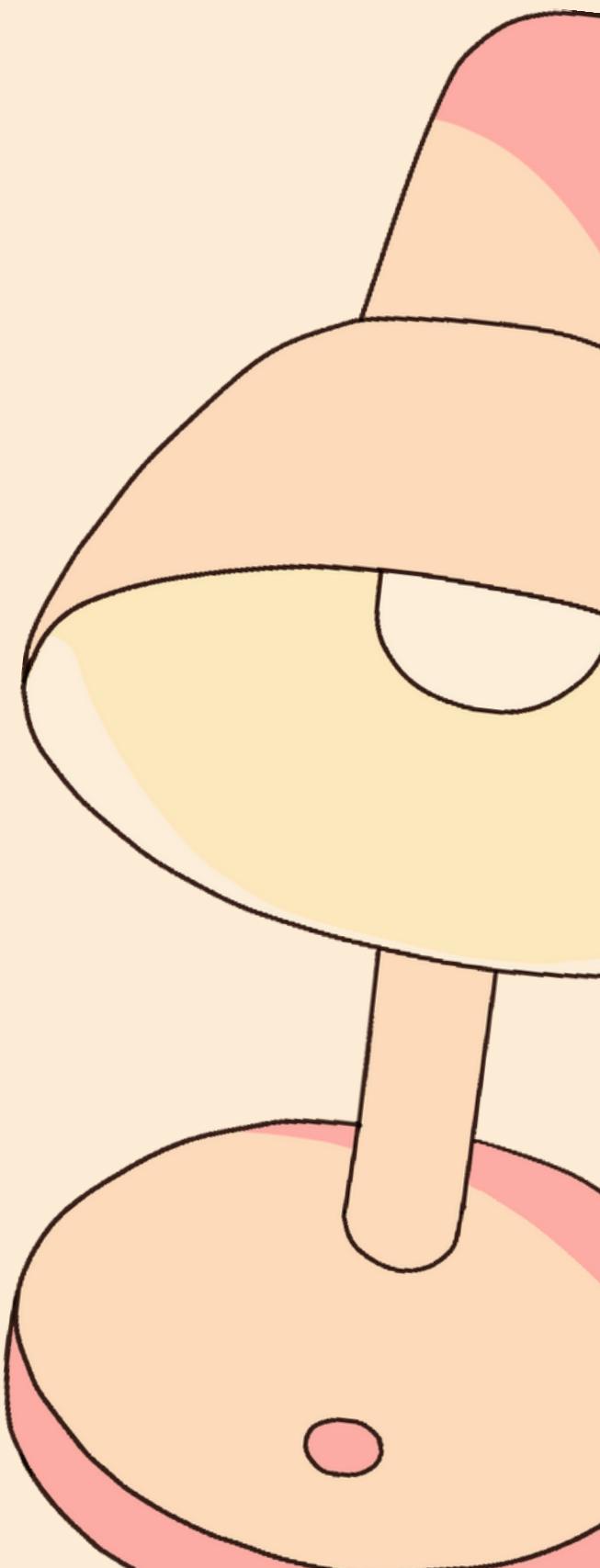
```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a la variable
3 b = 1
4 #asignamos operador
5 b += 1
6
7 print(b)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
2
○ Miguel-PC ▶ □ PYTHON ▶ ▾ □
```

operadores

Operadores de asignación

Resta y
asignación: "-="



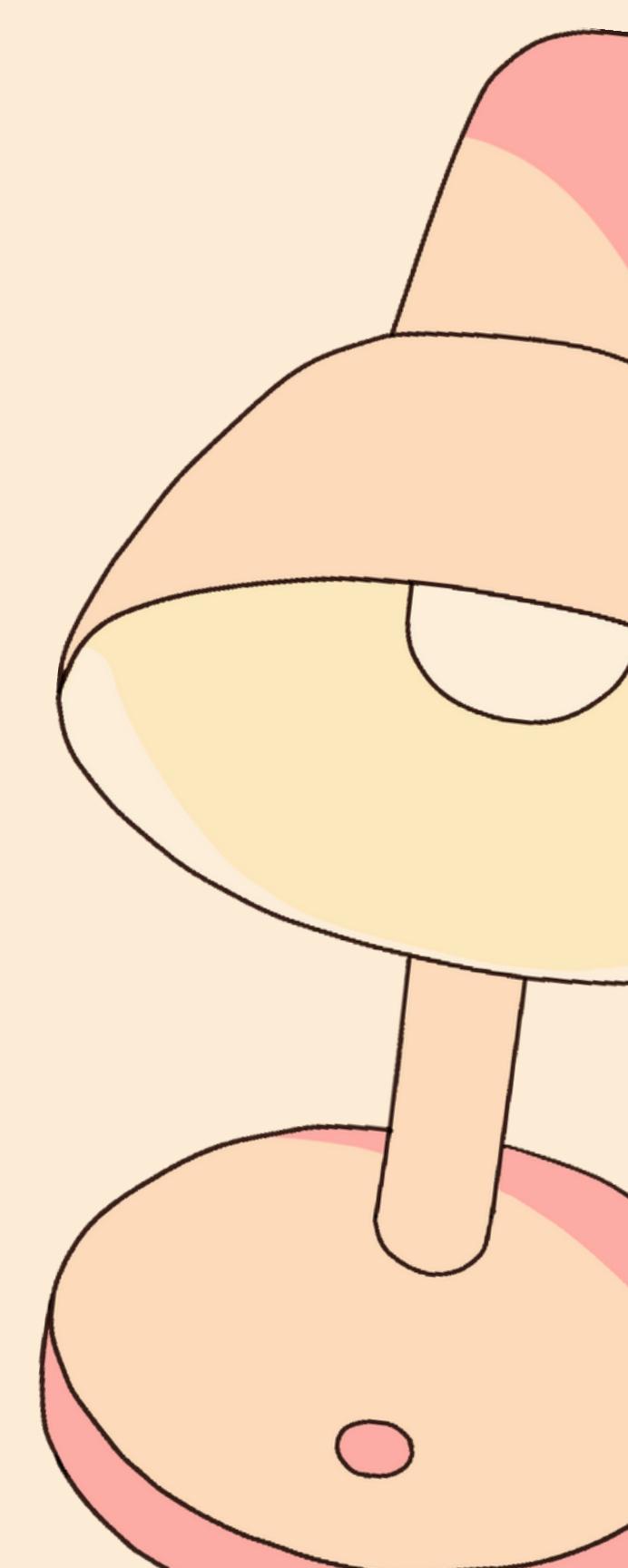
```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a la variable
3 b = 1
4 #asignamos operador
5 b -= 1
6
7 print(b)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
0
○ Miguel-PC ▶ PYTHON ▶ ✓
```

operadores

Operadores de asignación

Multiplicación y
asignación: "*="



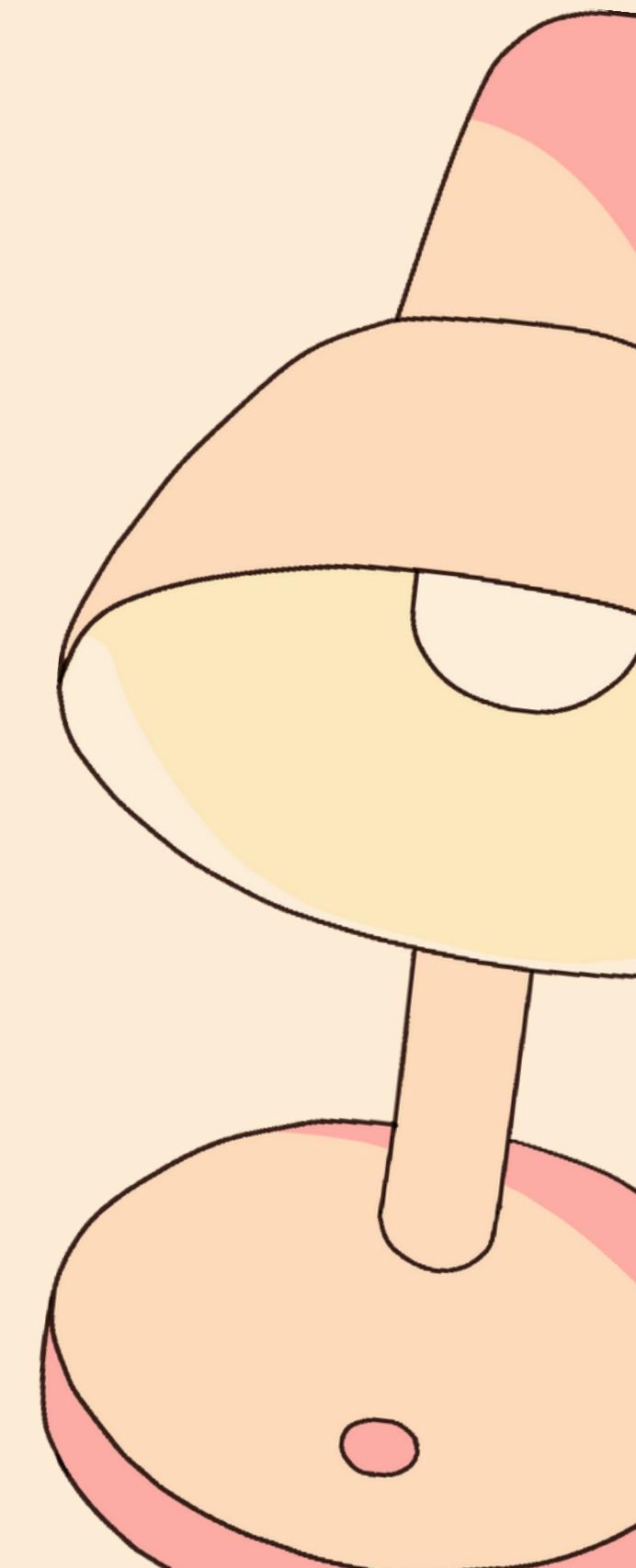
```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a la variable
3 b = 2
4 #asignamos operador
5 b *= 2
6
7 print(b)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
• PYTHON & C:/Users/Miguel-PC/AppData/L
o/folders/KALI/PYTHON/Enseñando.py
4
○ Miguel-PC ▶ PYTHON ▶ ✓ |
```

operadores

Operadores de asignación

División y
asignación: "/="



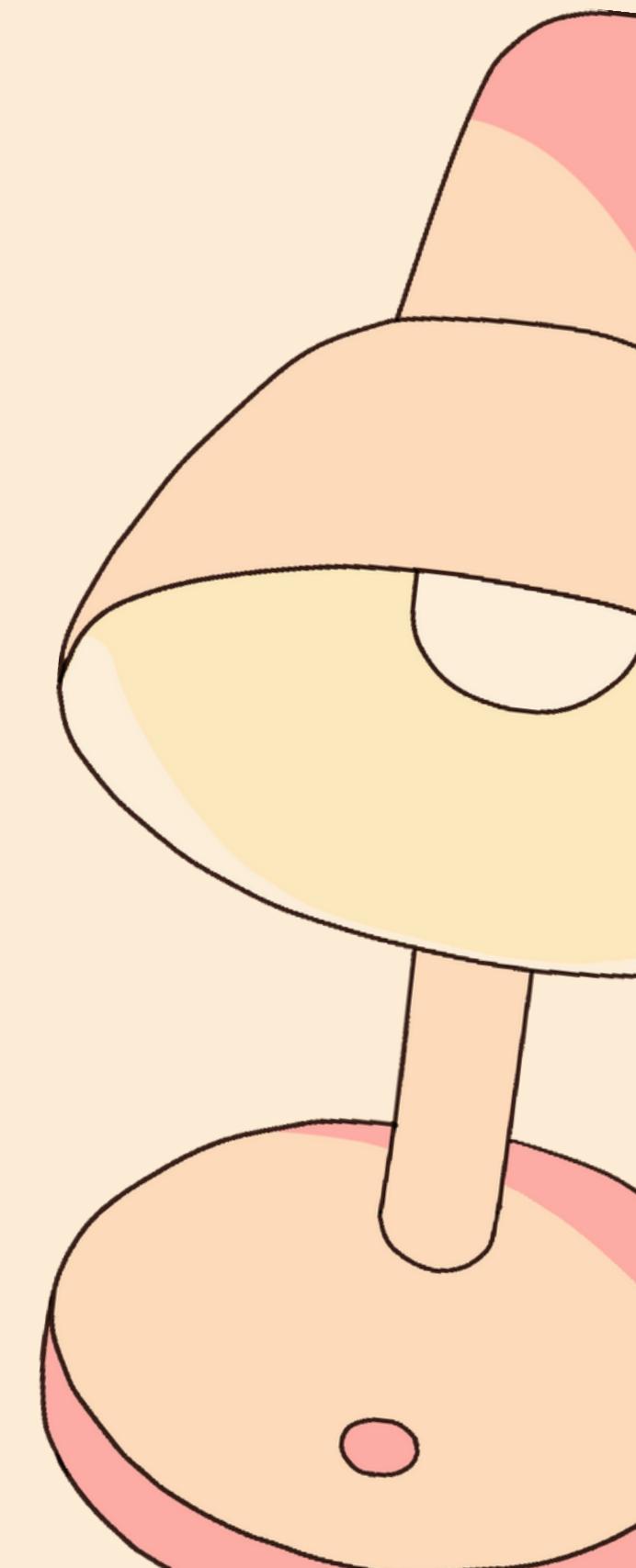
```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a la variable
3 b = 10
4 #asignamos operador
5 b /= 2
6
7 print(b)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
5.0
○ Miguel-PC PYTHON
```

operadores

Operadores de asignación

División entera y
asignación: "`//=`"



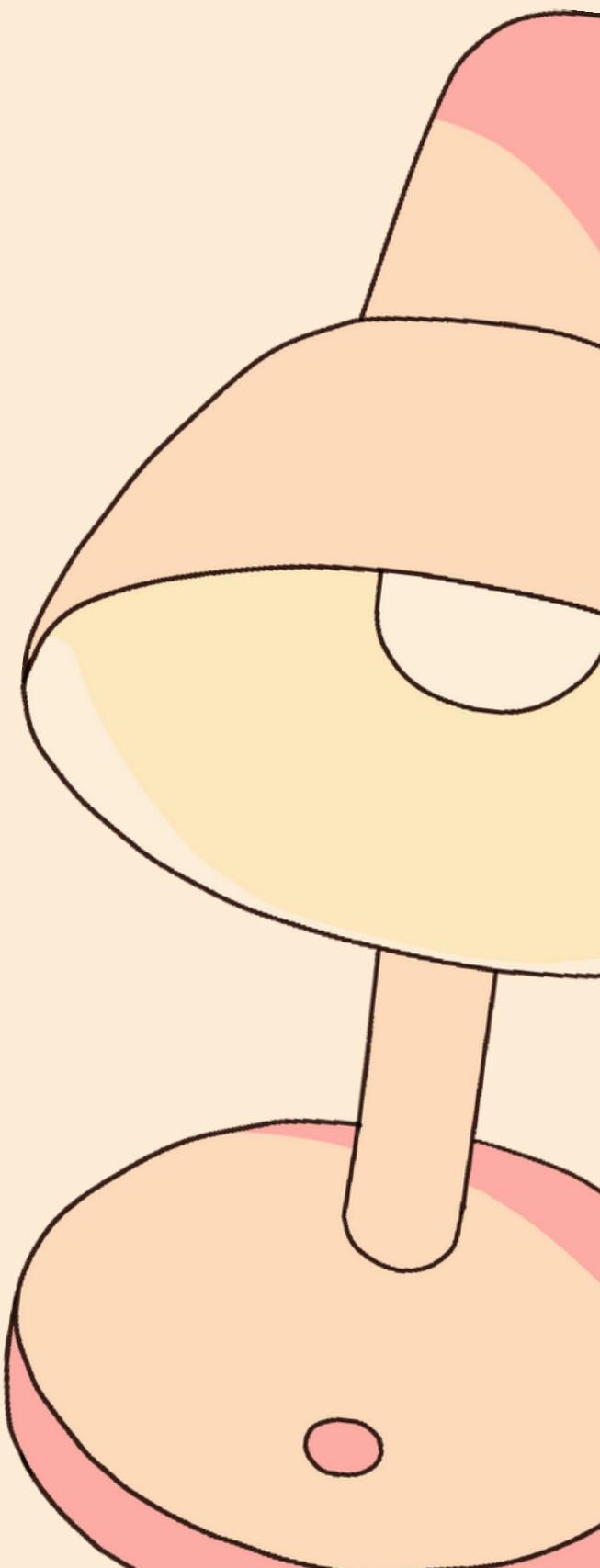
```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a la variable
3 b = 10
4 #asignamos operador
5 b //= 2
6
7 print(b)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/
o/folders/KALI/PYTHON/Enseñando.py
5
○ Miguel-PC ▶ PYTHON ✓
```

operadores

Operadores de asignación

Módulo y
asignación: "%="



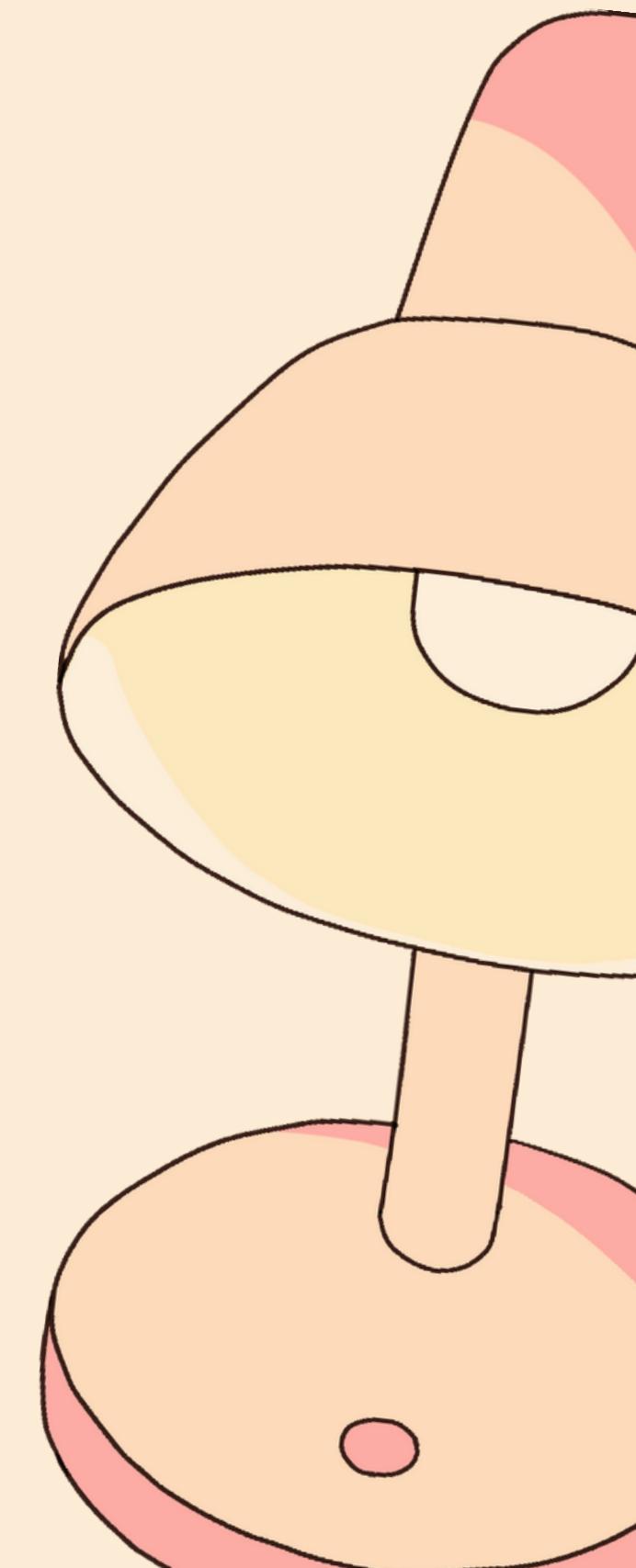
```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a la variable
3 b = 10
4 #asignamos operador
5 b %= 2
6
7 print(b)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON > & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
0
○ Miguel-PC > □ PYTHON ▶ ✓ □
```

operadores

Operadores de asignación

Potencia y
asignación: " $**=$ "



```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a la variable
3 b = 10
4 #asignamos operador
5 b **= 2
6
7 print(b)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON > & C:/Users/Miguel-PC/AppData/
o/folders/KALI/PYTHON/Enseñando.py
100
○ Miguel-PC > □ PYTHON ➤ ✓
```

operadores

Operadores de asignación

ejercicios :

Problema: Supongamos que tienes una tienda de gatitos y quieres llevar un registro de la cantidad de gatitos disponibles. Cada vez que llega un nuevo gatito, deseas actualizar el registro sumando la cantidad de gatitos existentes con la nueva llegada.

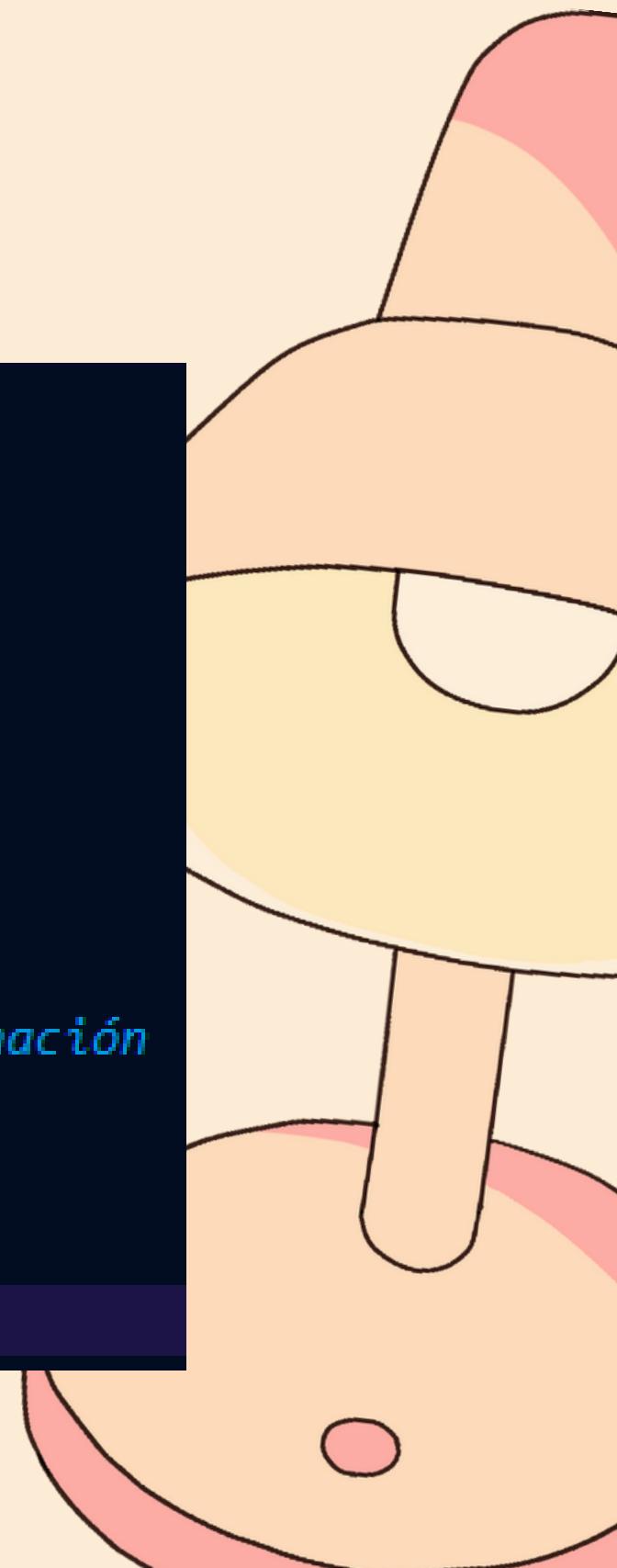
Tareas:

1. Crea una variable llamada `gatitos_existentes` y asignale un valor inicial para representar la cantidad actual de gatitos en tu tienda.
2. Lee la cantidad de nuevos gatitos que llegaron y almacena este valor en una variable llamada `nuevos_gatitos`.
3. Utiliza el operador de asignación adecuado para actualizar la variable `gatitos_existentes` sumando los nuevos gatitos a la cantidad existente.
4. Imprime en pantalla la cantidad total de gatitos después de la actualización.

operadores

Operadores de asignación ejercicios : solucion

```
Enseñando.py •
Enseñando.py > ...
1 # Variable para representar la cantidad de gatitos existentes
2 gatitos_existentes = 10
3
4 # Leer la cantidad de nuevos gatitos
5 nuevos_gatitos = 3
6
7 # Actualizar la cantidad de gatitos existentes utilizando el operador de asignación
8 gatitos_existentes += nuevos_gatitos
9
10 # Imprimir la cantidad total de gatitos después de la actualización
11 print("La cantidad total de gatitos es:", gatitos_existentes)
```



Operadores de comparación

Igualdad: "=="

```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a la variable
3 a = 1
4 b = 1
5 #asignamos operador
6 c = a == b
7
8 print(c)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
True
○ Miguel-PC ▶ PYTHON ▶ ↴ ↵
```

```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a la variable
3 a = 1
4 b = 1
5 #asignamos operador
6
7
8 print(a == b)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
True
○ Miguel-PC ▶ PYTHON ▶ ↴ ↵
```

```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a la variable
3 a = 1 == 2
4
5 #asignamos operador
6
7
8 print(a)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
False
○ Miguel-PC ▶ PYTHON ▶ ↴ ↵
```

el operador de comparacion de igualdad == sirve para comparar dos variables o datos , si la comparacion es correcta devolvera un True si no es correcta un False

Operadores de comparación

Desigualdad: "!="

```
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a La variable
3 a = 1
4 b = 1
5 #asignamos operador
6 c = a != b
7
8 print(c)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/o/folders/KALI/PYTHON/Enseñando.py
False
- Miguel-PC ▶ PYTHON

```
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a La variable
3 a = 1
4 b = 1
5 #asignamos operador
6
7
8 print(a != b)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/o/folders/KALI/PYTHON/Enseñando.py
False
- Miguel-PC ▶ PYTHON

```
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a La variable
3 a = 1 != 2
4
5 #asignamos operador
6
7
8 print(a)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/o/folders/KALI/PYTHON/Enseñando.py
True
- Miguel-PC ▶ PYTHON

Operadores de comparación

Mayor que: ">"

```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a La variable
3 a = 1
4 b = 2
5 #asignamos operador
6 c = a > b
7
8 print(c)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/L
o/folders/KALI/PYTHON/Enseñando.py
False
○ Miguel-PC ▶ PYTHON ▶ ✓ |
```

```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a La variable
3 a = 1
4 b = 2
5 #asignamos operador
6
7
8 print(a > b)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/L
o/folders/KALI/PYTHON/Enseñando.py
False
○ Miguel-PC ▶ PYTHON ▶ ✓ |
```

```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a La variable
3 a = 1 > 0
4
5 #asignamos operador
6
7
8 print(a)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/L
o/folders/KALI/PYTHON/Enseñando.py
True
○ Miguel-PC ▶ PYTHON ▶ ✓ |
```

Operadores de comparación

Menor que: "<"

```
1 #Operadores de asignación
2 #asignamos un numero a la variable
3 a = 1
4 b = 2
5 #asignamos operador
6 c = a < b
7
8 print(c)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
True
- Miguel-PC ▶ PYTHON ▶

```
1 #Operadores de asignación
2 #asignamos un numero a la variable
3 a = 1
4 b = 2
5 #asignamos operador
6
7 print(a < b)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
True
- Miguel-PC ▶ PYTHON ▶

```
1 #Operadores de asignación
2 #asignamos un numero a la variable
3 a = 1 < 0
4 #asignamos operador
5
6
7 print(a)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
False
- Miguel-PC ▶ PYTHON ▶

Operadores de comparación

Mayor o igual que:
">="

```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a La variable
3 a = 1
4 b = 1
5 #asignamos operador
6 c = a >= b
7
8 print(c)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
True
○ Miguel-PC ▶ PYTHON ▶ ✓
```

```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a La variable
3 a = 1
4 b = 1
5 #asignamos operador
6
7
8 print(a >= b)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
True
○ Miguel-PC ▶ PYTHON ▶ ✓
```

```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a La variable
3 a = 0 >= 1
4
5 #asignamos operador
6
7
8 print(a)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
False
○ Miguel-PC ▶ PYTHON ▶ ✓
```

el resultado fue True por que a pesar de que b no es mayor que a , a y b si son iguales , o sea no cumplieron la condición de ser mayor pero si iguales

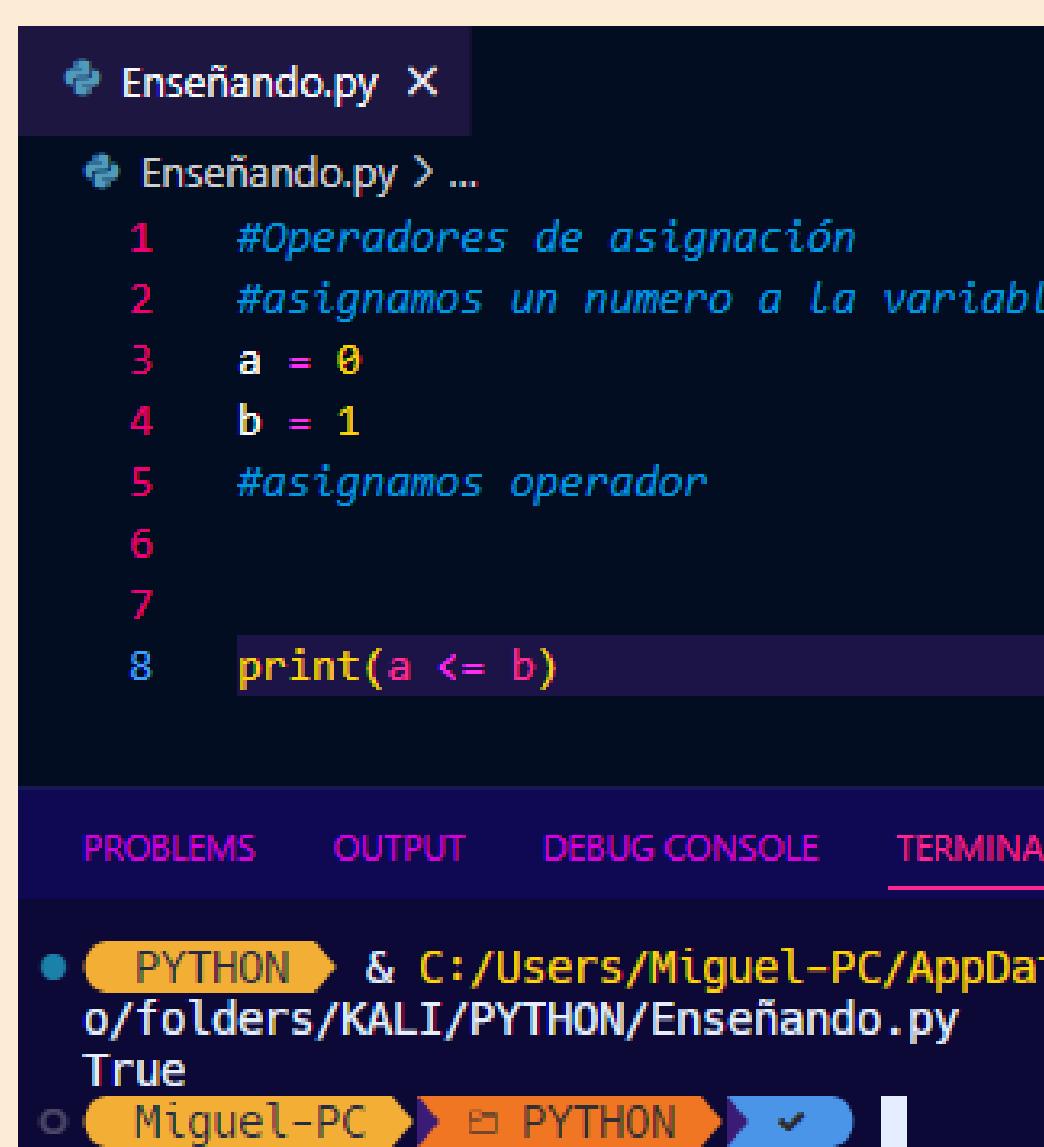
Operadores de comparación

Menor o igual que:
" \leq "



```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a la variable
3 a = 0
4 b = 1
5 #asignamos operador
6 c = a <= b
7
8 print(c)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
True
○ Miguel-PC ▶ PYTHON ▶
```



```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a la variable
3 a = 0
4 b = 1
5 #asignamos operador
6
7
8 print(a <= b)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
False
○ Miguel-PC ▶ PYTHON ▶
```



```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a la variable
3 a = 2 <= 1
4
5 #asignamos operador
6
7
8 print(a)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
False
○ Miguel-PC ▶ PYTHON ▶
```

Operadores de comparación

Ejercicios :

Problema: Supongamos que tienes una tienda de gatitos y deseas verificar si la cantidad de gatitos existentes supera un límite máximo permitido. En lugar de utilizar la instrucción if, deseas utilizar operadores de comparación para obtener un resultado booleano directamente.

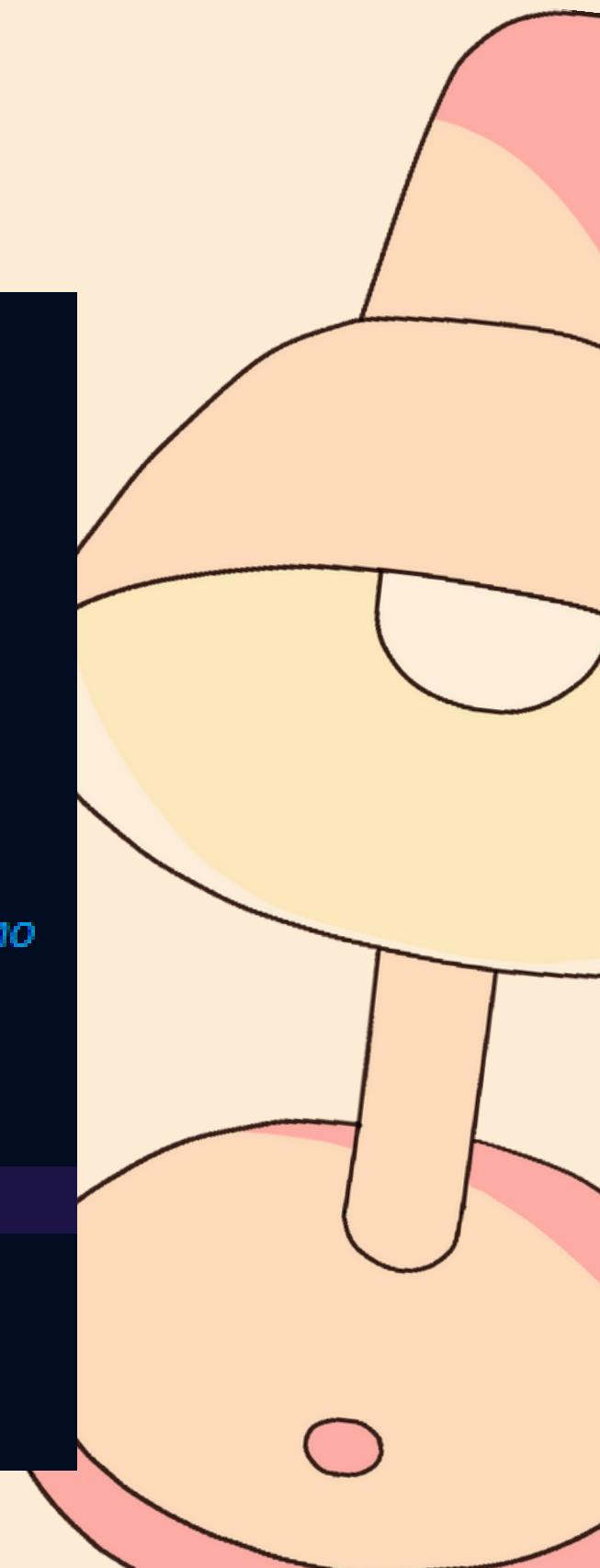
Tareas:

1. Crea una variable llamada `gatitos_existentes` y asignale un valor inicial para representar la cantidad actual de gatitos en tu tienda.
2. Crea una variable llamada `limite_maximo` y asignale el valor máximo permitido para la cantidad de gatitos.
3. Utiliza un operador de comparación para verificar si `gatitos_existentes` es mayor que `limite_maximo` y almacena el resultado en una variable llamada `se_supera_límite`.
4. Imprime el valor de `se_supera_límite`, que será True si la cantidad de gatitos supera el límite máximo y False en caso contrario.

Operadores de comparación

Ejercicios : solución

```
Enseñando.py •
Enseñando.py > ...
1 # Variable para representar la cantidad de gatitos existentes
2 gatitos_existentes = 12
3
4 # Variable para representar el límite máximo permitido de gatitos
5 limite_maximo = 10
6
7 # Utilizar operadores de comparación para verificar si se supera el límite máximo
8 se_suptera_límite = gatitos_existentes > limite_maximo
9
10 # Imprimir el resultado de la comparación
11 print("Se supera el límite máximo:", se_suptera_límite)
12
13
14
```



Operadores lógicos

Y lógico: "and"

Lo que estamos diciendo en la variable a es que , si uno es identico a uno y 5 no es identico a 9 devuelveme True o False , en este caso como las dos condiciones son verdaderas sera True

```
Enseñando.py > ...
#Operadores de asignación
#asignamos un numero a la variable
a = 1
b = 1
#asignamos operador
c = a == 1 and b == 1
print(c)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
True
○ Miguel-PC ▶ PYTHON ▶ ↴
```

and se utiliza para verificar si dos condiciones son verdaderas al mismo tiempo. Si alguna de las condiciones es falsa, el resultado será False.

```
Enseñando.py > ...
#Operadores de asignación
#asignamos un numero a la variable
a = 1
b = 1
#asignamos operador
print(a == 1 and b == 2)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
False
○ Miguel-PC ▶ PYTHON ▶ ↴
```

```
Enseñando.py > ...
#Operadores de asignación
#asignamos un numero a la variable
a = 1 == 1 and 5 != 9
#asignamos operador
print(a)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
True
○ Miguel-PC ▶ PYTHON ▶ ↴
```

Operadores lógicos

O lógico: "or"

A diferencia de AND aqui con el operador lógico or solamente se requiere que una de las dos condiciones sea verdadera para que el resultado final sea True

```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a La variable
3 a = 1
4 b = 1
5 #asignamos operador
6 c = a == 1 or b == 8
7
8 print(c)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/
o/folders/KALI/PYTHON/Enseñando.py
True
○ Miguel-PC ▶ PYTHON ▶ ✓
```

```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a La variable
3 a = 1
4 b = 1
5 #asignamos operador
6
7
8 print(a == 2 or b == 1)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/
o/folders/KALI/PYTHON/Enseñando.py
True
○ Miguel-PC ▶ PYTHON ▶ ✓
```

```
Enseñando.py X
Enseñando.py > ...
1 #Operadores de asignación
2 #asignamos un numero a La variable
3 a = 1 == 2 or 23 == 1
4
5 #asignamos operador
6
7
8 print(a)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/
o/folders/KALI/PYTHON/Enseñando.py
False
○ Miguel-PC ▶ PYTHON ▶ ✓
```

Operadores lógicos

ejercicios :

Problema: Supongamos que tienes una tienda de gatitos y deseas verificar si puedes aceptar nuevos gatitos en función de dos condiciones: la cantidad actual de gatitos en tu tienda y si tienes suficiente espacio disponible en tu establecimiento.

Tareas:

1. Crea una variable llamada `gatitos_existentes` y asignale un valor inicial para representar la cantidad actual de gatitos en tu tienda.
2. Crea una variable llamada `espacio_disponible` y asignale un valor booleano (True o False) para indicar si tienes suficiente espacio para nuevos gatitos.
3. Utiliza operadores lógicos para verificar si puedes aceptar nuevos gatitos. La condición será que la cantidad de gatitos existentes sea menor que un límite máximo permitido y que haya espacio disponible en tu tienda.
4. Almacena el resultado de la verificación en una variable llamada `se_pueden_aceptar_gatitos`.
5. Imprime el valor de `se_pueden_aceptar_gatitos`, que será True si se cumplen ambas condiciones y False en caso contrario.

Operadores lógicos

ejercicios :

python

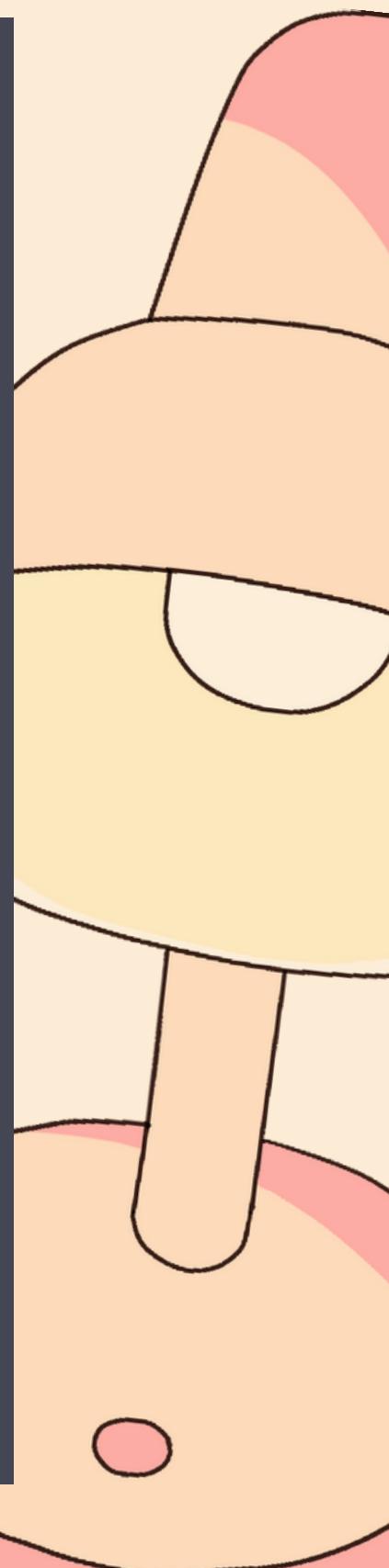
 Copy code

```
# Variable para representar la cantidad de gatitos existentes
gatitos_existentes = 8

# Variable para representar si tienes suficiente espacio disponible
espacio_disponible = True

# Utilizar operadores lógicos para verificar si se pueden aceptar nuevos gatitos
se_pueden_aceptar_gatitos = gatitos_existentes < 10 and espacio_disponible

# Imprimir el resultado de la verificación
print("Se pueden aceptar nuevos gatitos:", se_pueden_aceptar_gatitos)
```



strings

Definir strings:

En Python, los strings son secuencias de caracteres encerrados entre comillas, ya sean simples o dobles.
Aquí tienes algunos ejemplos:

string en una variable

```
Enseñando.py X
Enseñando.py > ...
1 #strings
2 Saludando = "Hola chiques"
3 print(Saludando)
```

The screenshot shows a code editor window titled "Enseñando.py". The code consists of three lines: a comment "#strings", an assignment statement "Saludando = "Hola chiques"" where the string is enclosed in double quotes, and a call to the print function "print(Saludando)". The code editor has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL at the bottom.

comillas dobles

```
Enseñando.py X
Enseñando.py > ...
1 #strings
2 Saludando = 'Hola chiques'
3 print(Saludando)
```

The screenshot shows a code editor window titled "Enseñando.py". The code is identical to the previous one, but the string "Hola chiques" is enclosed in single quotes instead of double quotes. The code editor has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL at the bottom.

comillas simples

```
Enseñando.py X
Enseñando.py
1 #strings
2 print("Hola chiques")
```

The screenshot shows a code editor window titled "Enseñando.py". The code consists of two lines: a comment "#strings" and a call to the print function "print("Hola chiques")". The code editor has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL at the bottom.

comillas dobles

```
Enseñando.py X
Enseñando.py
1 #strings
2 print('Hola chiques')
```

The screenshot shows a code editor window titled "Enseñando.py". The code is identical to the previous one, but the string "Hola chiques" is enclosed in single quotes instead of double quotes. The code editor has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL at the bottom.

comillas simples

string en un print

strings

Acceder a caracteres individuales:

En Python, puedes acceder a caracteres individuales de un string utilizando la indexación. Cada carácter en un string tiene una posición o índice, empezando desde 0 para el primer carácter. Puedes acceder a un carácter específico utilizando corchetes [] después del nombre del string, y dentro de los corchetes colocas el índice del carácter que deseas acceder. Aquí tienes un ejemplo:

```
Enseñando.py X
Enseñando.py > ...
1 #strings
2 saludar = "Hola"
3
4
5 print(saludar[0])
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PYTHON & C:/Users/Miguel-PC/AppData/folders/KALI/PYTHON/Enseñando.py H

Miguel-PC PYTHON

Saludar = "Hola"
[0] [1] [2] [3]

A diferencia de empezar a contar desde uno como normalmente hacemos en código es diferente, se cuenta el cero a así que tenemos que tener en cuenta que siempre el primer carácter tiene el índice [0], el segundo carácter tendrá el índice [1], el tercero tendrá el índice [2] y por último en nuestro texto Hola el último carácter tendrá el índice [3]

```
Enseñando.py X
Enseñando.py > ...
1 #strings
2 saludar = "Hola"
3
4 caracter1 = saludar[0]
5 print(caracter1)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PYTHON & C:/Users/Miguel-PC/AppData/folders/KALI/PYTHON/Enseñando.py H

Miguel-PC PYTHON

desde el print podemos llamar a nuestra variable saludar que contiene el texto ,y con [] podemos elegir la letra que queremos por ejemplo nosotros pusimos 0 significa que quiero el primer carácter. o sea H

strings

Longitud de un string:

Puedes utilizar la función len() para obtener la cantidad de caracteres en un string. Aquí tienes un ejemplo:

```
Enseñando.py X
Enseñando.py > ...
1 #strings
2 saludar = "Hola"
3 longitud = len(saludar)
4
5 print(longitud)
```

En el ejemplo anterior, hemos definido una variable llamada saludar que contiene el string "Hola". Luego, hemos utilizado la función len() para obtener la longitud del string, es decir, la cantidad de caracteres que contiene. El resultado se almacena en la variable longitud o como hayas llamado a tu variable que contendrá la operación o si no también hay otra forma y es hacer la operación directamente en el print , y luego se imprime en la consola utilizando la función print().

The screenshot shows a Python code editor interface. At the top, there are tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMI'. Below the tabs, the code is displayed in a dark-themed code editor:

```
Enseñando.py X
Enseñando.py > ...
1 #strings
2 saludar = "Hola"
3 longitud = len(saludar)
4
5 print(longitud)
```

At the bottom of the editor, there is a status bar with the following information:

- A terminal icon followed by the text "PYTHON" and the path "& C:/Users/Miguel-PC/AppD.../folders/KALI/PYTHON/Enseñando.py".
- The number "4" indicating the current line of code.
- An icon for the operating system (Windows) followed by the text "Miguel-PC".
- An icon for Python followed by the text "PYTHON".
- A checkmark icon indicating the code is valid.
- A progress bar at the end of the status bar.

The screenshot shows a Python code editor interface with a terminal tab active. The terminal window displays the output of the script "Enseñando.py":

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppD.../folders/KALI/PYTHON/Enseñando.py
4
● Miguel-PC PYTHON ✓
```

The terminal output shows the string "Hola" and its length, which is 4. The status bar at the bottom of the editor is identical to the one in the previous screenshot.

strings

Concatenación de strings:

En Python, puedes concatenar (unir) dos o más strings utilizando el operador + :

```
Enseñando.py X
Enseñando.py > ...
1 #strings
2 string1 = "Hola"
3 string2 = "mundo"
4
5 print(string1 + " " + string2)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
```

• PYTHON & C:/Users/Miguel-PC/AppData/folders/KALI/PYTHON/Enseñando.py
Hola mundo

○ Miguel-PC ▶ PYTHON ➔ ✓

tambien se puede hacer sin usar una variable

En este ejemplo, hemos definido dos variables `string1` y `string2`, que contienen los strings "Hola" y "mundo", respectivamente. Luego, utilizamos el operador + para concatenarlos junto con un espacio en blanco entre ellos. El resultado de la concatenación se almacena en la variable `concatenacion` y se imprime en la consola.

```
Enseñando.py X
Enseñando.py > ...
1 #strings
2 string1 = "Hola"
3 string2 = "mundo"
4 concatenacion = string1 + " " + string2
5 print(concatenacion)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
```

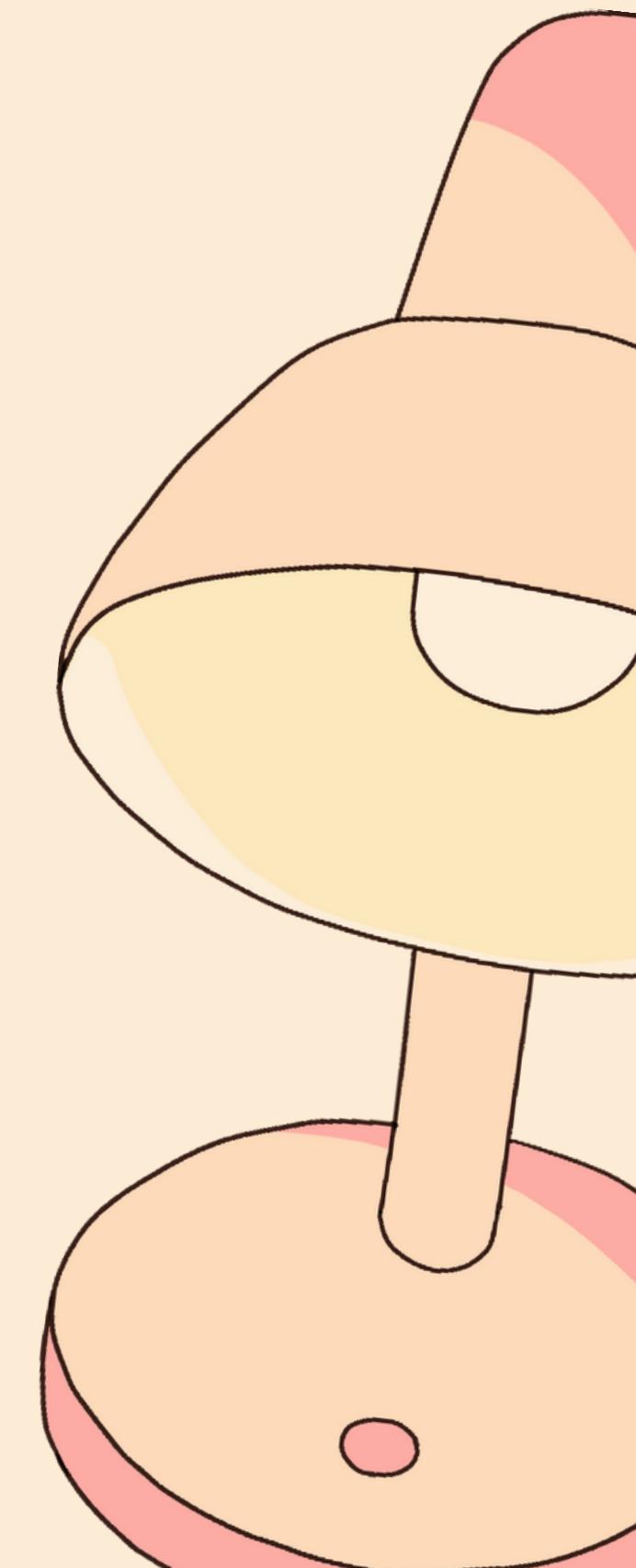
• PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
Hola mundo

○ Miguel-PC ▶ PYTHON ➔ ✓

strings

Formateo de strings:

En Python, puedes formatear strings utilizando diversas técnicas.
Utilizando el método `format()`:



```
Enseñando.py X
Enseñando.py > ...
1 #strings
2 nombre = "Juan"
3 edad = 25
4 saludo = "Hola, mi nombre es {} y tengo {} años.".format(nombre, edad)
5 print(saludo)
6

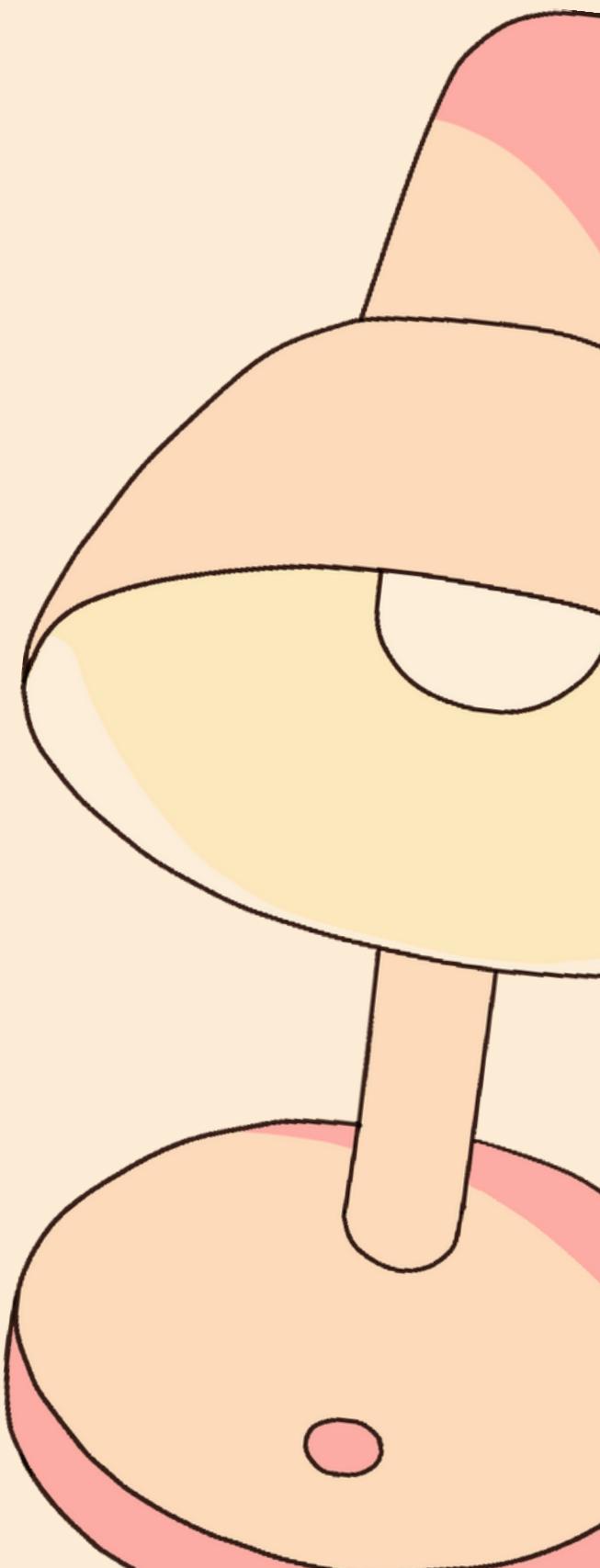
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python311/py
o/folders/KALI/PYTHON/Enseñando.py
Hola, mi nombre es Juan y tengo 25 años.
○ Miguel-PC ▶ PYTHON ✓ |
```

En este caso, utilizamos el método `format()` en el string `saludo` para indicar los marcadores de posición `{}`. Los valores correspondientes se pasan al método `format()` en el orden en que deben reemplazar los marcadores de posición.

strings

Formateo de strings:

En Python, puedes formatear strings utilizando diversas técnicas.
Utilizando f-strings (formateo de cadenas literales)::



```
Enseñando.py X
Enseñando.py > ...
1 #strings
2 nombre = "Juan"
3 edad = 25
4 saludo = f"Hola, mi nombre es {nombre} y tengo {edad} años."
5 print(saludo)
6
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

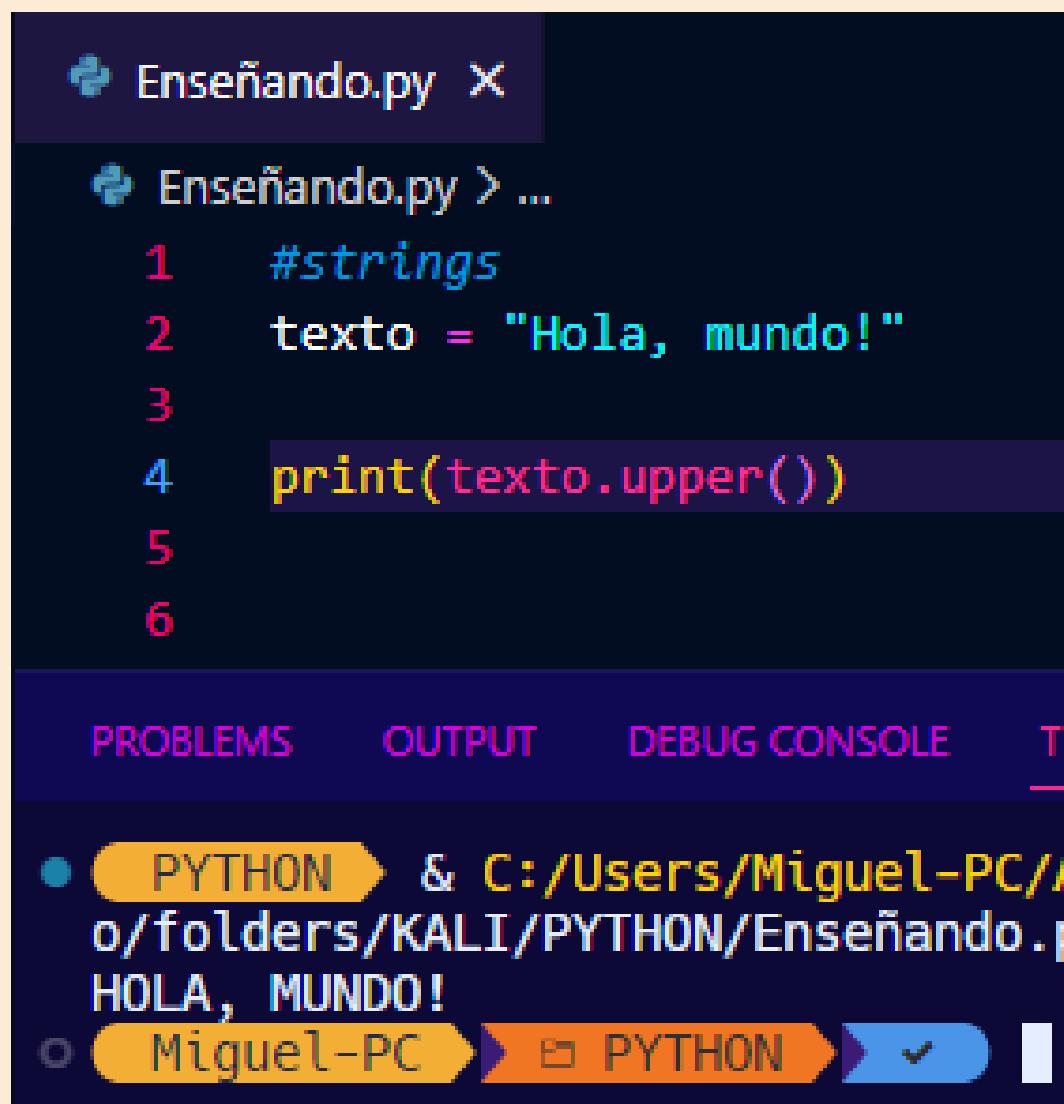
- PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python38/folders/KALI/PYTHON/Enseñando.py
Hola, mi nombre es Juan y tengo 25 años.
- Miguel-PC ▶ PYTHON ▶ ✓

Con f-strings, puedes incluir expresiones dentro de llaves {} directamente en el string. Las expresiones dentro de las llaves se evalúan y se reemplazan con sus valores correspondientes durante la ejecución del programa.:

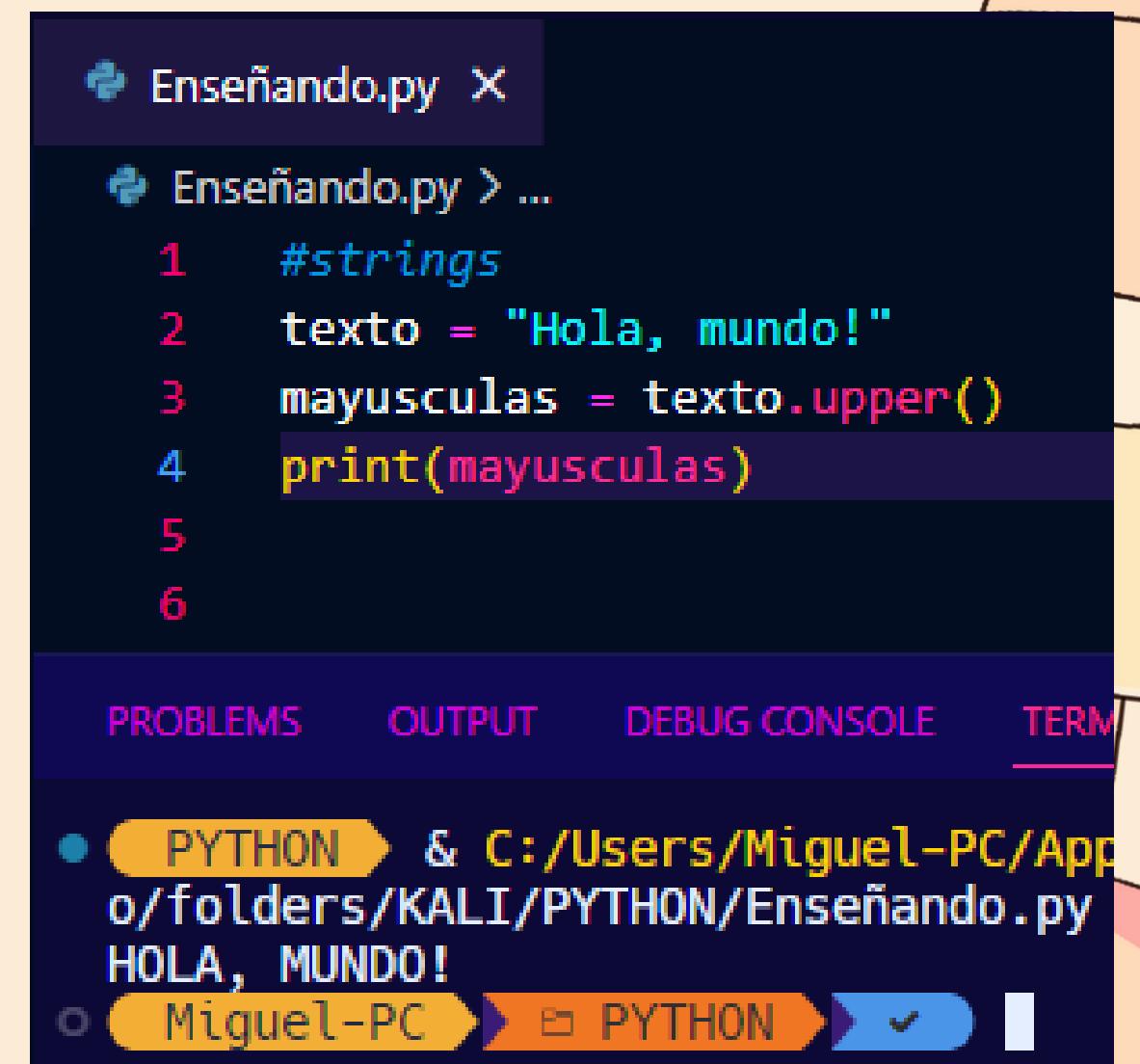
strings

Métodos de strings:

En Python, los strings son objetos y tienen varios métodos incorporados que te permiten realizar diversas operaciones y manipulaciones en los strings. Aquí tienes algunos métodos comunes de los strings en Python:
`upper()`: Convierte un string a mayúsculas.



```
#strings
texto = "Hola, mundo!"
print(texto.upper())
```

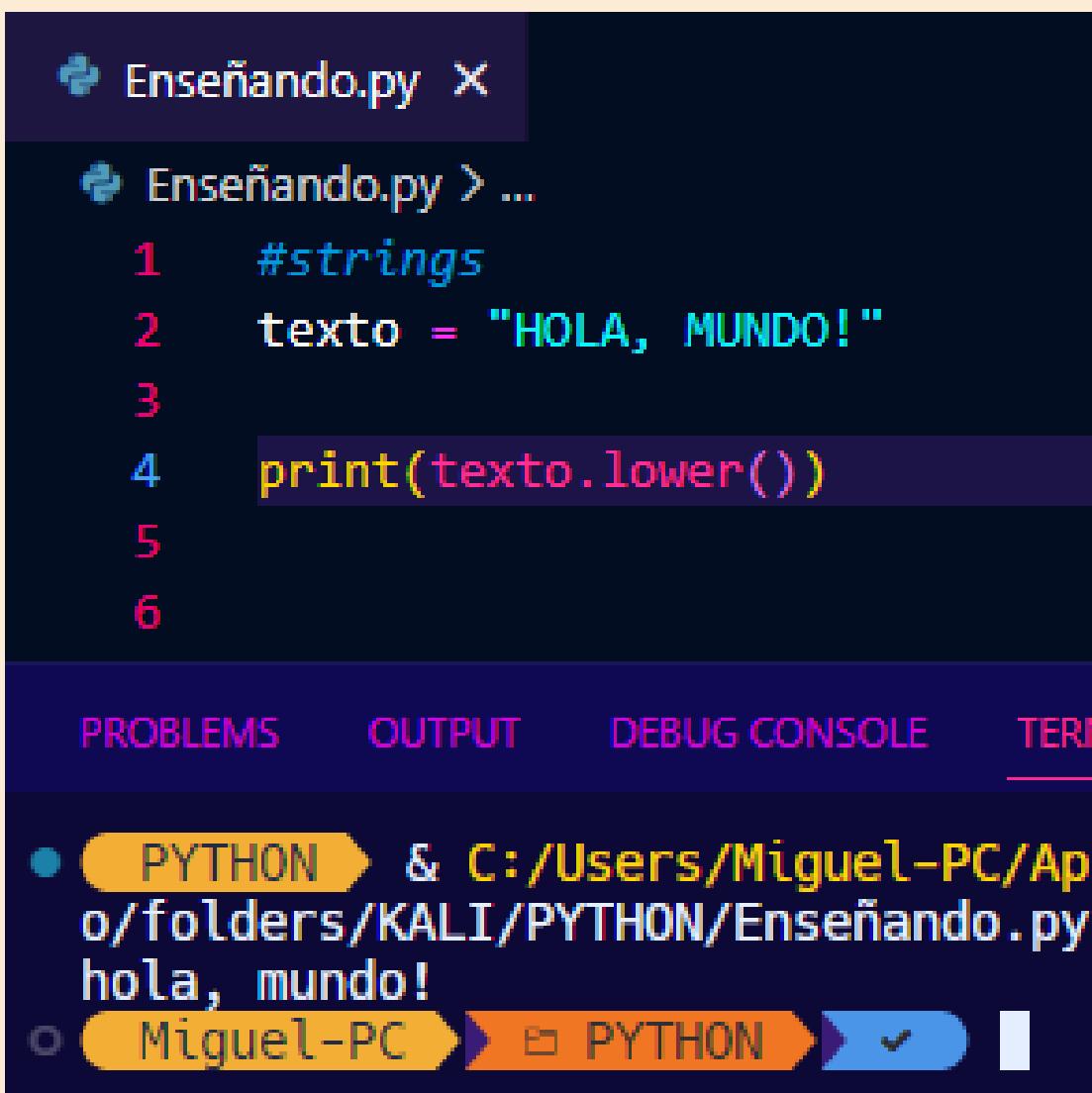


```
#strings
texto = "Hola, mundo!"
mayusculas = texto.upper()
print(mayusculas)
```

strings

Métodos de strings:

En Python, los strings son objetos y tienen varios métodos incorporados que te permiten realizar diversas operaciones y manipulaciones en los strings. Aquí tienes algunos métodos comunes de los strings en Python:
lower(): Convierte un string a minúsculas.



```
Enseñando.py > ...
#strings
texto = "HOLA, MUNDO!"
print(texto.lower())

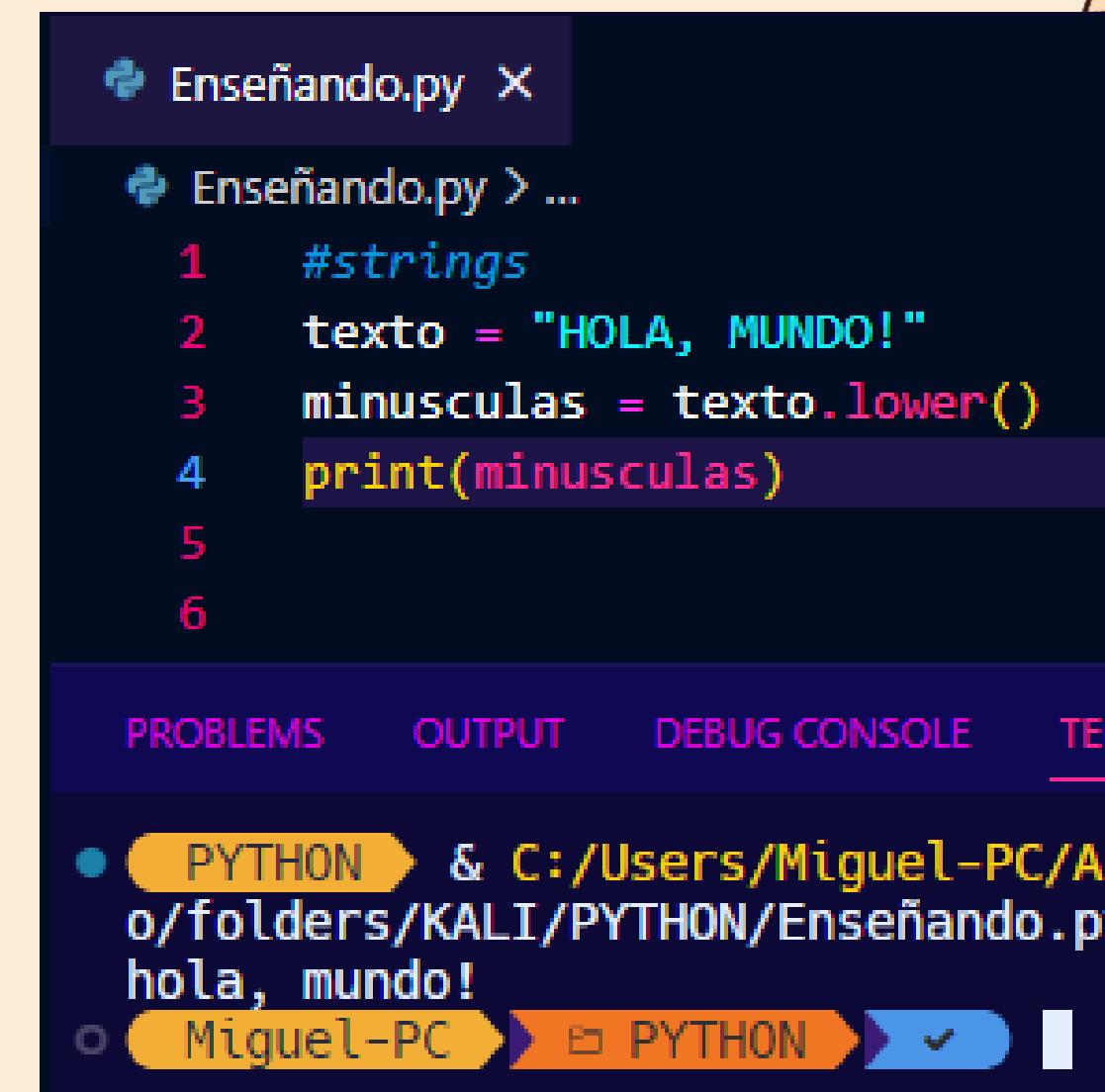
```

The screenshot shows a code editor window with a dark theme. At the top, there's a tab bar with 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERM'. Below the tabs, the code is displayed in a syntax-highlighted editor. The code itself is as follows:

```
1 #strings
2 texto = "HOLA, MUNDO!"
3
4 print(texto.lower())
5
6
```

At the bottom of the editor, there's a terminal window showing the output of the script. The terminal tab is also labeled 'TERM'. The output is:

```
PYTHON & C:/Users/Miguel-PC/AppData/folders/KALI/PYTHON/Enseñando.py
hola, mundo!
```



```
Enseñando.py > ...
#strings
texto = "HOLA, MUNDO!"
minusculas = texto.lower()
print(minusculas)

```

This screenshot shows the same code as the first one, but with a slight modification. Instead of printing the result directly, it stores the result of the `lower()` method in a variable named `minusculas`, and then prints that variable. The terminal output is identical to the first screenshot:

```
PYTHON & C:/Users/Miguel-PC/AppData/folders/KALI/PYTHON/Enseñando.py
hola, mundo!
```

strings

ejercicios :

Problema: Registro de gatitos

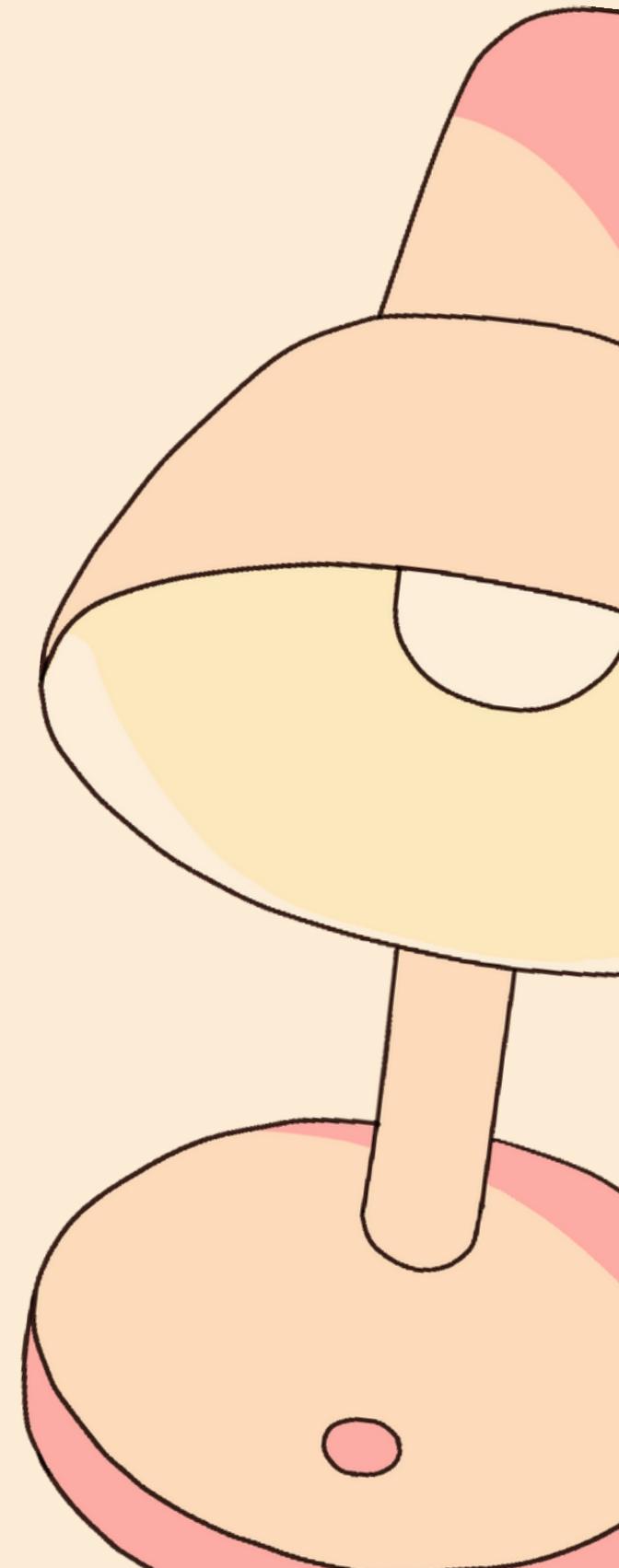
Supongamos que estás creando un programa para llevar un registro de los nombres de los gatitos en una tienda de mascotas. Tu objetivo es desarrollar un programa que permita realizar las siguientes acciones:

1. Definir una variable llamada `nombres_gatitos` y asignarle un valor inicial de una cadena vacía.
2. Solicitar al usuario que ingrese el nombre de un nuevo gatito y almacenarlo en la variable `nuevo_gatito`.
3. Actualizar la variable `nombres_gatitos` concatenando el nombre del nuevo gatito, separado por comas, utilizando el operador de concatenación `+`.
4. Calcular la longitud de la cadena `nombres_gatitos` utilizando la función `len()` y almacenarla en una variable llamada `cantidad_gatitos`.
5. Mostrar la cantidad de gatitos registrados utilizando la variable `cantidad_gatitos`.
6. Solicitar al usuario que ingrese un mensaje de saludo para los gatitos y utilizar el método de formateo de cadenas para agregar el mensaje al final de `nombres_gatitos`.

strings

ejercicios : solucion

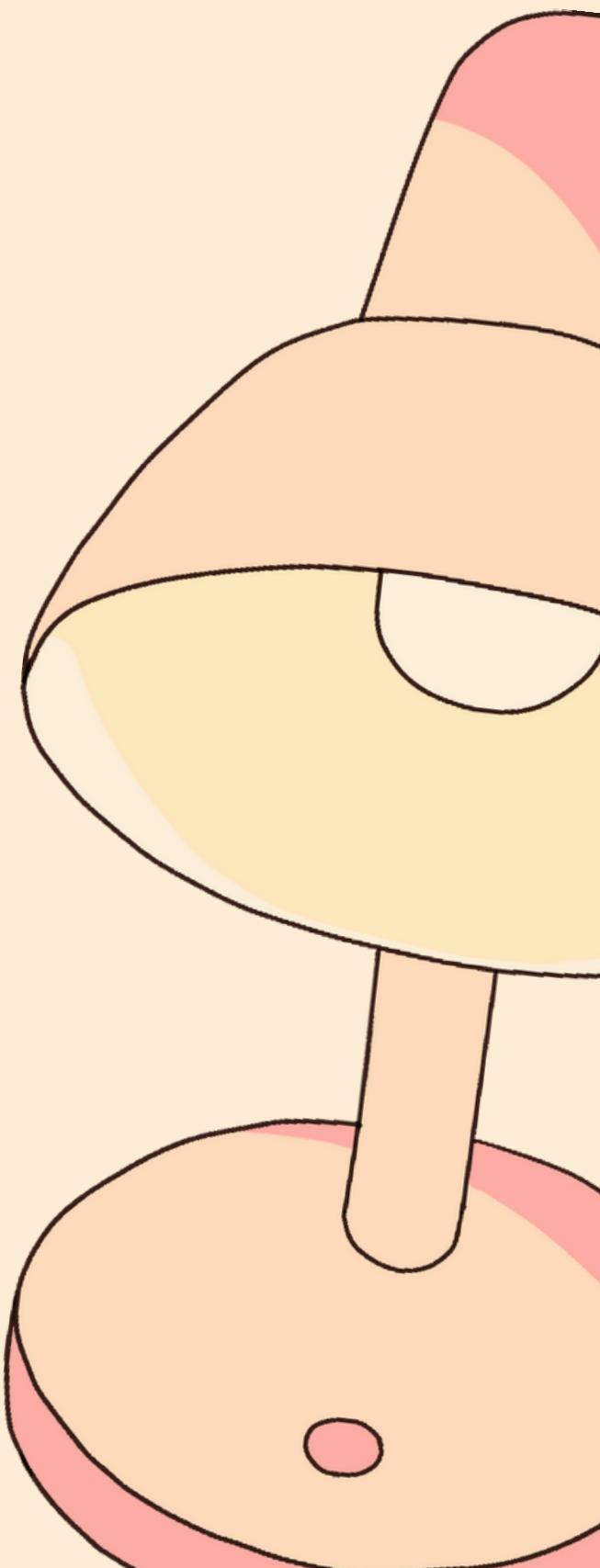
```
Enseñando.py •
Enseñando.py > ...
1 # Definir una variable para almacenar Los nombres de Los gatitos
2 nombres_gatitos = ""
3
4 # Solicitar al usuario que ingrese el nombre de un nuevo gatito
5 nuevo_gatito = input("Ingrese el nombre del nuevo gatito: ")
6
7 # Actualizar la variable nombres_gatitos concatenando el nuevo nombre
8 nombres_gatitos += nuevo_gatito
9
10 # Calcular la cantidad de gatitos registrados
11 cantidad_gatitos = len(nombres_gatitos)
12
13 # Mostrar la cantidad de gatitos registrados
14 print("Cantidad de gatitos registrados:", cantidad_gatitos)
15
16 # Solicitar al usuario un mensaje de saludo para Los gatitos
17 mensaje_saludo = input("Ingrese un mensaje de saludo para los gatitos: ")
18
19 # Agregar el mensaje de saludo al final de nombres_gatitos utilizando el formateo de cadenas
20 nombres_gatitos += ", " + mensaje_saludo
21
22 # Mostrar la lista de gatitos registrados
23 print("Lista de gatitos:", nombres_gatitos)
```



lista

Introducción a las listas:

Definición de listas: En Python, puedes definir una lista utilizando corchetes [] y separando los elementos por comas. Aquí tienes un ejemplo de cómo definir una lista:



```
Enseñando.py
Enseñando.py > ...
1 #Lista
2 frutas = ["manzana", "frutilla", "banana"]
```

En este ejemplo, hemos definido una lista llamada `frutas` que contiene tres elementos: "manzana", "frutilla", "banana". Los elementos están separados por comas y están encerrados entre corchetes.

lista

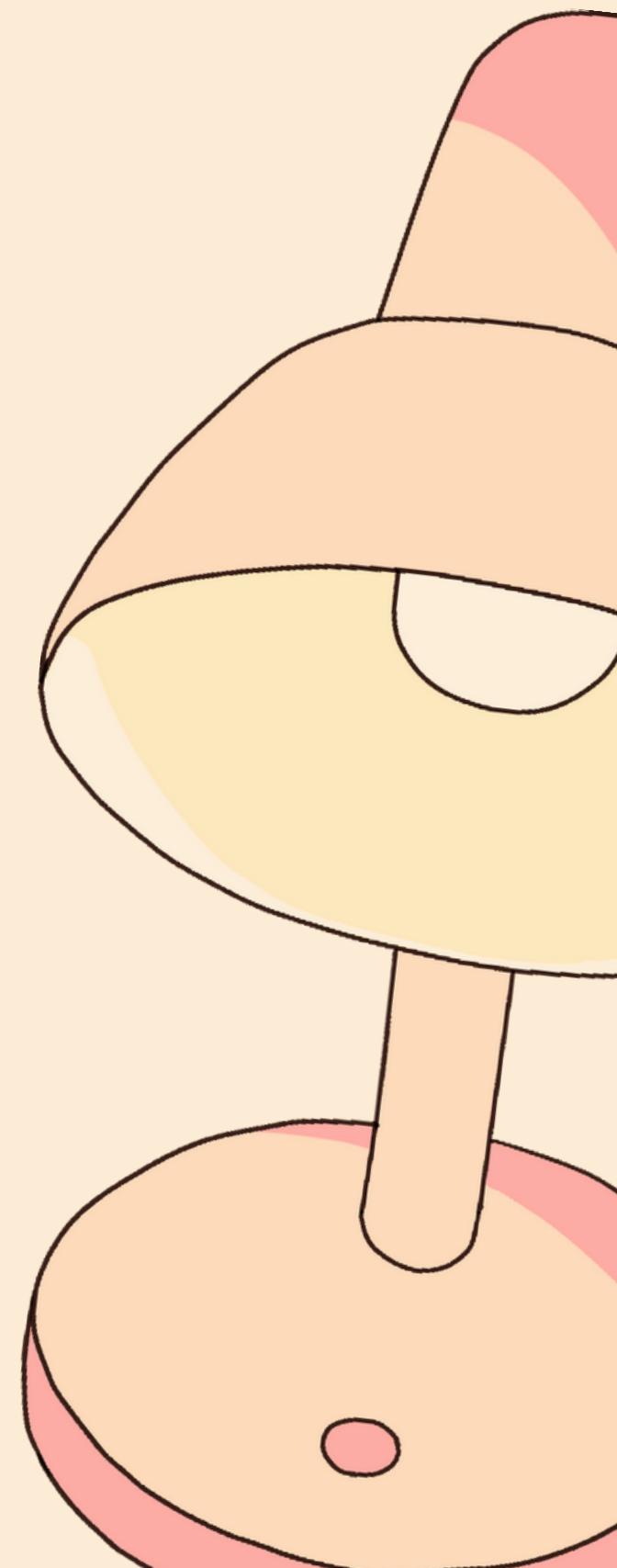
Acceso a elementos de una lista:

Índices y notación de índices:

Los elementos de una lista se indexan comenzando desde 0. El primer elemento tiene un índice de 0, el segundo tiene un índice de 1, y así sucesivamente.

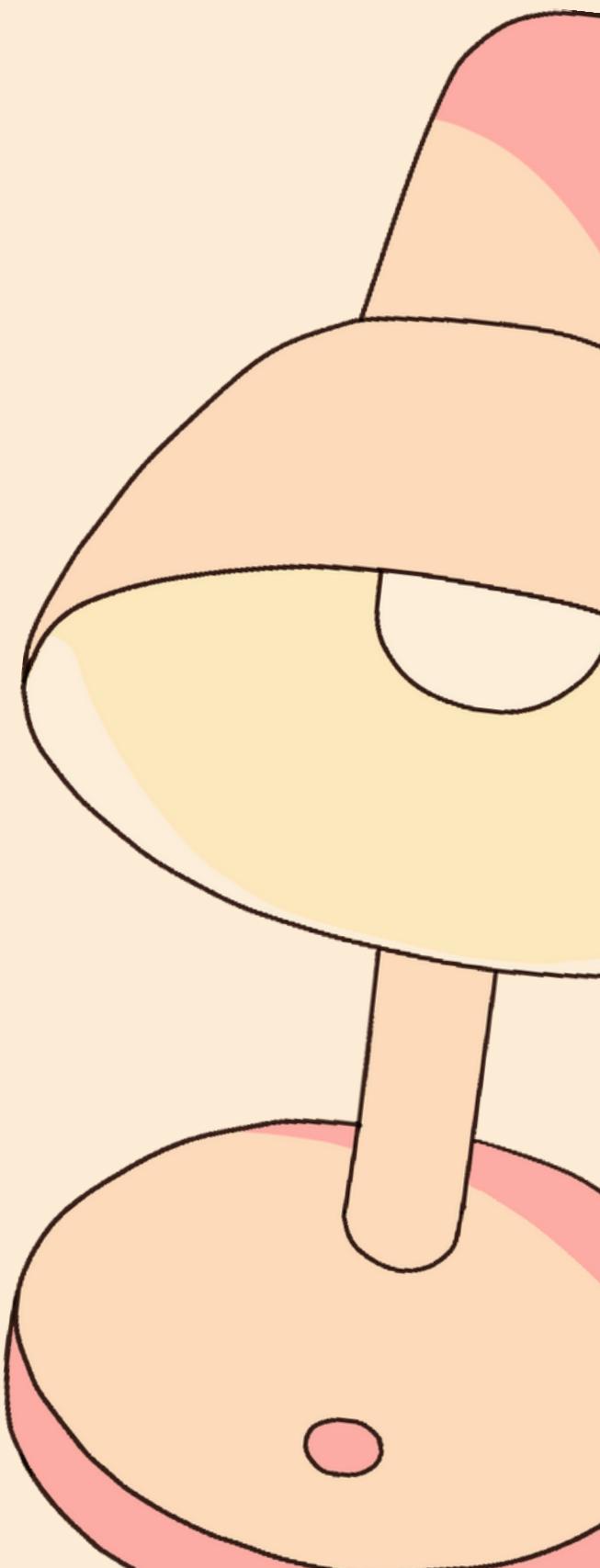
Puedes acceder a un elemento utilizando corchetes [] después del nombre de la lista y dentro de los corchetes colocas el índice del elemento que deseas acceder.

```
Enseñando.py • [0] [1] [2]
Enseñando.py > ...
1 #Lista
2 frutas = ["manzana", "frutilla", "banana"]
```



lista

Acceso a elementos de una lista:



```
Enseñando.py X
Enseñando.py > ...
2 frutas = ["manzana", "frutilla", "banana"]
3
4 primer_fruta = frutas[0]
5 print(primer_fruta)
6 segunda_fruta = frutas[1]
7 print(segunda_fruta)
8 tercera_fruta = frutas[2]
9 print(tercera_fruta)
10
```

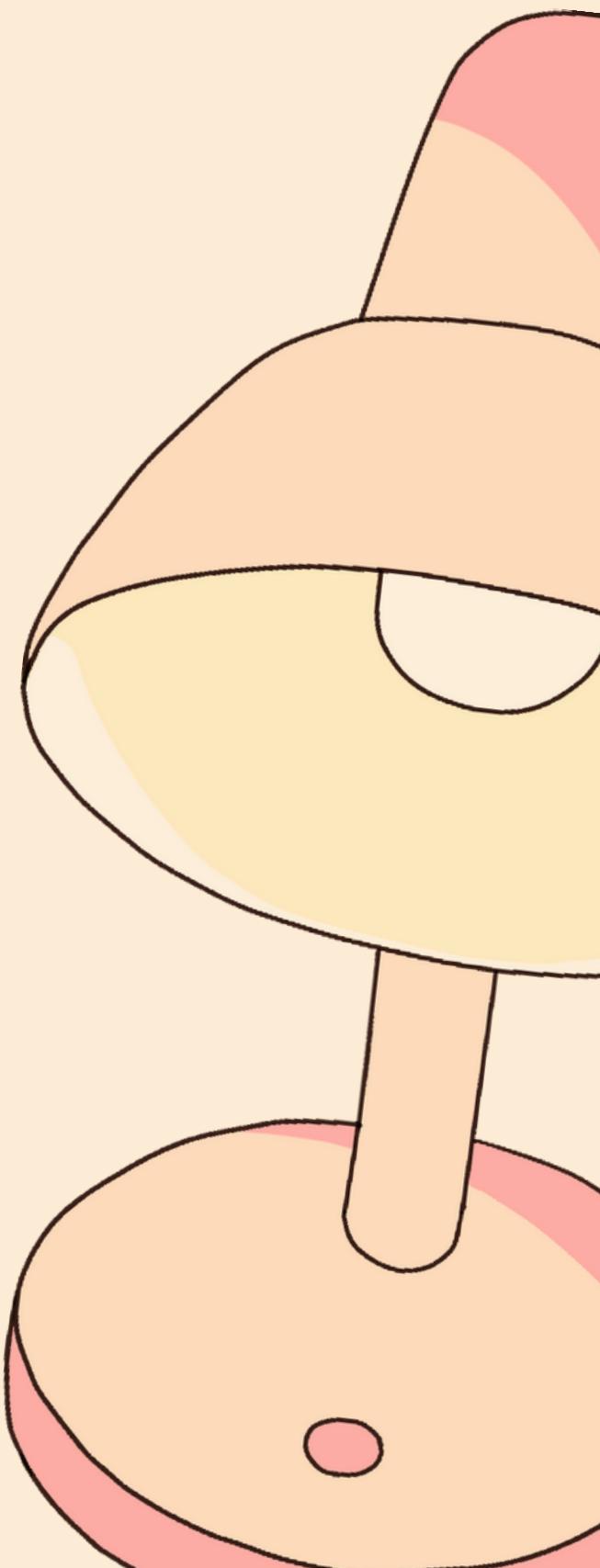
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local, o/folders/KALI/PYTHON/Enseñando.py
manzana
frutilla
banana

Miguel-PC PYTHON

lista

Acceso a elementos de una lista:



```
Enseñando.py X
Enseñando.py > ...
2 frutas = ["manzana", "frutilla", "banana"]
3
4 primer_fruta = frutas[0]
5 print(primer_fruta)
6 segunda_fruta = frutas[1]
7 print(segunda_fruta)
8 tercera_fruta = frutas[2]
9 print(tercera_fruta)
10
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local, o/folders/KALI/PYTHON/Enseñando.py
manzana
frutilla
banana

Miguel-PC PYTHON

lista

Acceso a elementos de una lista:

Acceso a elementos mediante índices negativos:

Puedes utilizar índices negativos para acceder a los elementos desde el final de la lista. El índice `-1` se refiere al último elemento, `-2` al penúltimo elemento, y así sucesivamente.

Frutas = `[""manzana", "frutilla", "banana"]`

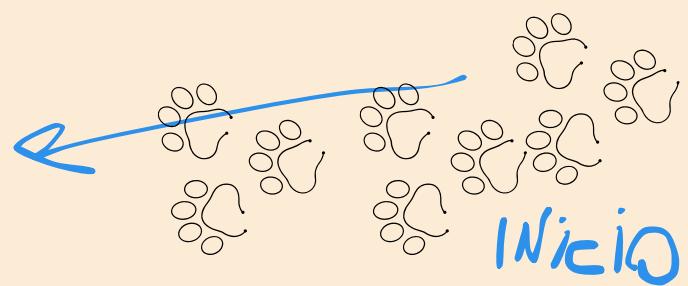
`[0]` `[1]` `[2]`

Cuando nosotros llamamos al índice `[-1]` lo que estamos haciendo es movernos para atrás, de esta forma:

imaginemos que estamos en el índice `[0]` y nos queremos mover al `[-1]` como seria eso?

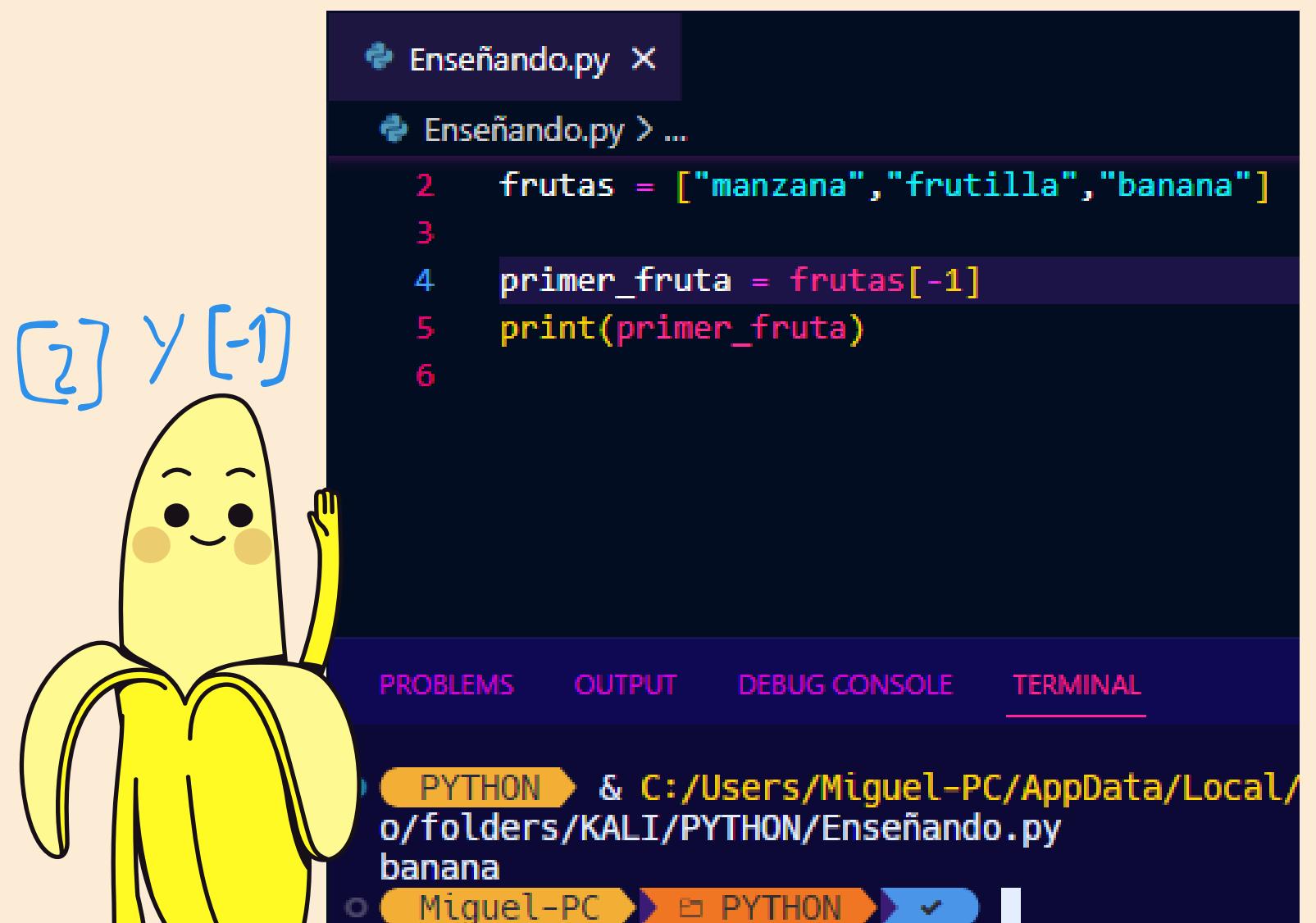
Frutas = `[""manzana", "frutilla", "banana"]`

`[0]` `[1]` `[2]`



lista

Acceso a elementos de una lista:



```
Enseñando.py X
Enseñando.py > ...
2   frutas = ["manzana", "frutilla", "banana"]
3
4   primer_fruta = frutas[-1]
5   print(primer_fruta)
6
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/folders/KALI/PYTHON/Enseñando.py
banana

Miguel-PC ▶ PYTHON ▶ □



lista

Acceso a elementos de una lista:

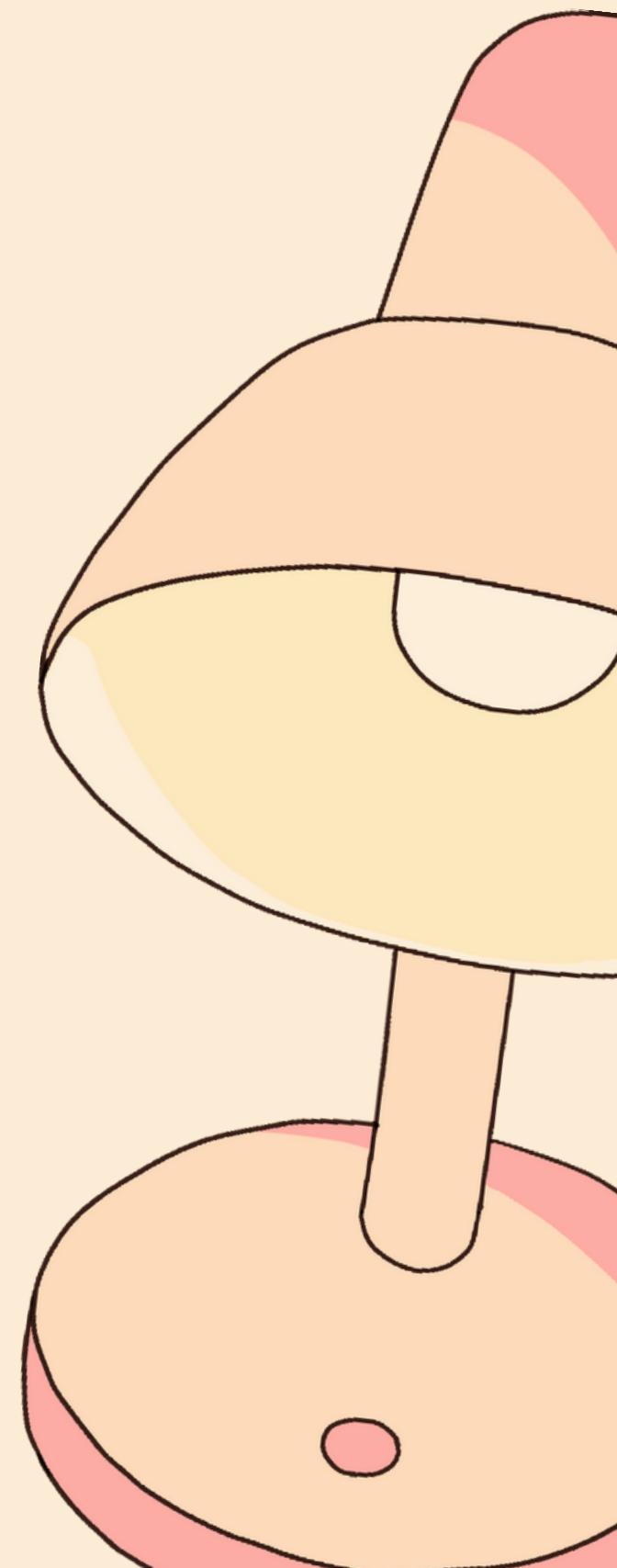


Enseñando.py

```
2 frutas = ["manzana", "frutilla", "banana"]
3
4 primer_fruta = frutas[0]
5 print(primer_fruta)
6 segunda_fruta = frutas[-1]
7 print(segunda_fruta)
8 tercera_fruta = frutas[-2]
9 print(tercera_fruta)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local
o/folders/KALI/PYTHON/Enseñando.py
manzana
banana
frutilla
- Miguel-PC ▶ PYTHON ✓

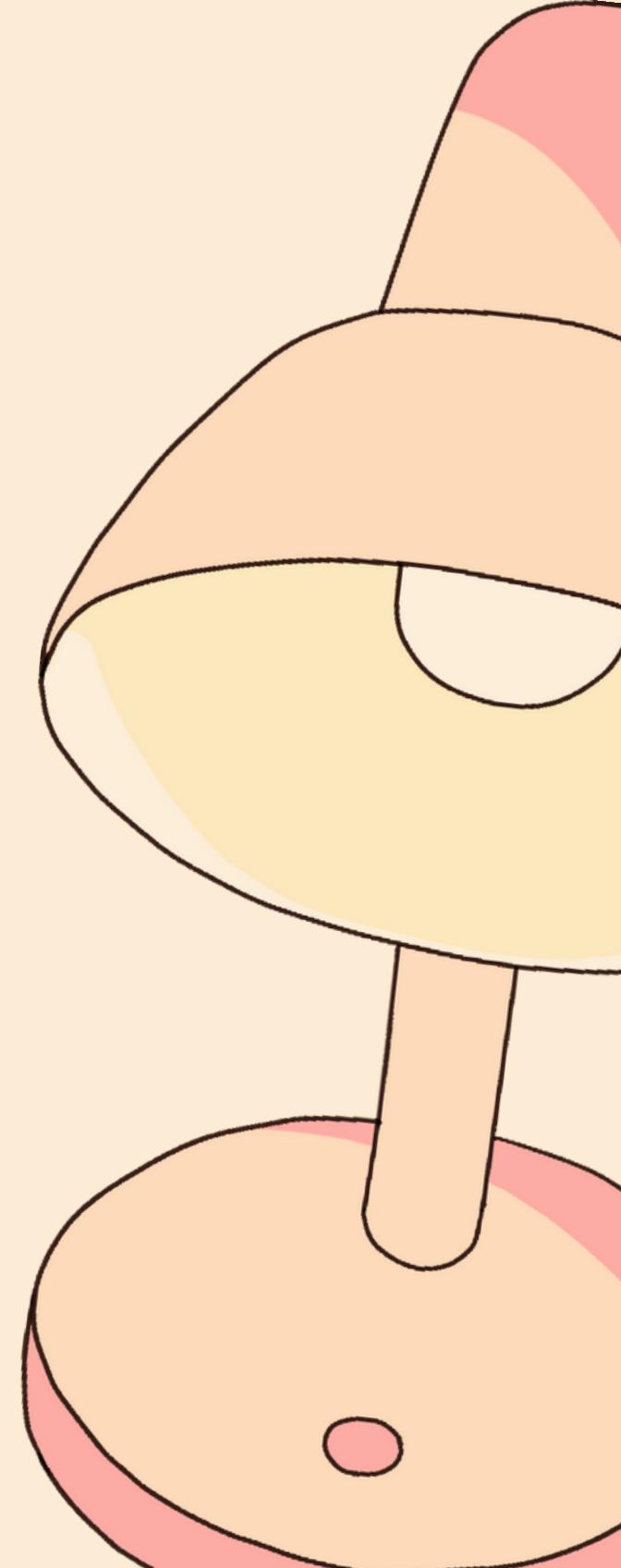


lista

Slicing (obtener subconjuntos de una lista):

Puedes utilizar el slicing para obtener un subconjunto de una lista, especificando un rango de índices.

El rango se especifica utilizando la notación [inicio:fin], donde inicio es el índice del primer elemento que se incluirá y fin es el índice del elemento justo después del último elemento que se incluirá.



```
Enseñando.py
Enseñando.py > ...
1 #lista
2 frutas = ["manzana", "frutilla", "banana"]
3
4 # Obtener los primeros dos elementos
5 primeros_dos = frutas[0:2]
6 print(primeros_dos)
7
8 # Obtener los elementos desde el segundo hasta el último
9 desde_segundo = frutas[1:]
10 print(desde_segundo)
11
12
```

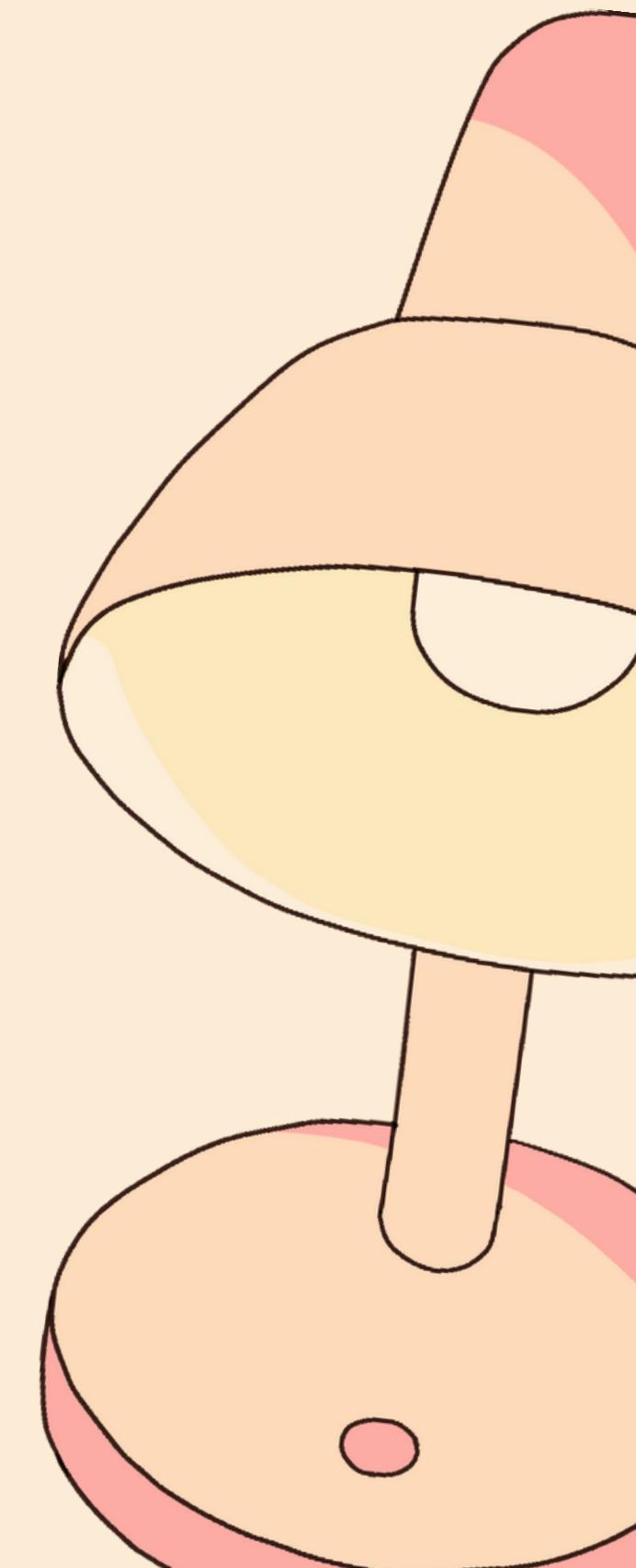
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/Python, o/folders/KALI/PYTHON/Enseñando.py
['manzana', 'frutilla']
['frutilla', 'banana']
- Miguel-PC □ PYTHON ▶ ✓

lista

Asignación de nuevos valores:

Puedes asignar un nuevo valor a un elemento específico de una lista utilizando la asignación directa con el operador de asignación `=`.



```
Enseñando.py X
Enseñando.py > ...
1 #lista
2 frutas = ["manzana", "plátano", "naranja", "piña"]
3
4 # Asignar un nuevo valor al segundo elemento
5 frutas[1] = "uva"
6 print(frutas)
7
```

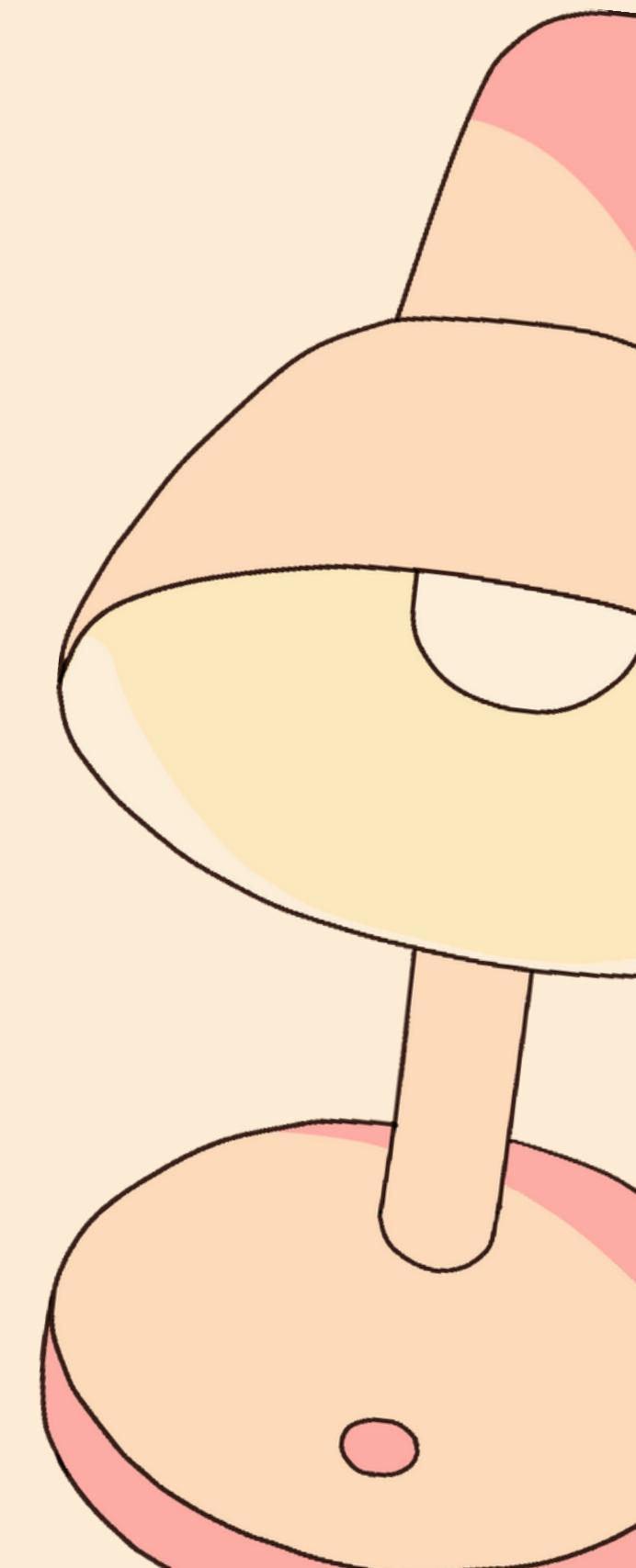
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/o/folders/KALI/PYTHON/Enseñando.py
['manzana', 'uva', 'naranja', 'piña']
- Miguel-PC PYTHON

lista

Modificación de elementos mediante slicing:

Puedes utilizar la técnica de slicing para modificar varios elementos de una lista a la vez. Por ejemplo:



```
Enseñando.py X
Enseñando.py > ...
1 #lista
2 frutas = ["manzana", "frutilla", "banana", "uvas", "naranjas", "kiwis"]
3
4 frutas[1:4] = ["mandarinas", "pomelos", "sandias"]
5 print(frutas)
6
7
```

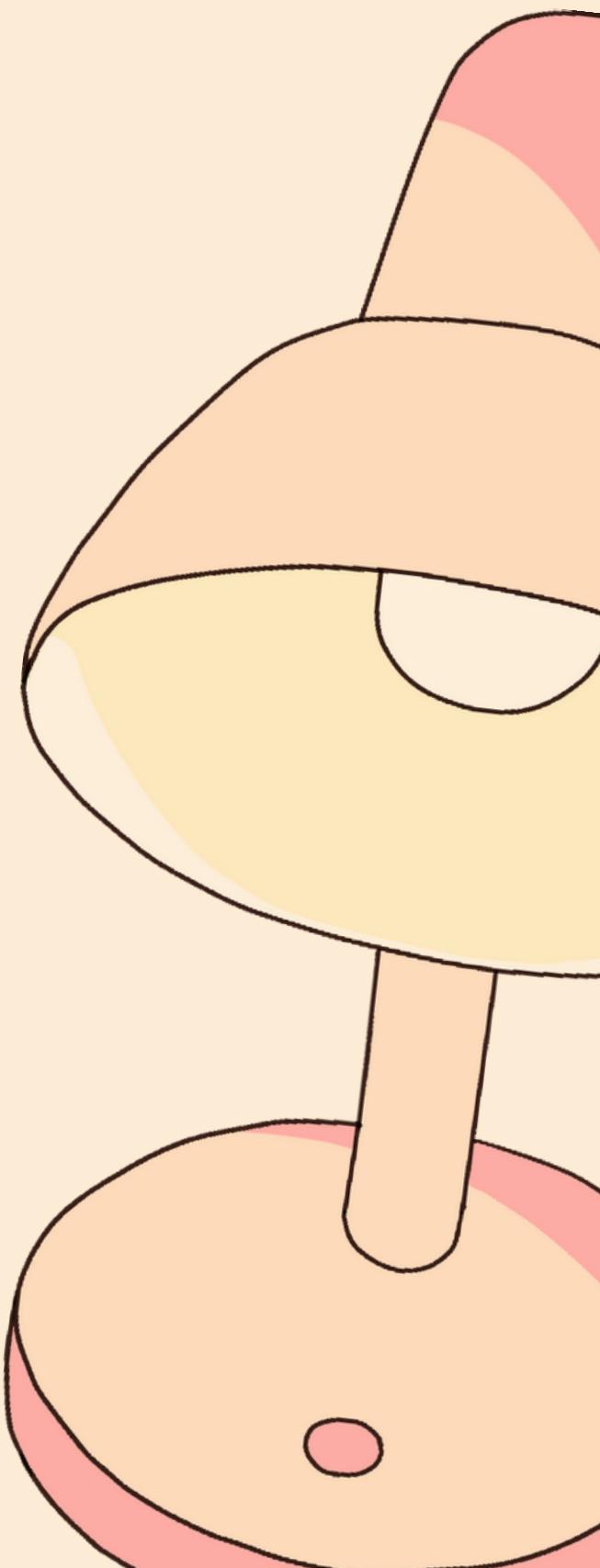
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python311
o/folders/KALI/PYTHON/Enseñando.py
['manzana', 'mandarinas', 'pomelos', 'sandias', 'naranjas', 'kiwis']
- Miguel-PC ▶ PYTHON ✓

lista

Métodos de las listas:

append(): agregar elementos al final de la lista



```
Enseñando.py X
Enseñando.py > ...
1 #Lista
2 frutas = ["manzana", "frutilla", "banana", ]
3 frutas.append("uvas")
4 print(frutas)
5
6
```

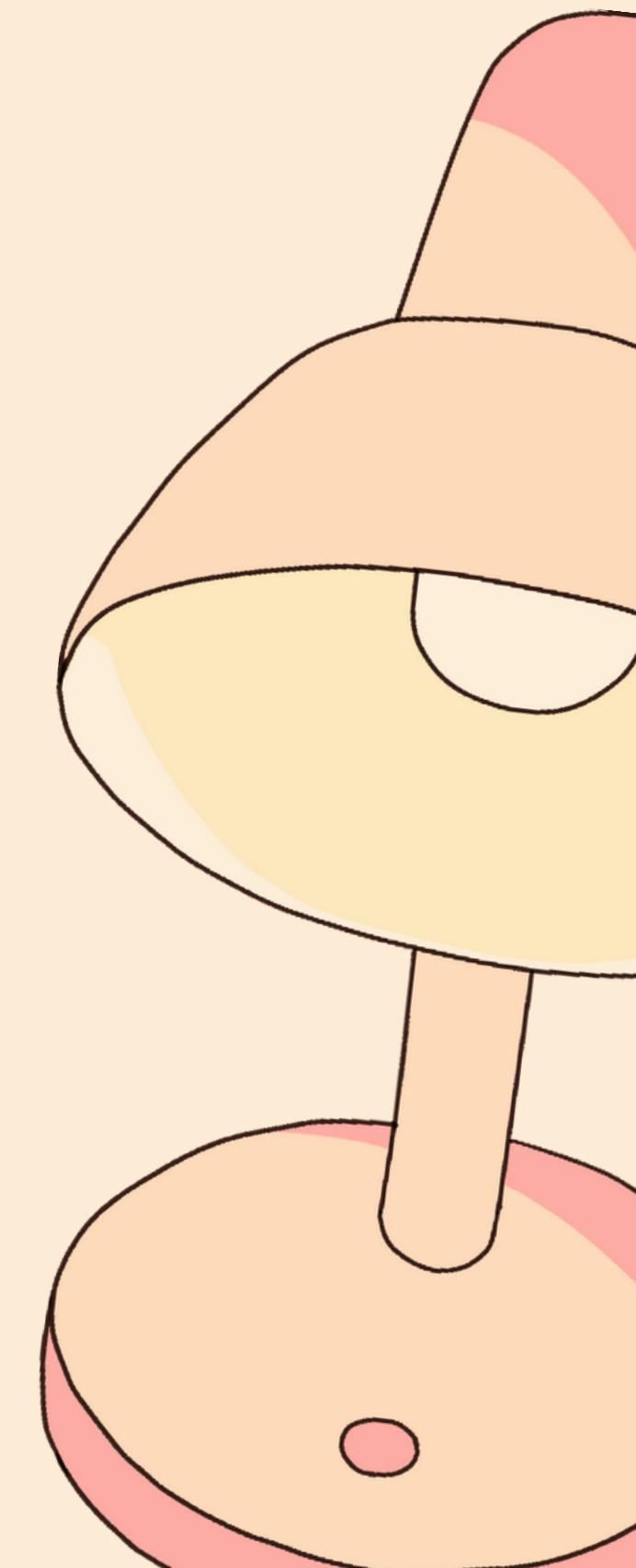
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/Po/folders/KALI/PYTHON/Enseñando.py
['manzana', 'frutilla', 'banana', 'uvas']
- Miguel-PC ➔ PYTHON ➔ ✓

lista

Métodos de las listas:

insert(): insertar elementos en una posición específica



```
Enseñando.py X
Enseñando.py > ...
1 #Lista
2 frutas = ["manzana", "frutilla", "banana", ]
3 frutas.insert(1, "uvas")
4 print(frutas)
5
6
```

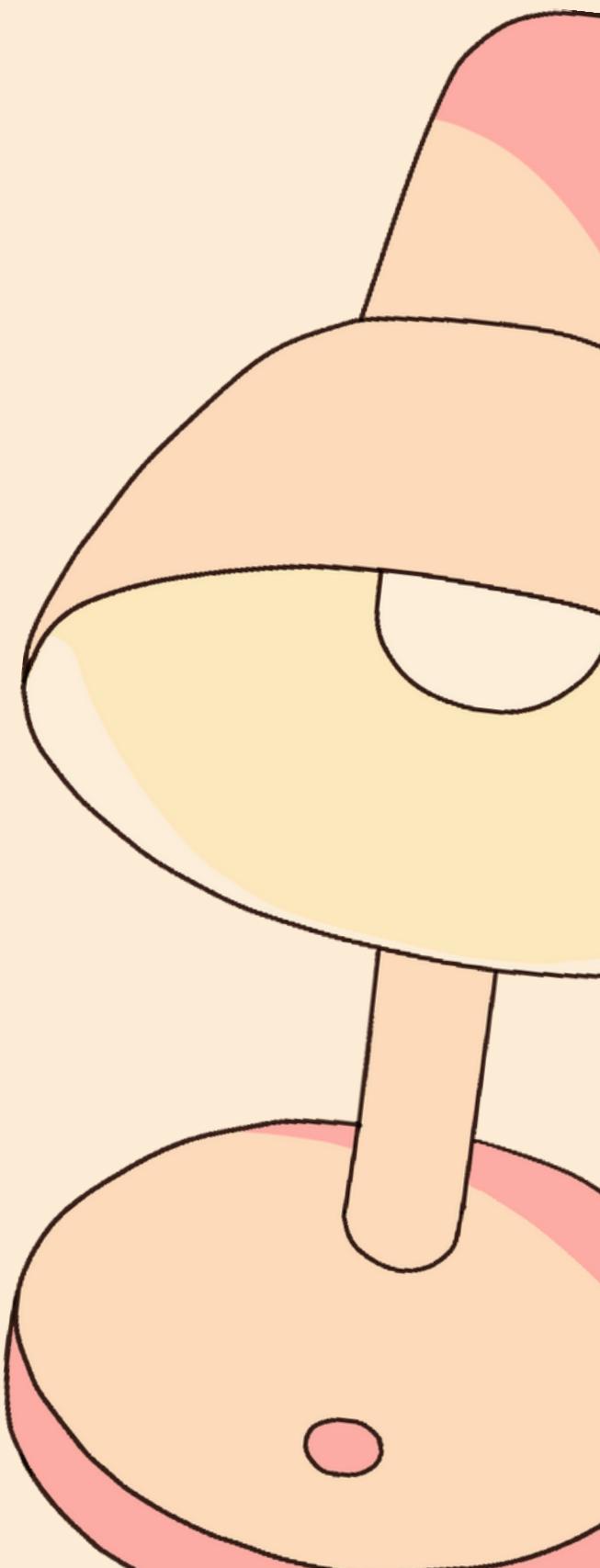
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/folders/KALI/PYTHON/Enseñando.py['manzana', 'uvas', 'frutilla', 'banana']
- Miguel-PC ▶ □ PYTHON ▶ ✓

lista

Métodos de las listas:

remove(): eliminar el primer elemento que coincide con un valor dado



A screenshot of a Python code editor showing a file named "Enseñando.py". The code defines a list of fruits and removes the first occurrence of "frutilla" using the `remove()` method. The output terminal shows the original list and the modified list after the removal.

```
1 #lista
2 frutas = ["manzana","frutilla","banana",]
3 frutas.remove("frutilla")
4 print(frutas)
5
6
```

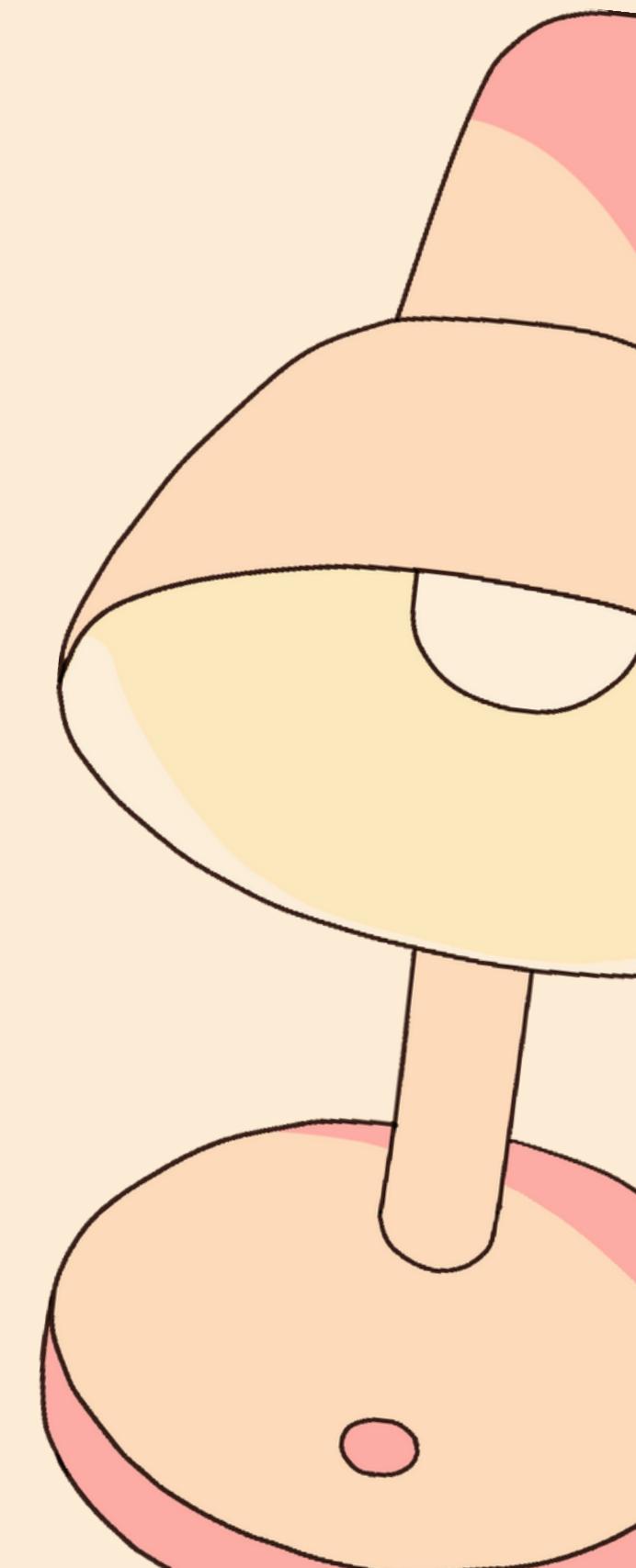
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/Folder/folders/KALI/PYTHON/Enseñando.py
['manzana', 'banana']
- Miguel-PC □ PYTHON ▶ ✓

lista

Métodos de las listas:

pop(): eliminar y obtener el último elemento de la lista



```
#lista
frutas = ["manzana", "frutilla", "banana"]
frutas.pop()
print(frutas)
```

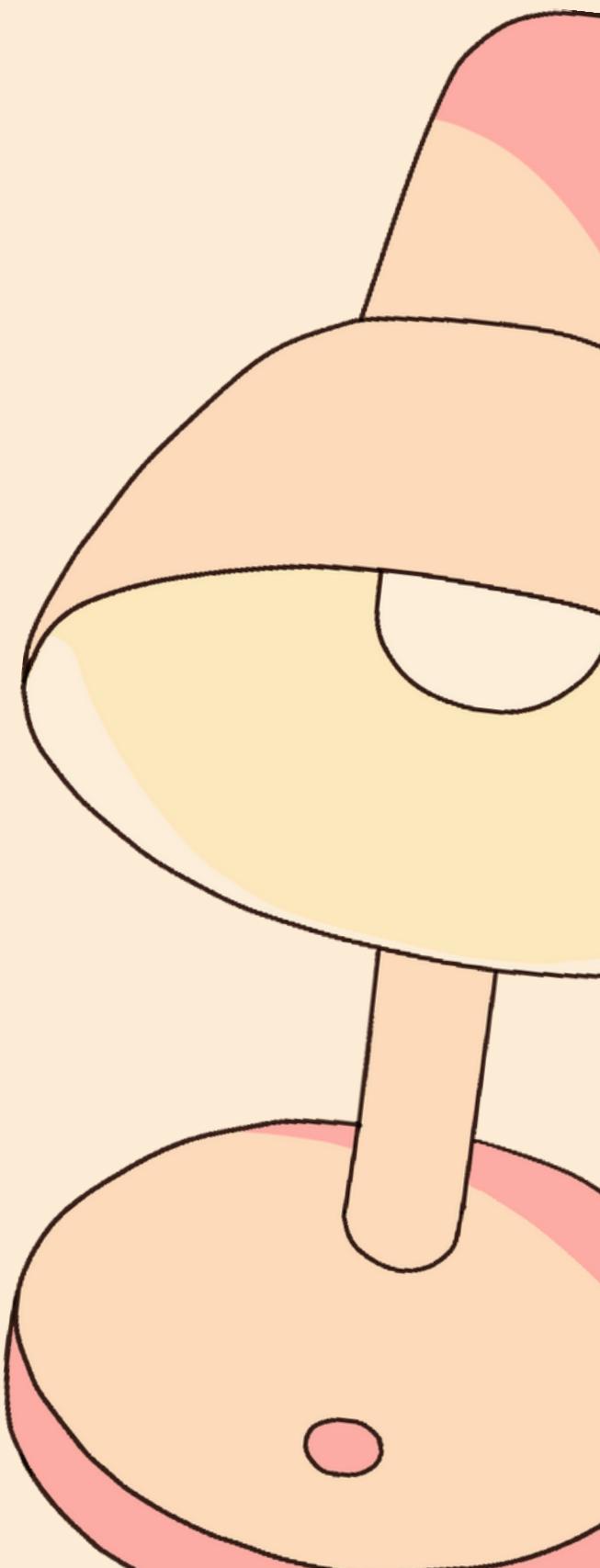
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/Po/folders/KALI/PYTHON/Enseñando.py
['manzana', 'frutilla']
- Miguel-PC ▶ PYTHON

lista

Métodos de las listas:

index(): encontrar el índice de un elemento



A screenshot of a Python code editor showing a file named "Enseñando.py". The code defines a list of fruits and uses the `index()` method to find the index of "frutilla". The output tab shows the result "1".

```
#Lista
frutas = ["manzana","frutilla","banana"]
indice = frutas.index("frutilla")
print(indice)
```

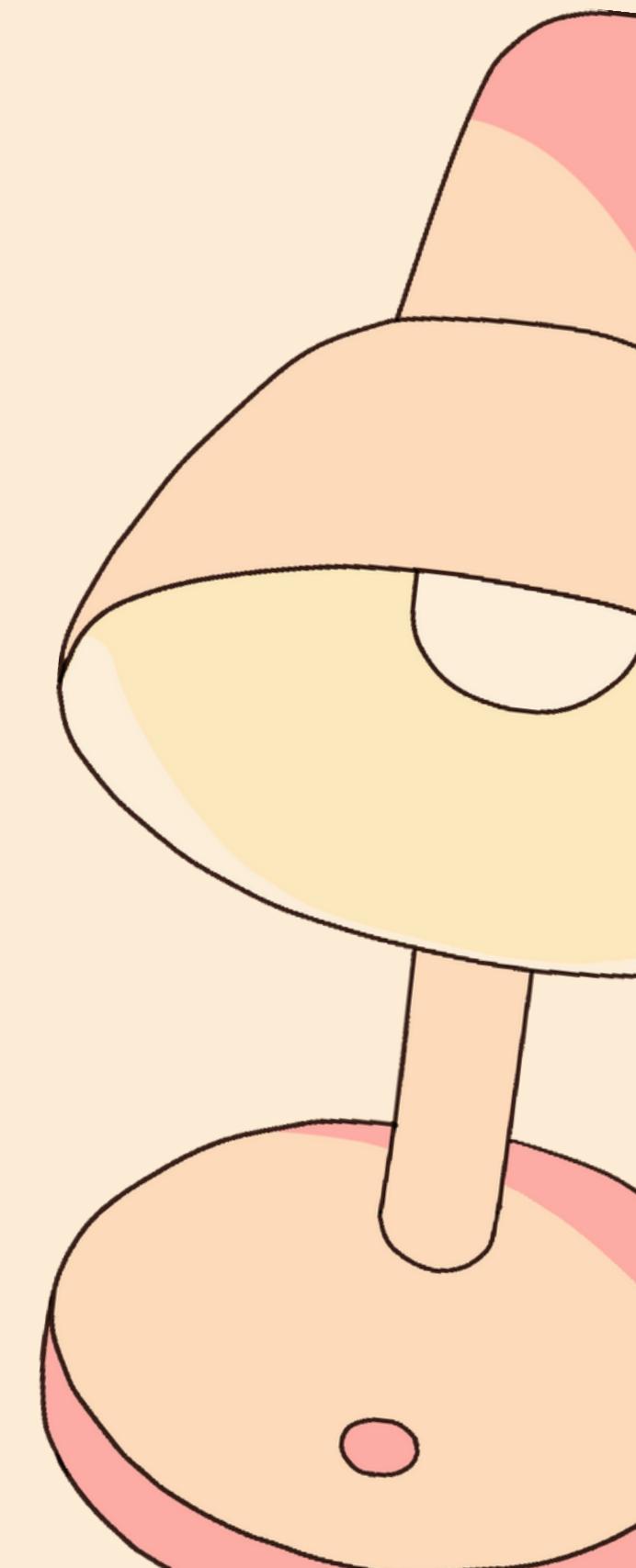
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/folders/KALI/PYTHON/Enseñando.py
1
- Miguel-PC □ PYTHON □ □

lista

Métodos de las listas:

count(): contar las ocurrencias de un elemento



```
Enseñando.py X
Enseñando.py > ...
1 #lista
2 frutas = ["manzana", "frutilla", "banana", "manzana", "manzana"]
3 contarrepetidas = frutas.count("manzana")
4 print(contarrepetidas)
5
6
```

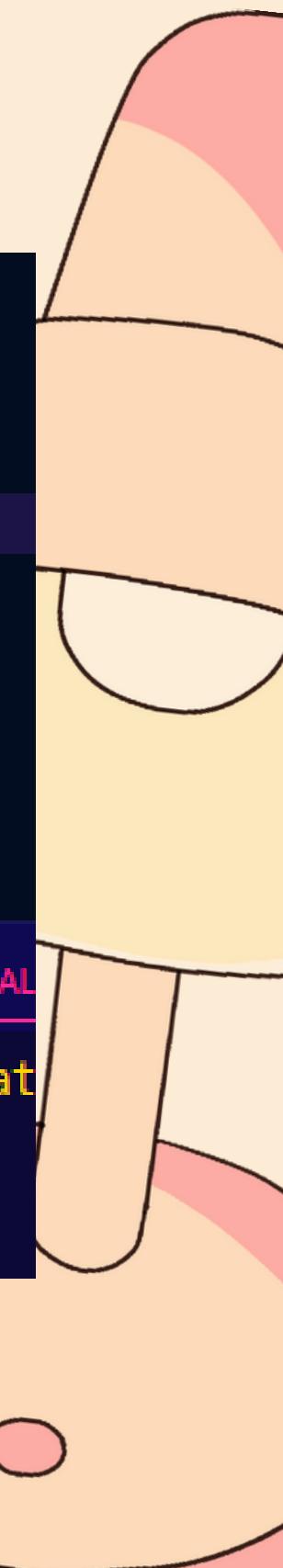
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python39/folders/KALI/PYTHON/Enseñando.py
3
- Miguel-PC PYTHON

lista

Métodos de las listas:

sort(): ordenar la lista

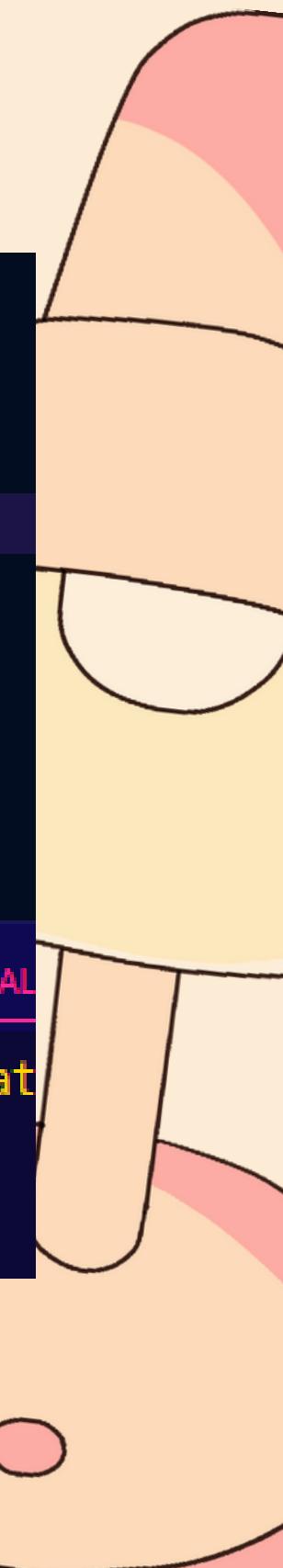


```
Enseñando.py X
Enseñando.py > ...
1 #Lista
2 frutas = ["manzana", "frutilla", "banana", "manzana", "manzana"]
3 frutas.sort()
4 print(frutas)
5
6
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python37/folders/KALI/PYTHON/Enseñando.py
['banana', 'frutilla', 'manzana', 'manzana', 'manzana']
- Miguel-PC ▶ PYTHON

orden alfabetico



```
Enseñando.py X
Enseñando.py > ...
1 #Lista
2 frutas = [3, 3, 5, 6, 2, 3, 2, 1, 1]
3 frutas.sort()
4 print(frutas)
5
6
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

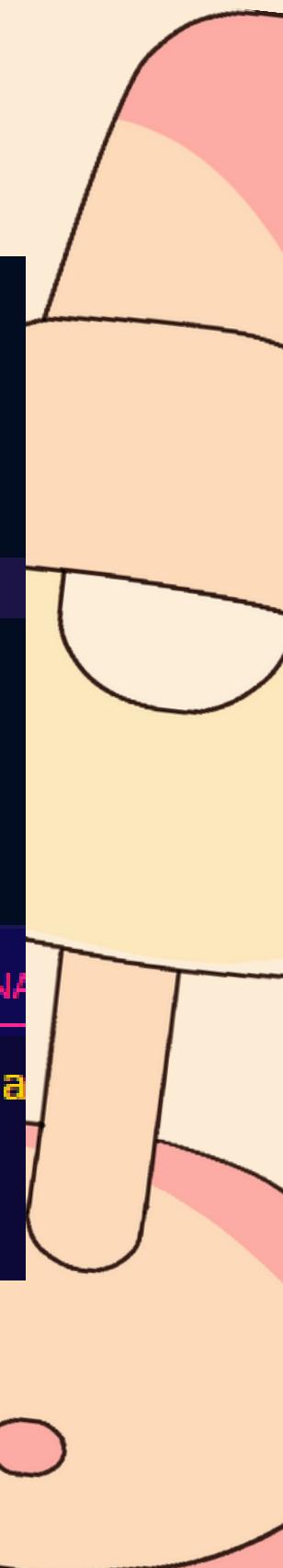
- PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python37/folders/KALI/PYTHON/Enseñando.py
[1, 1, 2, 2, 3, 3, 3, 5, 6]
- Miguel-PC ▶ PYTHON

orden numerico ascendente

lista

Métodos de las listas:

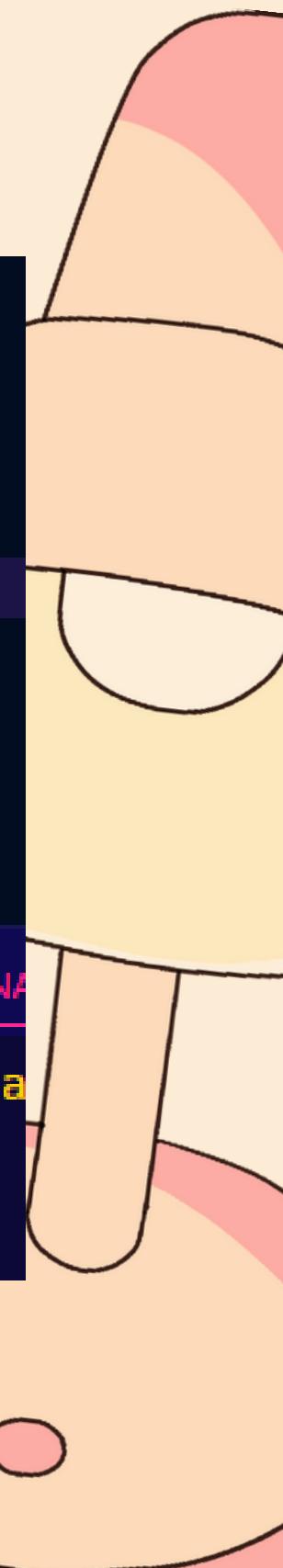
reverse(): invertir el orden de la lista



```
Enseñando.py X
Enseñando.py > ...
1 #lista
2 frutas = ["manzana","frutilla","banana","manzana","manzana"]
3 frutas.sort(reverse=True)
4 print(frutas)
5
6
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python36/folders/KALI/PYTHON/Enseñando.py
['manzana', 'manzana', 'manzana', 'frutilla', 'banana']
- Miguel-PC ▶ PYTHON ▶ ✓



```
Enseñando.py X
Enseñando.py > ...
1 #Lista
2 frutas = [3,3,5,6,2,3,2,1,1]
3 frutas.sort(reverse=True)
4 print(frutas)
5
6
```

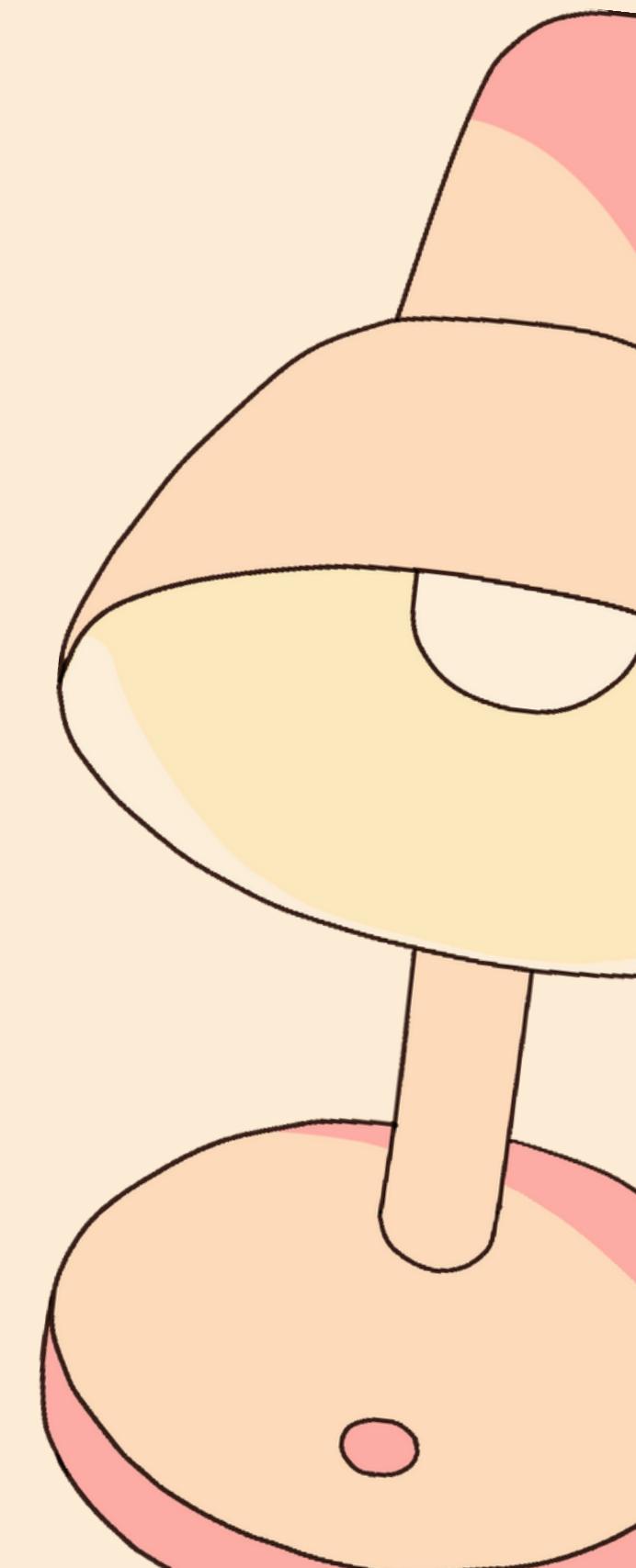
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python36/folders/KALI/PYTHON/Enseñando.py
[6, 5, 3, 3, 3, 2, 2, 1, 1]
- Miguel-PC ▶ PYTHON ▶ ✓

lista

Métodos de las listas:

`len()`: obtener la longitud de la lista



A screenshot of a Python code editor showing a file named `Enseñando.py`. The code defines a list of fruits and prints its length. The terminal below shows the output of the program.

```
#lista
frutas = ["manzana", "frutilla", "banana", "manzana", "manzana"]
print(len(frutas))
```

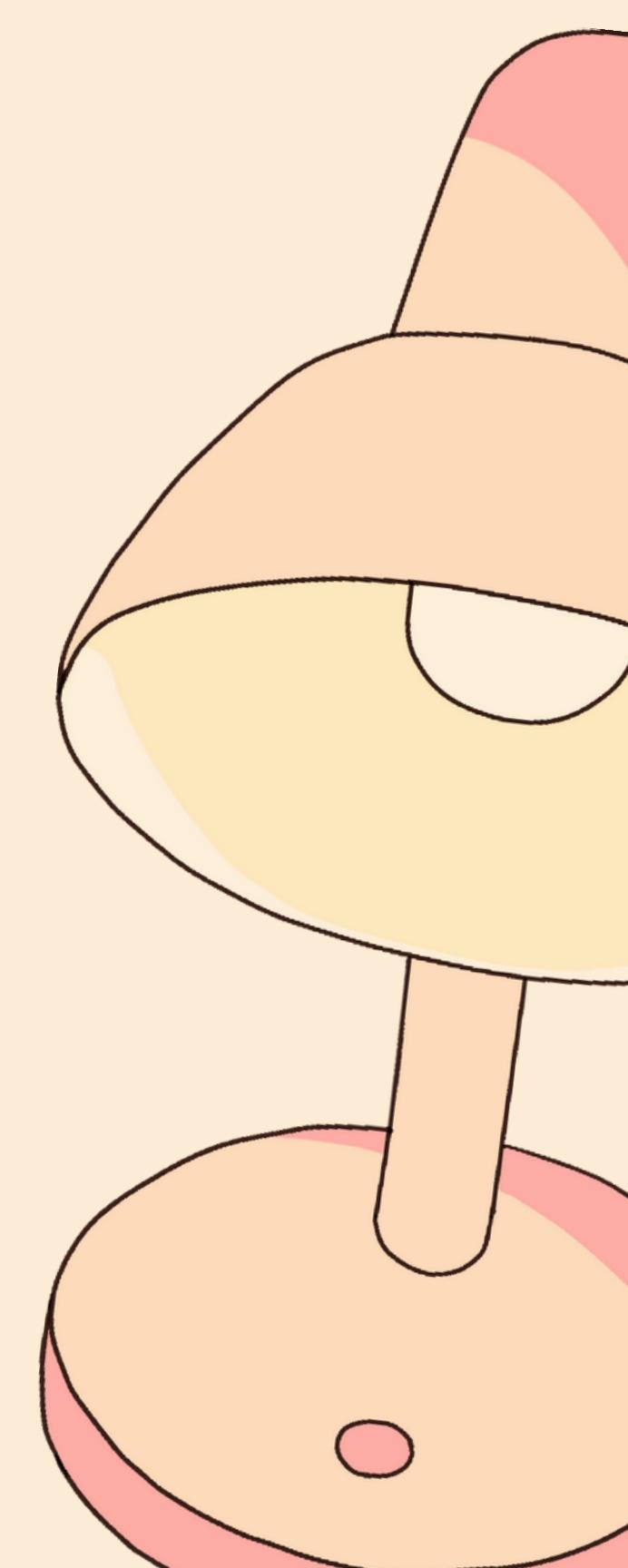
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON > & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python37/folders/KALI/PYTHON/Enseñando.py
5
- Miguel-PC > □ PYTHON ➤ ↴ ↵

lista

Operaciones y manipulaciones comunes de listas

Concatenación de listas: Puedes utilizar el operador de concatenación (+) para unir dos listas en una sola. Por ejemplo:



```
Enseñando.py X
Enseñando.py > ...
1 #Lista
2 lista1 = [1, 2, 3]
3 lista2 = [4, 5, 6]
4 lista_resultante = lista1 + lista2
5 print(lista_resultante)
6
7
8
```

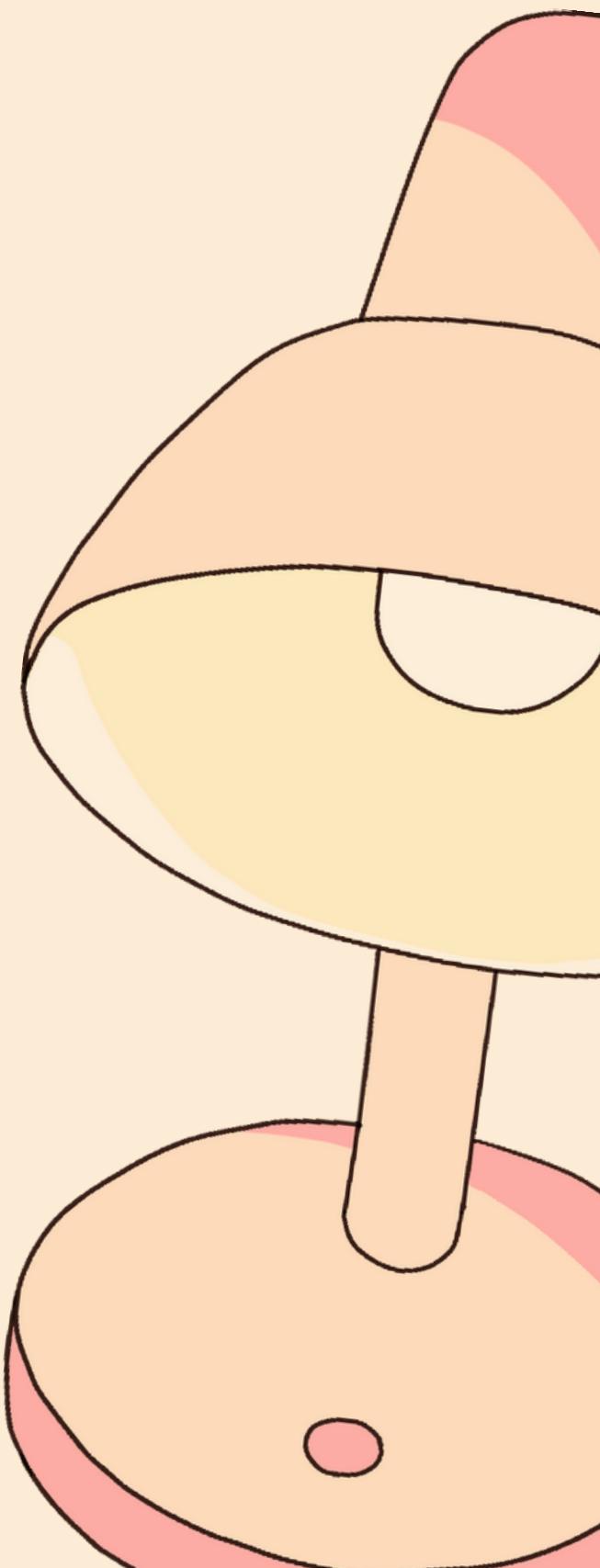
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
[1, 2, 3, 4, 5, 6]
- Miguel-PC PYTHON

lista

Operaciones y manipulaciones comunes de listas

Repetición de listas: Puedes utilizar el operador de repetición (*) para repetir una lista múltiples veces. Por ejemplo:



```
Enseñando.py > ...
Enseñando.py > ...
1 #lista
2 lista = [1, 2, 3]
3 lista_repetida = lista * 2
4 print(lista_repetida)
5
6
```

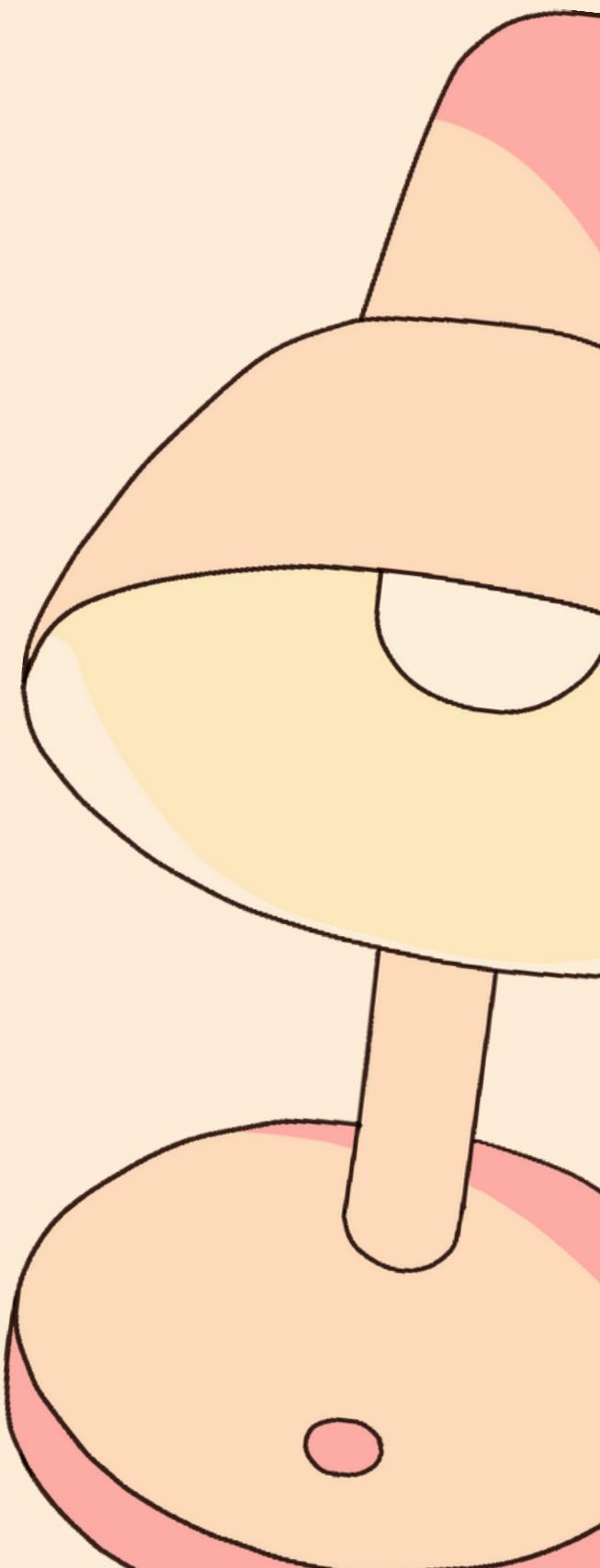
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
[1, 2, 3, 1, 2, 3]
- Miguel-PC PYTHON

lista

Operaciones y manipulaciones comunes de listas

Comprobación de pertenencia a una lista: Puedes utilizar el operador `in` para verificar si un elemento está presente en una lista. Devuelve `True` si el elemento está en la lista y `False` en caso contrario. Por ejemplo:



```
Enseñando.py X
Enseñando.py > ...
1 #Lista
2 lista = [1, 2, 3]
3 print(2 in lista) # Salida: True
4 print(4 in lista) # Salida: False
5
6
7
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py
True
False

Miguel-PC PYTHON ✓

lista

Listas anidadas:

Creación y manipulación de listas que contienen otras listas :Puedes crear una lista que contenga otras listas como elementos. Por ejemplo:

python

 Copy code

```
lista_anidada = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

En este caso, lista_anidada es una lista que contiene tres sublistas.

Acceso a elementos: Puedes acceder a los elementos de las listas anidadas utilizando la notación de índices múltiples. Por ejemplo:

python

 Copy code

```
lista_anidada = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
elemento = lista_anidada[0][1]
print(elemento) # Salida: 2
```

En este caso, accedemos al elemento con índice [0][1], que se encuentra en la primera sublista y en la segunda posición.

lista

Listas anidadas:

Manipulación de listas anidadas: Puedes modificar, agregar o eliminar elementos de las listas anidadas de la misma manera que lo harías con listas simples. Por ejemplo:

python

Copy code

```
lista_anidada = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
lista_anidada[1][0] = 10  
lista_anidada.append([11, 12, 13])  
lista_anidada.remove([7, 8, 9])
```

En este caso, modificamos el elemento [1][0] cambiándolo a 10, agregamos una nueva sublistas al final con append(), y eliminamos la sublistas [7, 8, 9] con remove().

```
1 #lista  
2 lista_anidada = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
3 lista_anidada[1][0] = 10  
4 lista_anidada.append([11, 12, 13])  
5 lista_anidada.remove([7, 8, 9])  
6 print(lista_anidada)  
7  
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

● PYTHON > & C:/Users/Miguel-PC/AppData/Local/Programs/
o/folders/KALI/PYTHON/Enseñando.py
[[1, 2, 3], [10, 5, 6], [11, 12, 13]]
○ Miguel-PC > PYTHON > ▾

tuplas

Introducción a las tuplas:

las tuplas no se pueden modificar una vez creadas, lo que las hace útiles para almacenar datos que no deben cambiar. Pueden contener elementos de diferentes tipos,

Sintaxis de las tuplas: Para definir una tupla, se utilizan paréntesis () y los elementos se separan por comas. Por ejemplo:

```
python  
  
tupla = (1, 2, "Hola", True)
```

En este caso, hemos creado una tupla llamada tupla que contiene cuatro elementos: el número 1, el número 2, la cadena "Hola" y el valor booleano True.

También es posible crear una tupla sin utilizar paréntesis, simplemente separando los elementos por comas. Por ejemplo:

```
python  
  
tupla = 1, 2, "Hola", True
```

El resultado es el mismo que en el ejemplo anterior

tuplas

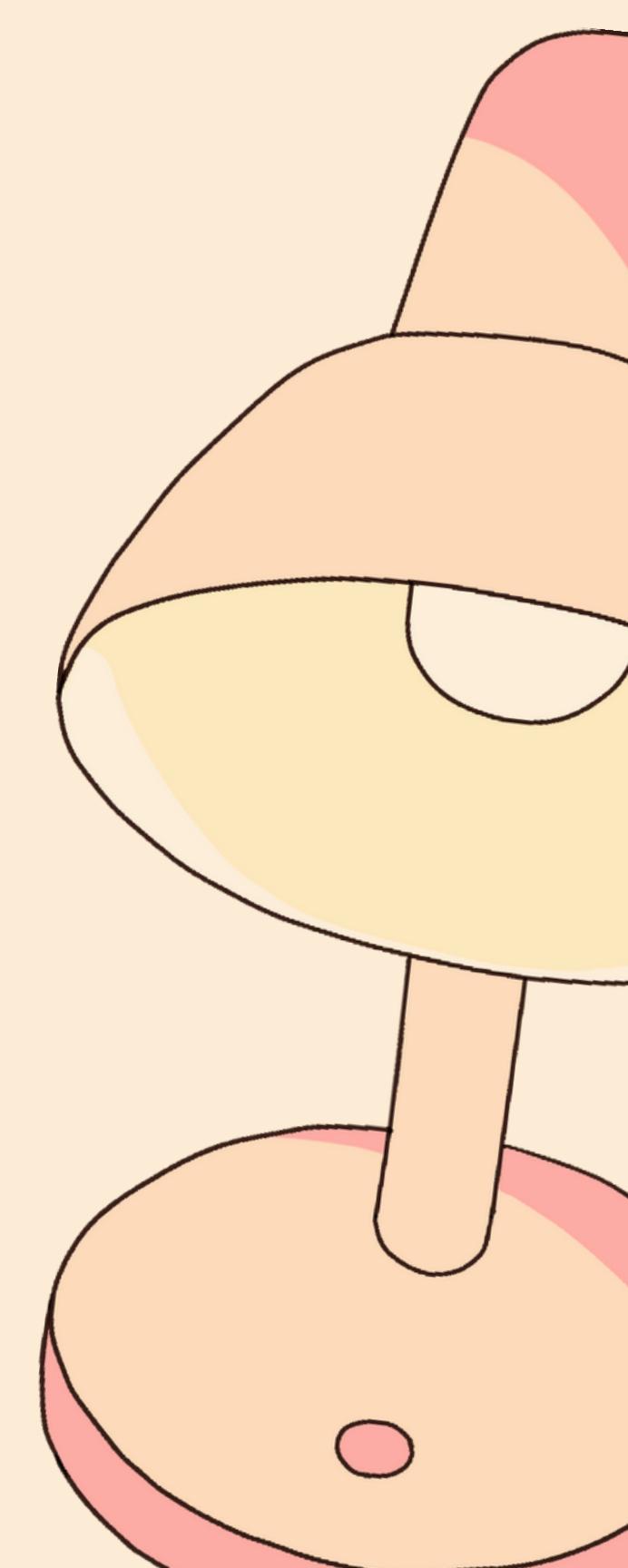
Acceso a elementos de una tupla

Índices y notación de índices: Los elementos de una tupla se indexan utilizando números enteros que representan la posición del elemento en la tupla. La notación de índices se realiza utilizando corchetes [] después del nombre de la tupla. El índice del primer elemento es 0, el segundo es 1, y así sucesivamente.

Acceso a elementos individuales: Puedes acceder a un elemento específico de una tupla utilizando su índice. Por ejemplo, si tienes una tupla llamada tupla y quieres acceder al tercer elemento, puedes hacerlo de la siguiente manera:

```
python

tupla = ("manzana", "banana", "naranja")
elemento = tupla[2] # Accede al tercer elemento (índice 2)
print(elemento) # Imprime: naranja
```



A screenshot of a Python code editor showing a file named 'Enseñando.py'. The code is identical to the one above, demonstrating tuple indexing. The code editor interface includes tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. In the 'TERMINAL' tab, the output shows the execution of the script and the printed value 'naranja'.

```
Enseñando.py X
Enseñando.py > ...
1 #tuplas
2 tupla = "manzana", "banana", "naranja"
3 elemento = tupla[2] # Accede al tercer elemento (índice 2)
4 print(elemento) # Imprime: naranja
5
6

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
• PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python37/folders/KALI/PYTHON/Enseñando.py
naranja
○ Miguel-PC ▶ PYTHON ▶ ✓
```

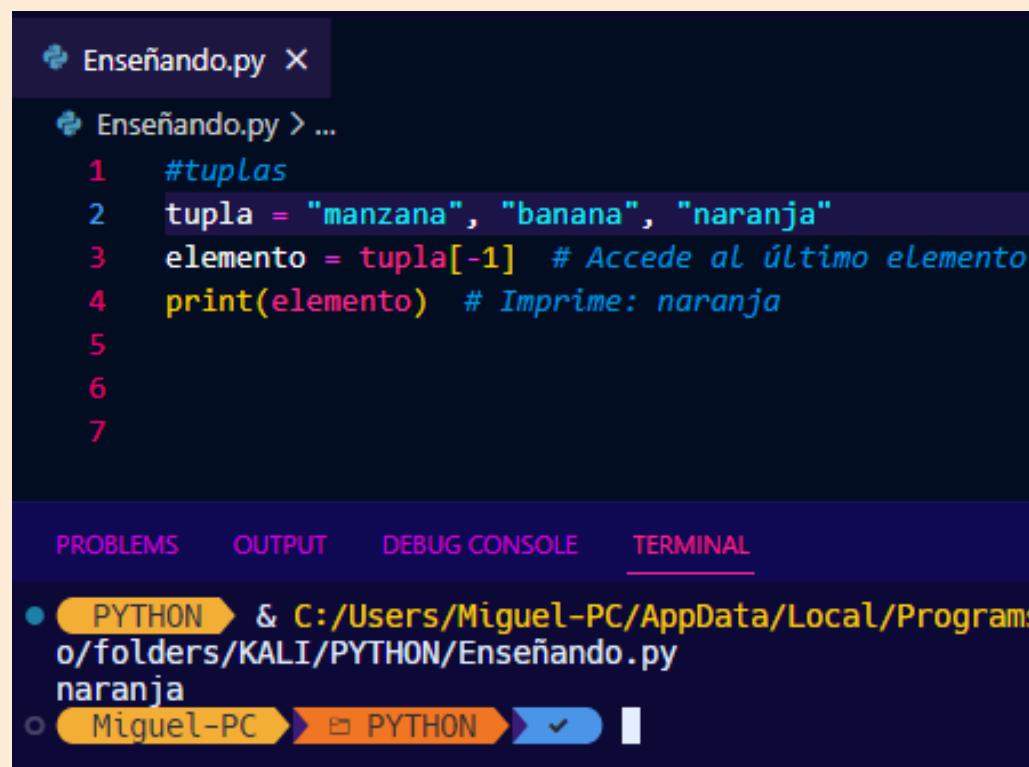
tuplas

Acceso a elementos de una tupla

Acceso a elementos mediante índices negativos: También es posible acceder a elementos de una tupla utilizando índices negativos. En este caso, el índice -1 hace referencia al último elemento, el -2 al penúltimo, y así sucesivamente. Por ejemplo:

```
python

tupla = ("manzana", "banana", "naranja")
elemento = tupla[-1] # Accede al último elemento
print(elemento) # Imprime: naranja
```



```
Enseñando.py X
Enseñando.py > ...
1 #tuplas
2 tupla = "manzana", "banana", "naranja"
3 elemento = tupla[-1] # Accede al último elemento
4 print(elemento) # Imprime: naranja
5
6
7

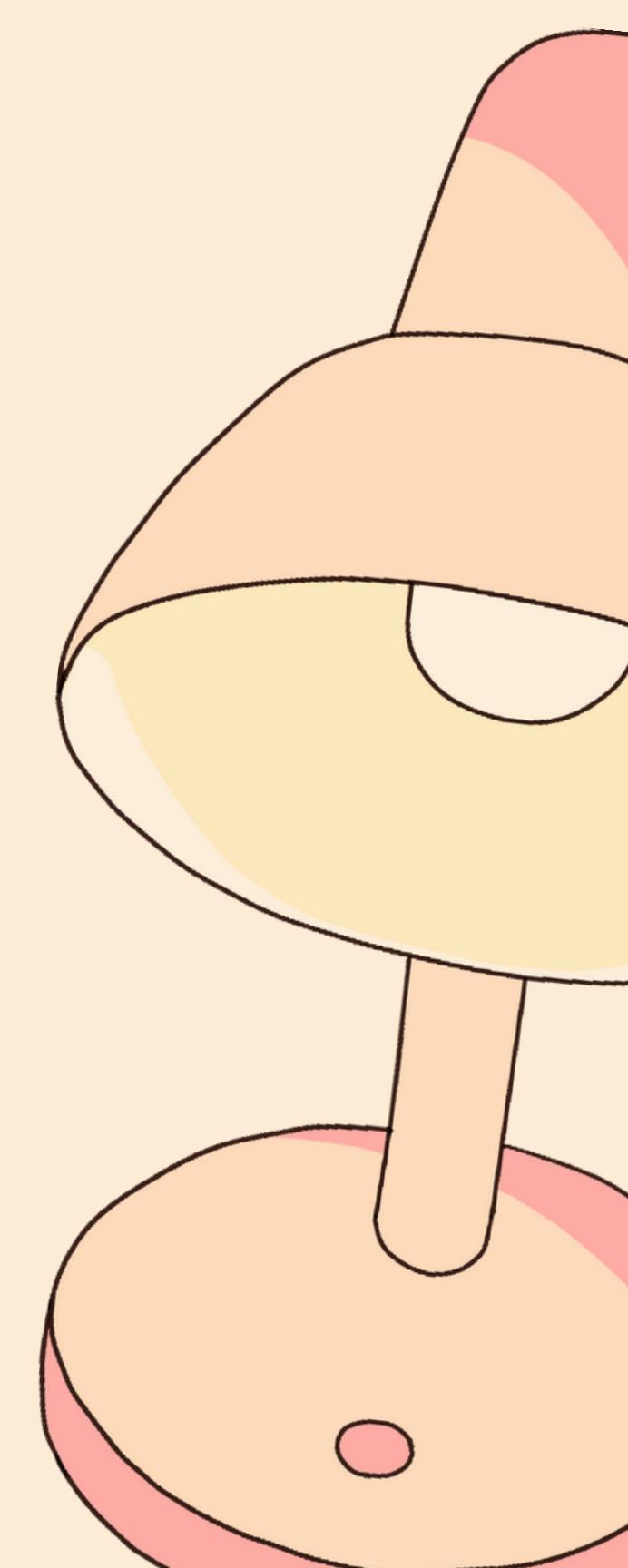
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/o/folders/KALI/PYTHON/Enseñando.py
naranja
○ Miguel-PC ▶ PYTHON ▶
```

Acceso a elementos mediante índices negativos: También es posible acceder a elementos de una tupla utilizando índices negativos. En este caso, el índice -1 hace referencia al último elemento, el -2 al penúltimo, y así sucesivamente. Por ejemplo:

tuplas

Acceso a elementos de una tupla

Slicing (obtener subconjuntos de una tupla): El slicing se utiliza para obtener un subconjunto de elementos de una tupla. Puedes especificar un rango de índices utilizando la notación `inicio:fin`, donde `inicio` es el índice del primer elemento incluido y `fin` es el índice del primer elemento excluido. Por ejemplo:



```
Enseñando.py X
Enseñando.py > ...
1 #tuplas
2 tupla = "manzana", "banana", "naranja", "kiwi", "mango"
3 subtupla = tupla[1:4] # Obtiene un subconjunto de elementos desde el índice 1 hasta el 3
4 print(subtupla) # Imprime: ("banana", "naranja", "kiwi")
5
6
7
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

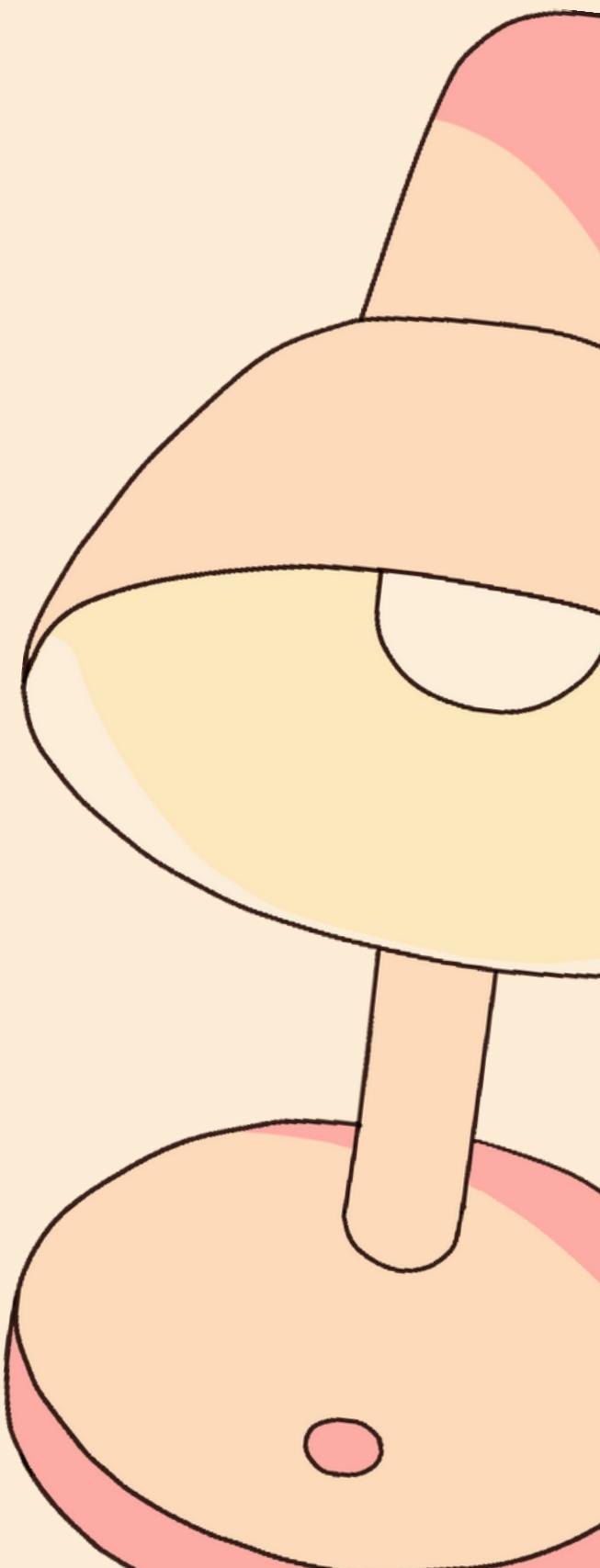
- PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python311/python.exe c:/Users/Miguel/folders/KALI/PYTHON/Enseñando.py ('banana', 'naranja', 'kiwi')
- Miguel-PC ▶ PYTHON

Recuerda que las tuplas son estructuras de datos inmutables, lo que significa que no se pueden modificar una vez creadas. Por lo tanto, el acceso a los elementos de una tupla solo permite la lectura de los valores, no su modificación.

tuplas

Métodos y operaciones en tuplas:

Para resumir puedes usar los métodos y operaciones de una lista en tuplas
index(), count(), Concatenación y demás



tuplas

Empaquetado y desempaquetado de tuplas

Empaquetado de valores en una tupla: El empaquetado de valores en una tupla se refiere a la acción de agrupar varios valores en una sola tupla. Esto se puede hacer simplemente separando los valores por comas y rodeándolos con paréntesis. Por ejemplo:

```
python  
  
tupla = 1, 2, 3  
print(tupla) # Imprime: (1, 2, 3)
```

En este caso, los valores 1, 2 y 3 se empaquetan en la tupla.

Desempaquetado de una tupla en variables individuales: El desempaquetado de una tupla implica extraer los valores de la tupla y asignarlos a variables individuales. Esto se puede hacer asignando la tupla a una secuencia de variables, donde el número de variables debe coincidir con el número de elementos en la tupla. Por ejemplo:

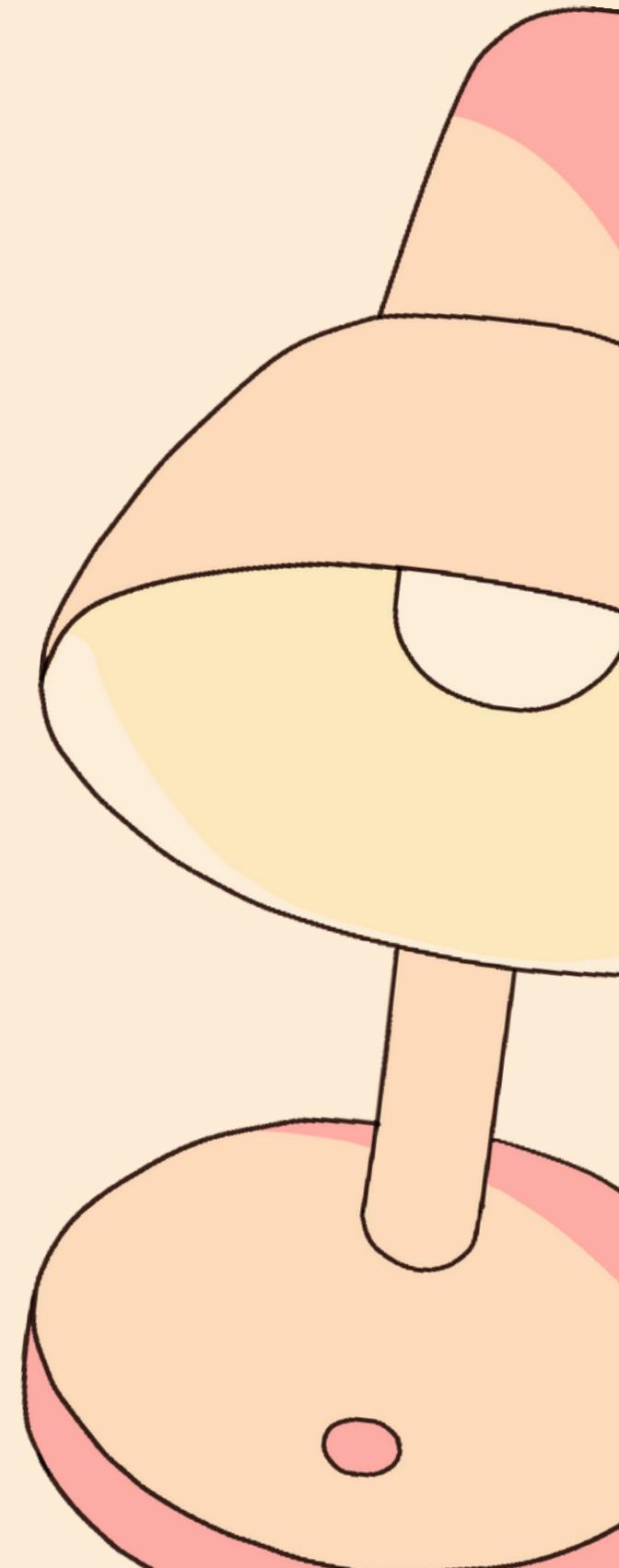
```
python  
  
tupla = (1, 2, 3)  
a, b, c = tupla  
print(a) # Imprime: 1  
print(b) # Imprime: 2  
print(c) # Imprime: 3
```

tuplas

Empaqueado y desempaqueado de tuplas

En este caso, los valores de la tupla se desempaquetan y se asignan a las variables a, b y c respectivamente.

El empaquetado y desempaqueado de tuplas es una técnica útil cuando se necesita trabajar con conjuntos de valores relacionados. El empaquetado permite agrupar valores en una tupla, mientras que el desempaqueado permite extraer los valores de una tupla en variables individuales para su posterior uso.



sets

Introducción a los conjuntos

Definición de conjuntos: Un conjunto en Python es una colección desordenada de elementos únicos. Los conjuntos se utilizan para almacenar elementos sin repetición y no tienen un orden específico. Los conjuntos se definen utilizando llaves {} o la función set(). Por ejemplo:

```
python  
  
conjunto1 = {1, 2, 3}  
conjunto2 = set([4, 5, 6])
```

En este caso, se crean dos conjuntos, conjunto1 que contiene los elementos 1, 2 y 3, y conjunto2 que contiene los elementos 4, 5 y 6.

Sintaxis de los conjuntos: Los conjuntos en Python se definen entre llaves {}. Los elementos dentro de un conjunto se separan por comas. Cada elemento en un conjunto debe ser único, es decir, no puede haber duplicados. Por ejemplo:

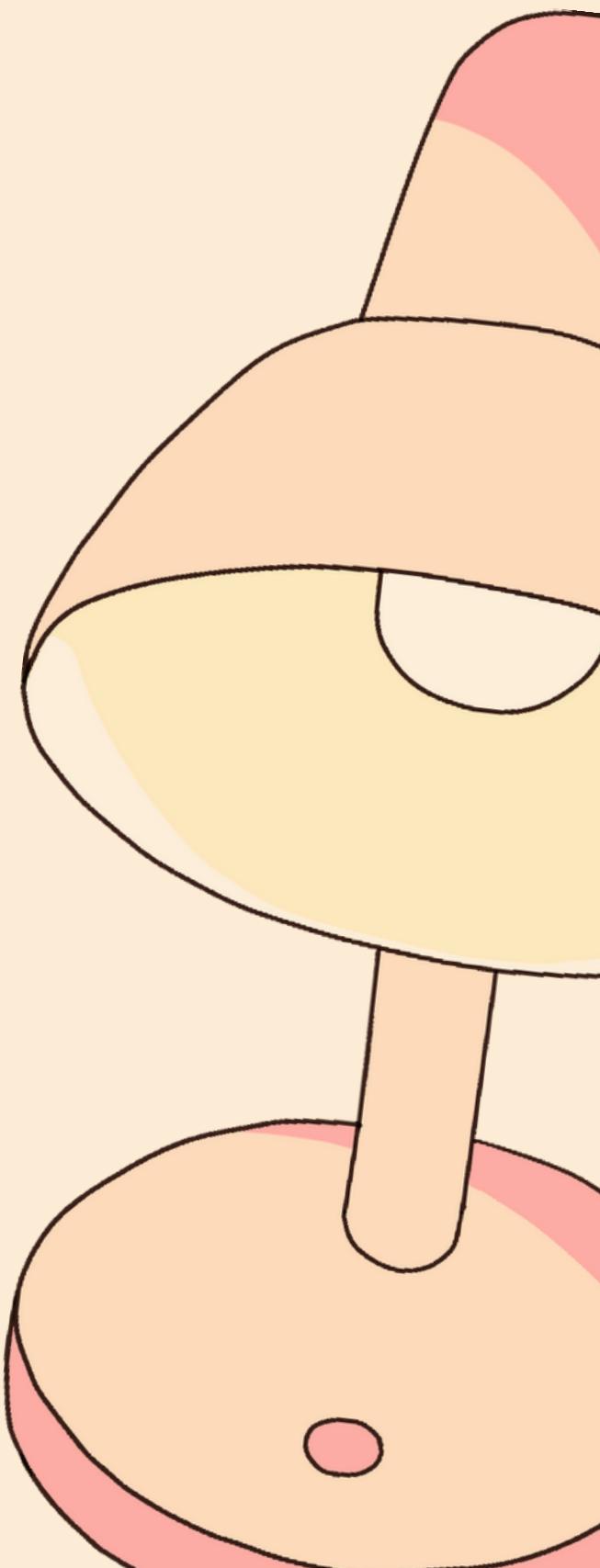
```
python  
  
conjunto = {1, 2, 3}
```

sets

Introducción a los conjuntos

En este caso, se define un conjunto llamado conjunto que contiene los elementos 1, 2 y 3.

Los conjuntos en Python son útiles cuando se necesita almacenar una colección de elementos únicos y no se requiere un orden específico. La sintaxis de los conjuntos es sencilla y permite crear conjuntos de forma rápida y sencilla utilizando llaves o la función set().



sets

Creación y modificación de conjuntos

Creación de conjuntos vacíos: Se puede crear un conjunto vacío utilizando la función `set()` o utilizando llaves vacías `{}`. Por ejemplo:

```
python  
  
conjunto_vacio = set()
```

En este caso, se crea un conjunto vacío llamado `conjunto_vacio`.

Creación de conjuntos con elementos iniciales: Se pueden crear conjuntos con elementos iniciales enumerándolos entre llaves `{}`. Por ejemplo:

```
python  
  
conjunto = {1, 2, 3}
```

En este caso, se crea un conjunto llamado `conjunto` que contiene los elementos 1, 2 y 3.

sets

Creación y modificación de conjuntos

Agregar elementos a un conjunto (add()): Para agregar un elemento a un conjunto existente, se utiliza el método add(). Por ejemplo:

```
python
conjunto = {1, 2, 3}
conjunto.add(4)
```

En este caso, se agrega el elemento 4 al conjunto.

Eliminar elementos de un conjunto (remove(), discard(), pop()): Para eliminar elementos de un conjunto, se pueden utilizar diferentes métodos. El método remove() elimina un elemento específico del conjunto. Si el elemento no existe en el conjunto, se genera un error. El método discard() también elimina un elemento, pero si el elemento no está presente, no se genera ningún error. El método pop() elimina y devuelve un elemento aleatorio del conjunto. Por ejemplo:

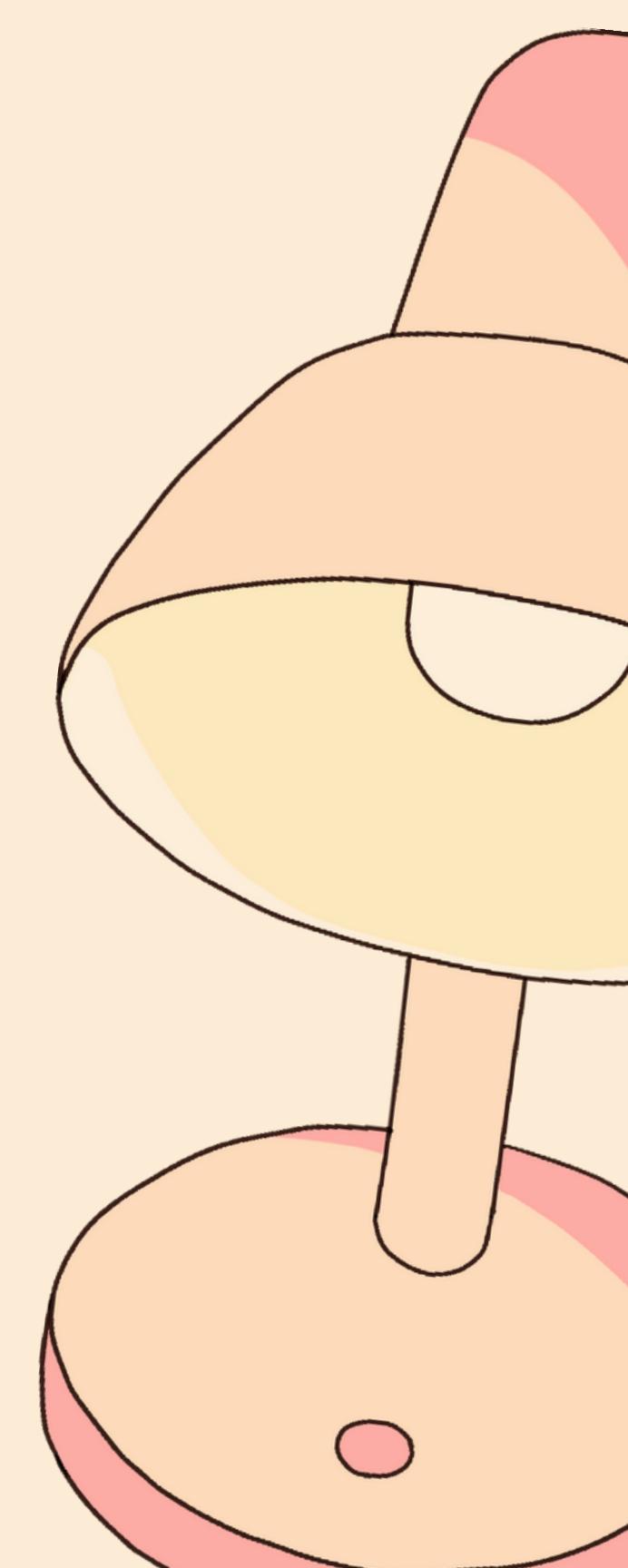
```
python
conjunto = {1, 2, 3}
conjunto.remove(2)
conjunto.discard(4)
elemento = conjunto.pop()
```

sets

Creación y modificación de conjuntos

En este caso, se elimina el elemento 2 del conjunto utilizando el método `remove()`, se intenta eliminar el elemento 4 (que no existe) utilizando el método `discard()`, y se elimina y se asigna a la variable `elemento` un elemento aleatorio del conjunto utilizando el método `pop()`.

La creación de conjuntos vacíos o con elementos iniciales es sencilla utilizando la función `set()` o las llaves `{}`. Los elementos se agregan a un conjunto utilizando el método `add()`, y se pueden eliminar utilizando los métodos `remove()`, `discard()` o `pop()`.



```
Enseñando.py
Enseñando.py > ...
1 #sets
2 conjunto = {1, 2, 3}
3 conjunto.remove(2)
4 conjunto.discard(4)
5 elemento = conjunto.pop()
6 print(conjunto)

PROBLEMS OUTPUT DEBUG CONSOLE TERM
● PYTHON & C:/Users/Miguel-PC/App
o/folders/KALI/PYTHON/Enseñando.py
{3}
○ Miguel-PC ▶ PYTHON ▶ ✓ |
```

sets

Operaciones y métodos de los conjuntos:

Operaciones de conjuntos: Los conjuntos en Python admiten varias operaciones para realizar operaciones comunes entre conjuntos.

Unión (|): La operación de unión combina dos conjuntos y devuelve un nuevo conjunto que contiene todos los elementos presentes en ambos conjuntos, sin duplicados. Se puede realizar utilizando el operador |. Por ejemplo:

```
python
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
union = conjunto1 | conjunto2
```

En este caso, union contendrá el conjunto {1, 2, 3, 4, 5}.

Intersección (&): La operación de intersección devuelve un nuevo conjunto que contiene los elementos comunes a ambos conjuntos. Se puede realizar utilizando el operador &. Por ejemplo:

```
python
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
interseccion = conjunto1 & conjunto2
```

En este caso, interseccion contendrá el conjunto {3}.

sets

Operaciones y métodos de los conjuntos:

Diferencia (-): La operación de diferencia devuelve un nuevo conjunto que contiene los elementos que están en el primer conjunto pero no en el segundo conjunto. Se puede realizar utilizando el operador -. Por ejemplo:

```
python
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
diferencia = conjunto1 - conjunto2
```

En este caso, diferencia contendrá el conjunto {1, 2}.

Diferencia simétrica (^): La operación de diferencia simétrica devuelve un nuevo conjunto que contiene los elementos que están en uno de los conjuntos, pero no en ambos conjuntos. Se puede realizar utilizando el operador ^ . Por ejemplo:

```
python
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
diferencia_simetrica = conjunto1 ^ conjunto2
```

En este caso, diferencia_simetrica contendrá el conjunto {1, 2, 4, 5}.

sets

Operaciones y métodos de los conjuntos:

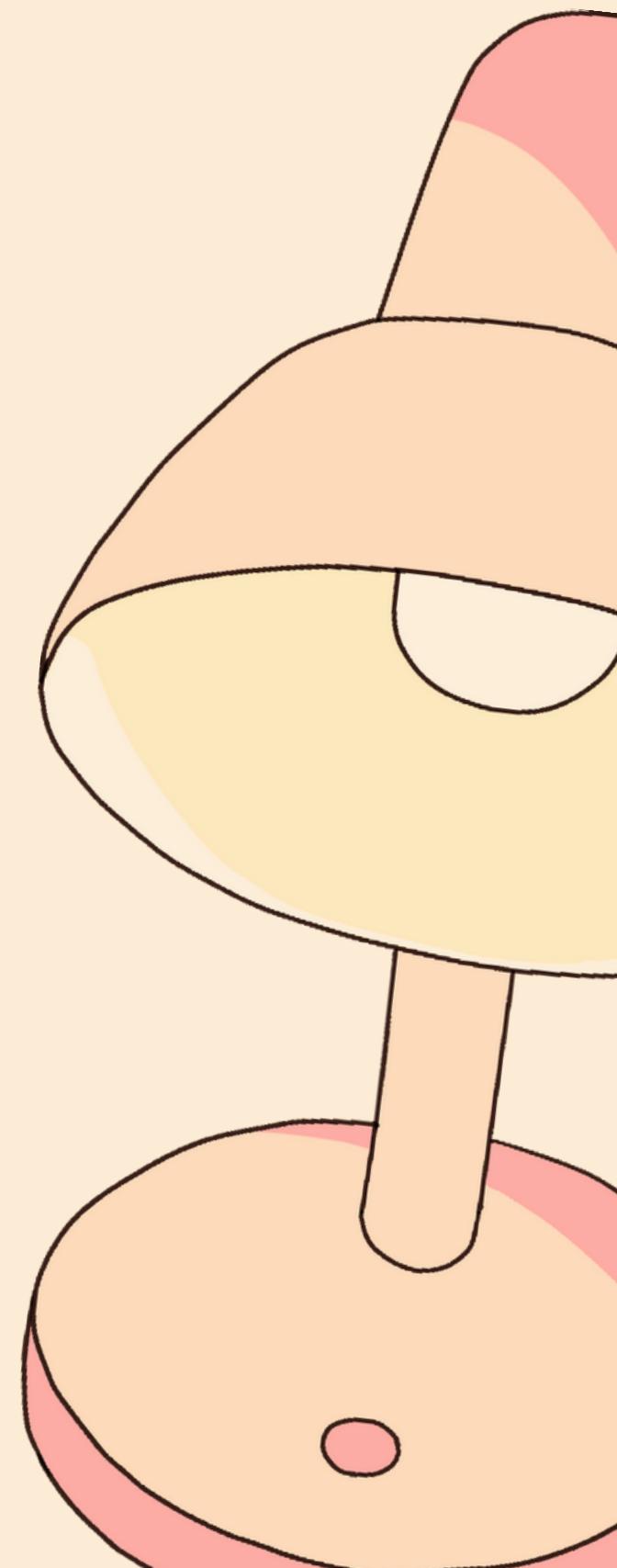
Métodos de los conjuntos: Los conjuntos en Python también tienen métodos incorporados que permiten realizar operaciones específicas en conjuntos.

union(): Devuelve un nuevo conjunto que es la unión de dos conjuntos.

```
python
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
union = conjunto1.union(conjunto2)
```

intersection(): Devuelve un nuevo conjunto que es la intersección de dos conjuntos.

```
python
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
interseccion = conjunto1.intersection(conjunto2)
```



sets

Operaciones y métodos de los conjuntos:

difference(): Devuelve un nuevo conjunto que es la diferencia entre dos conjuntos.

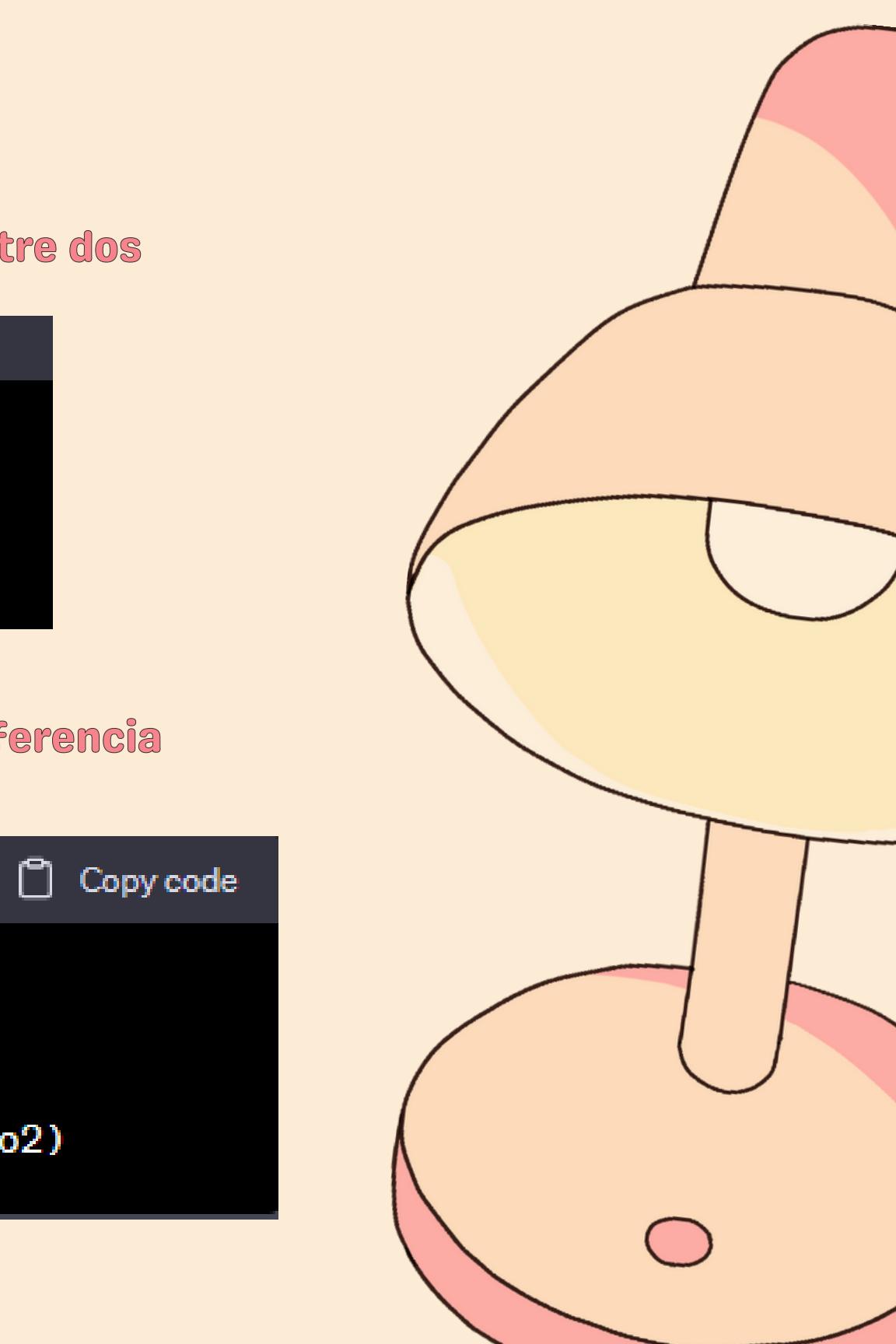
```
python

conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
diferencia = conjunto1.difference(conjunto2)
```

symmetric_difference(): Devuelve un nuevo conjunto que es la diferencia simétrica entre dos conjuntos.

```
python

conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
diferencia_simetrica = conjunto1.symmetric_difference(conjunto2)
```



SETS

OPERACIONES Y MÉTODOS DE LOS CONJUNTOS:

issubset(): Verifica si un conjunto es subconjunto de otro conjunto.

```
python

conjunto1 = {1, 2}
conjunto2 = {1, 2, 3, 4, 5}
es_subconjunto = conjunto1.issubset(conjunto2)
```

issuperset(): Verifica si un conjunto es un superconjunto de otro conjunto.

```
python

conjunto1 = {1, 2, 3, 4, 5}
conjunto2 = {1, 2}
es_superconjunto = conjunto1.issuperset(conjunto2)
```

SETS

PROPIEDADES DE LOS CONJUNTOS:

Elementos únicos en un conjunto: Un conjunto en Python garantiza que no puede contener elementos duplicados. Si intentas agregar un elemento que ya está presente en el conjunto, no se realizará ninguna duplicación. Esto es especialmente útil cuando necesitas almacenar una colección de elementos únicos y no te importa el orden de los elementos.

```
python  
  
conjunto = {1, 2, 3, 2, 4, 3}  
print(conjunto)
```

La salida de este código será: {1, 2, 3, 4}. Como puedes ver, los elementos duplicados (2 y 3) se eliminan automáticamente en el conjunto, y solo se mantiene una instancia de cada elemento.

SETS

PROPIEDADES DE LOS CONJUNTOS:

No tienen orden específico: Los conjuntos en Python no mantienen un orden específico de los elementos. Esto significa que el orden en el que se agregaron los elementos puede no ser necesariamente el orden en que se conservan en el conjunto. Al imprimir un conjunto o iterar sobre sus elementos, el orden puede variar entre diferentes ejecuciones del programa.

```
python  
  
conjunto = {3, 1, 2}  
print(conjunto)
```

La salida de este código podría ser `{1, 2, 3}` o `{3, 1, 2}`. El orden de los elementos puede cambiar, pero los elementos seguirán siendo los mismos. Estas propiedades de los conjuntos en Python los hacen útiles en situaciones donde necesitas almacenar una colección de elementos únicos y el orden de los elementos no es relevante. Los conjuntos se pueden utilizar para verificar la pertenencia de un elemento, eliminar duplicados y realizar operaciones de conjuntos eficientemente.

SETS

VERIFICACIÓN DE PERTENENCIA Y TAMAÑO DE CONJUNTOS:

Verificación de pertenencia: Puedes verificar si un elemento está presente en un conjunto utilizando el operador `in`. Devolverá un valor booleano `True` si el elemento está en el conjunto, y `False` en caso contrario.

```
python

conjunto = {1, 2, 3, 4, 5}
print(2 in conjunto) # True
print(6 in conjunto) # False
```

En este ejemplo, verificamos si los elementos 2 y 6 pertenecen al conjunto.

Tamaño de un conjunto: Para obtener el tamaño o la cantidad de elementos en un conjunto, puedes utilizar la función `len()`. Devolverá un entero que representa la cantidad de elementos en el conjunto.

```
python

conjunto = {1, 2, 3, 4, 5}
print(len(conjunto)) # 5
```

SETS

VERIFICACIÓN DE PERTENENCIA Y TAMAÑO DE CONJUNTOS:

Aquí, obtenemos el tamaño del conjunto, que es igual a 5. Estos métodos son útiles para verificar si un elemento específico está presente en un conjunto y obtener el tamaño del conjunto. Puedes utilizarlos para realizar comprobaciones y cálculos basados en la pertenencia y el tamaño de conjuntos en tus programas de Python.

DICCIONARIOS

INTRODUCCIÓN A LOS DICCIONARIOS:

Definición de diccionarios: Un diccionario en Python es una estructura de datos que te permite almacenar pares clave-valor. Cada elemento del diccionario consiste en una clave única y su respectivo valor asociado. Puedes pensar en un diccionario como un conjunto de definiciones, donde cada palabra (clave) tiene su significado (valor).

Sintaxis de los diccionarios: Los diccionarios se definen utilizando llaves {}. Cada par clave-valor se separa por dos puntos :. La clave y el valor pueden ser de cualquier tipo de dato válido en Python, como enteros, cadenas, listas, tuplas, etc.

```
python

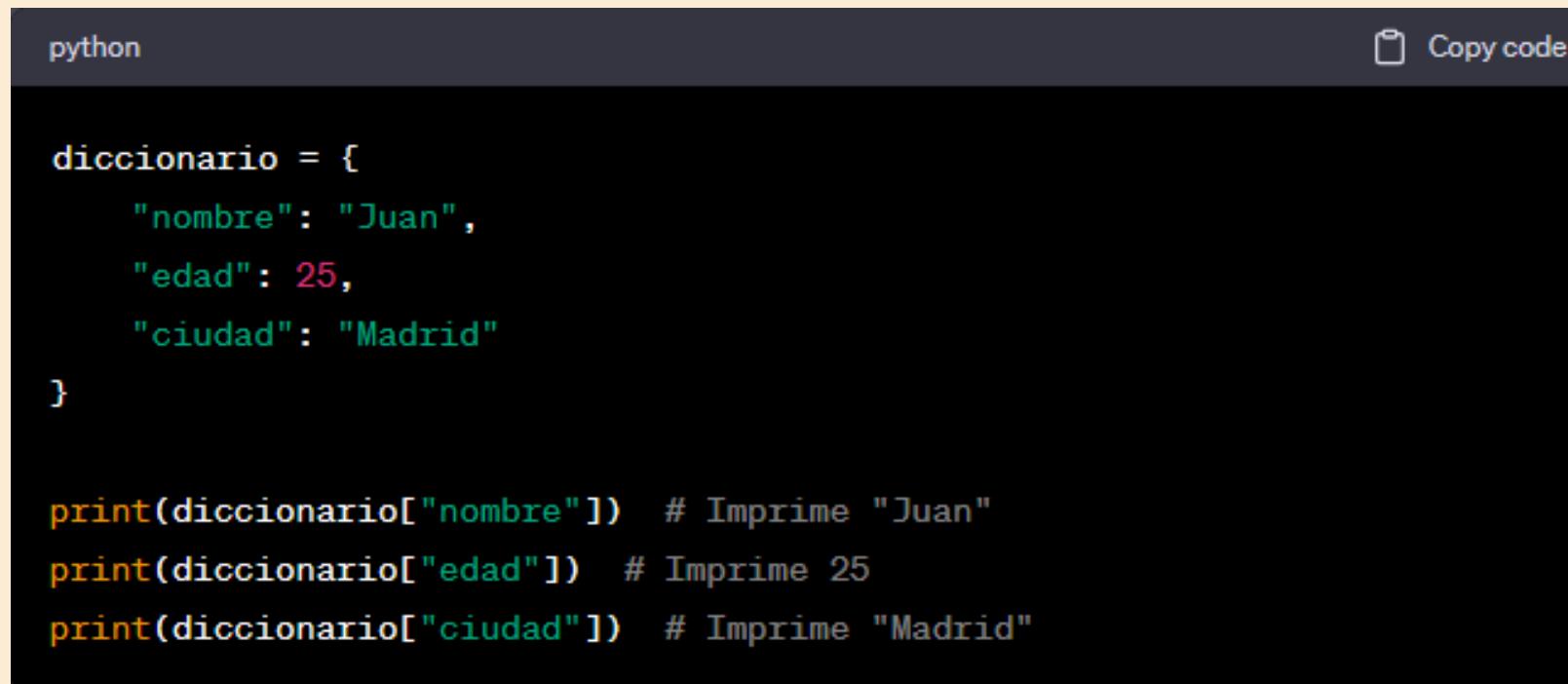
diccionario = {
    "clave1": valor1,
    "clave2": valor2,
    "clave3": valor3
}
```

En este ejemplo, se crea un diccionario con tres elementos. Cada elemento tiene una clave y su respectivo valor asociado.

DICCIONARIOS

INTRODUCCIÓN A LOS DICCIONARIOS:

Puedes acceder a los valores de un diccionario utilizando la sintaxis de corchetes [] y proporcionando la clave correspondiente. Por ejemplo:



```
python

diccionario = {
    "nombre": "Juan",
    "edad": 25,
    "ciudad": "Madrid"
}

print(diccionario["nombre"]) # Imprime "Juan"
print(diccionario["edad"]) # Imprime 25
print(diccionario["ciudad"]) # Imprime "Madrid"
```

En este caso, accedemos a los valores del diccionario utilizando las claves "nombre", "edad" y "ciudad".

Los diccionarios son útiles cuando necesitas almacenar información estructurada en forma de pares clave-valor y deseas acceder rápidamente a los valores utilizando sus claves. Puedes utilizar diccionarios para representar datos como registros, configuraciones, asociaciones y más.

DICCIONARIOS

CREACIÓN Y MODIFICACIÓN DE DICCIONARIOS:

Creación de diccionarios vacíos: Puedes crear un diccionario vacío utilizando llaves {} o utilizando la función dict(). Por ejemplo:

```
python  
  
diccionario_vacio = {}  
diccionario_vacio = dict()
```

Creación de diccionarios con elementos iniciales: Puedes crear un diccionario con elementos iniciales especificando los pares clave-valor dentro de las llaves {}. Por ejemplo:

```
python  
  
diccionario = {  
    "clave1": valor1,  
    "clave2": valor2,  
    "clave3": valor3  
}
```

DICCIONARIOS

CREACIÓN Y MODIFICACIÓN DE DICCIONARIOS:

Agregar elementos a un diccionario: Puedes agregar elementos a un diccionario asignando un nuevo valor a una clave existente o utilizando una nueva clave. Por ejemplo:

```
python  
  
diccionario = {"clave1": valor1}  
diccionario["clave2"] = valor2
```

Modificar valores en un diccionario: Puedes modificar el valor asociado a una clave existente en un diccionario. Simplemente asigna un nuevo valor a la clave correspondiente. Por ejemplo:

```
python  
  
diccionario = {"clave": valor_inicial}  
diccionario["clave"] = nuevo_valor
```

DICCIONARIOS

CREACIÓN Y MODIFICACIÓN DE DICCIONARIOS:

En estos ejemplos, `valor1`, `valor2`, `valor3`, `valor_inicial` y `nuevo_valor` representan los valores que deseas asignar a las claves correspondientes en el diccionario. Recuerda que en los diccionarios, las claves deben ser únicas, pero los valores pueden ser duplicados. Puedes utilizar las claves para acceder y modificar los valores en un diccionario de manera eficiente. Los diccionarios son muy útiles cuando necesitas almacenar y manipular datos estructurados en forma de pares clave-valor.

DICCIONARIOS

ACCESO A ELEMENTOS DE UN DICCIONARIO:

Acceso mediante claves: Puedes acceder a los valores de un diccionario utilizando las claves correspondientes. Utiliza la sintaxis de corchetes [] y proporciona la clave entre los corchetes. Por ejemplo:

```
python Copy code
diccionario = {"clave1": valor1, "clave2": valor2, "clave3": valor3}
valor = diccionario["clave2"]
print(valor) # Imprime el valor asociado a "clave2"
```

Obtener valores asociados a una clave: Puedes utilizar la misma sintaxis de corchetes [] para obtener el valor asociado a una clave específica. Por ejemplo:

```
python Copy code
diccionario = {"nombre": "Juan", "edad": 25, "ciudad": "Madrid"}
nombre = diccionario["nombre"]
edad = diccionario["edad"]
ciudad = diccionario["ciudad"]
```

DICCIONARIOS

ACCESO A ELEMENTOS DE UN DICCIONARIO:

Obtener todas las claves de un diccionario: Puedes utilizar el método `keys()` para obtener una lista con todas las claves de un diccionario. Por ejemplo:

```
python Copy code
diccionario = {"nombre": "Juan", "edad": 25, "ciudad": "Madrid"}
claves = diccionario.keys()
```

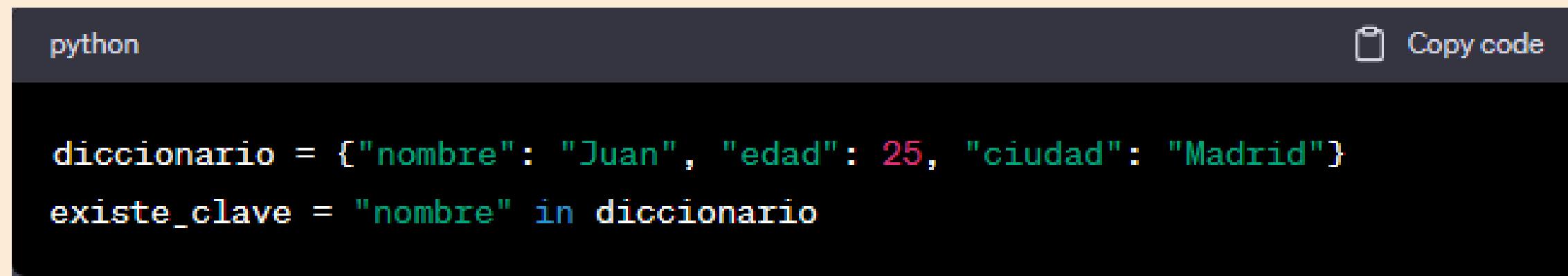
Obtener todos los valores de un diccionario: Puedes utilizar el método `values()` para obtener una lista con todos los valores de un diccionario. Por ejemplo:

```
python Copy code
diccionario = {"nombre": "Juan", "edad": 25, "ciudad": "Madrid"}
valores = diccionario.values()
```

DICCIONARIOS

ACCESO A ELEMENTOS DE UN DICCIONARIO:

Verificar si una clave existe en un diccionario: Puedes utilizar el operador `in` para verificar si una clave existe en un diccionario. El operador devuelve `True` si la clave está presente y `False` si no lo está. Por ejemplo:



```
python
Copy code

diccionario = {"nombre": "Juan", "edad": 25, "ciudad": "Madrid"}
existe_clave = "nombre" in diccionario
```

En este caso, `existe_clave` contendrá el valor `True` porque la clave "nombre" está presente en el diccionario.

Recuerda que los diccionarios en Python no están ordenados, por lo que el orden en que se devuelven las claves o valores puede variar. Sin embargo, puedes utilizar las claves para acceder a los valores de manera eficiente en un diccionario.

DICCIONARIOS

MÉTODOS Y OPERACIONES DE LOS DICCIONARIOS:

- **Métodos para añadir, eliminar y modificar elementos:**
update(): Este método permite agregar nuevos pares clave-valor a un diccionario o actualizar los valores de las claves existentes. Puedes pasar otro diccionario o una secuencia de pares clave-valor como argumento. Por ejemplo:

```
python                                         Copy code

diccionario = {"clave1": valor1, "clave2": valor2}
diccionario.update({"clave3": valor3, "clave4": valor4})
```

- **pop():** Este método permite eliminar un elemento de un diccionario utilizando su clave como argumento. El método devuelve el valor asociado a la clave eliminada. Por ejemplo:

```
python                                         Copy code

diccionario = {"clave1": valor1, "clave2": valor2}
valor_eliminado = diccionario.pop("clave1")
```

DICCIONARIOS

MÉTODOS Y OPERACIONES DE LOS DICCIONARIOS:

del: La declaración del permite eliminar un elemento o incluso un diccionario completo utilizando su clave. Por ejemplo:

```
python
```

 Copy code

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

Métodos para obtener información sobre el diccionario:

- **keys():** Este método devuelve una lista con todas las claves presentes en el diccionario. Por ejemplo:

```
python
```

 Copy code

```
diccionario = {"nombre": "Juan", "edad": 25, "ciudad": "Madrid"}  
claves = diccionario.keys()
```

DICCIONARIOS

MÉTODOS Y OPERACIONES DE LOS DICCIONARIOS:

values(): Este método devuelve una lista con todos los valores presentes en el diccionario. Por ejemplo:

```
python
```

[Copy code](#)

```
diccionario = {"nombre": "Juan", "edad": 25, "ciudad": "Madrid"}  
valores = diccionario.values()
```

items(): Este método devuelve una lista de tuplas, donde cada tupla contiene una clave y su valor correspondiente. Por ejemplo:

```
python
```

[Copy code](#)

```
diccionario = {"nombre": "Juan", "edad": 25, "ciudad": "Madrid"}  
elementos = diccionario.items()
```

DICCIONARIOS

PROPIEDADES DE LOS DICCIONARIOS:

Los diccionarios en Python son estructuras de datos que almacenan elementos en pares clave-valor. Son útiles cuando necesitas asociar una clave única a un valor.

Las claves son únicas y los valores pueden ser de cualquier tipo de datos.

Ejemplo:

```
python
datos_personales = {"nombre": "Juan", "edad": 25, "ciudad": "Madrid"}  
Copy code
```

En este ejemplo, "nombre", "edad" y "ciudad" son las claves, y "Juan", 25 y "Madrid" son los valores asociados respectivamente.

Los diccionarios son útiles cuando quieras acceder a los valores utilizando las claves en lugar de índices, y permiten almacenar información diversa en un solo objeto.

DICCIONARIOS

EJERCICIOS :

Problema:

El refugio de gatitos "Patitas Felices" necesita llevar un registro de los gatitos que han sido rescatados. Cada gatito tiene un nombre, una edad y un estado de adopción. Además, se debe asignar un cuidador responsable para cada gatito.

El refugio ha recibido a los siguientes gatitos:

1. Nombre: Luna, Edad: 2 meses, Adopción: No
2. Nombre: Simba, Edad: 6 meses, Adopción: Sí
3. Nombre: Mia, Edad: 1 año, Adopción: No
4. Nombre: Max, Edad: 3 meses, Adopción: No

El refugio cuenta con los siguientes cuidadores disponibles:

1. Cuidador: Ana
2. Cuidador: Pedro
3. Cuidador: Laura

Utilizando listas, tuplas, sets y diccionarios, registra los datos de cada gatito y asigna manualmente un cuidador para cada uno. Muestra el registro actualizado con los datos de cada gatito, incluyendo su nombre, edad, estado de adopción y el nombre de su cuidador asignado.

DICCIONARIOS

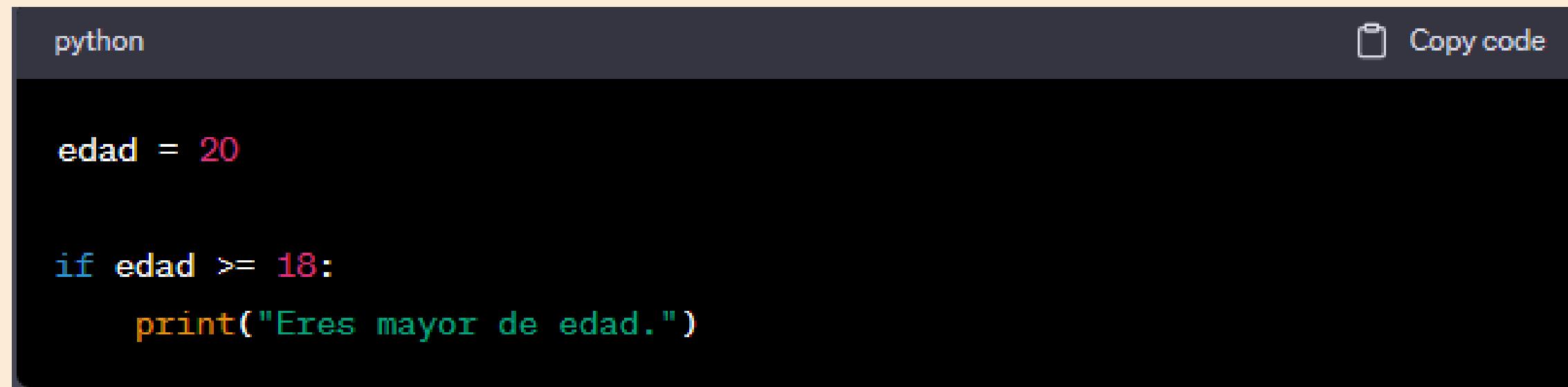
EJERCICIOS : SOLUCION

```
Enseñando.py > ...
1  gatitos = [
2      {"Nombre": "Luna", "Edad": "2 meses", "Adopcion": "No"},
3      {"Nombre": "Simba", "Edad": "6 meses", "Adopcion": "Sí"},
4      {"Nombre": "Mia", "Edad": "1 año", "Adopcion": "No"},
5      {"Nombre": "Max", "Edad": "3 meses", "Adopcion": "No"}
6  ]
7
8 cuidadores = {"Ana", "Pedro", "Laura"}
9
10 registro = []
11
12 gatitos[0]["Cuidador"] = "Ana"
13 gatitos[1]["Cuidador"] = "Pedro"
14 gatitos[2]["Cuidador"] = "Laura"
15 gatitos[3]["Cuidador"] = "Ana"
16
17 registro.extend(gatitos)
18
19 # Imprimir datos del primer gatito
20 print("Nombre:", registro[0]["Nombre"])
21 print("Edad:", registro[0]["Edad"])
22 print("Adopcion:", registro[0]["Adopcion"])
23 print("Cuidador:", registro[0]["Cuidador"])
24 print("-----")
25
26 # Imprimir datos del segundo gatito
27 print("Nombre:", registro[1]["Nombre"])
28 print("Edad:", registro[1]["Edad"])
29 print("Adopcion:", registro[1]["Adopcion"])
30 print("Cuidador:", registro[1]["Cuidador"])
31 print("-----")
32 |
33 # Imprimir datos del tercer gatito
34 print("Nombre:", registro[2]["Nombre"])
35 print("Edad:", registro[2]["Edad"])
36 print("Adopcion:", registro[2]["Adopcion"])
37 print("Cuidador:", registro[2]["Cuidador"])
38 print("-----")
39
40 # Imprimir datos del cuarto gatito
41 print("Nombre:", registro[3]["Nombre"])
42 print("Edad:", registro[3]["Edad"])
43 print("Adopcion:", registro[3]["Adopcion"])
44 print("Cuidador:", registro[3]["Cuidador"])
45 print("-----")
46
```

CONDICIONALES

IF STATEMENTS (DECLARACIONES IF)

En Python, las declaraciones if se utilizan para tomar decisiones condicionales en función de la evaluación de una expresión. El bloque de código dentro del if se ejecuta solo si la condición es verdadera (evalúa a True). Aquí tienes un ejemplo y una explicación de cómo funcionan las declaraciones if en Python:



A screenshot of a Python code editor window. The title bar says "python". The code area contains the following Python code:

```
python

edad = 20

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

The code defines a variable `edad` with the value 20. It then checks if `edad` is greater than or equal to 18. If true, it prints the message "Eres mayor de edad.".

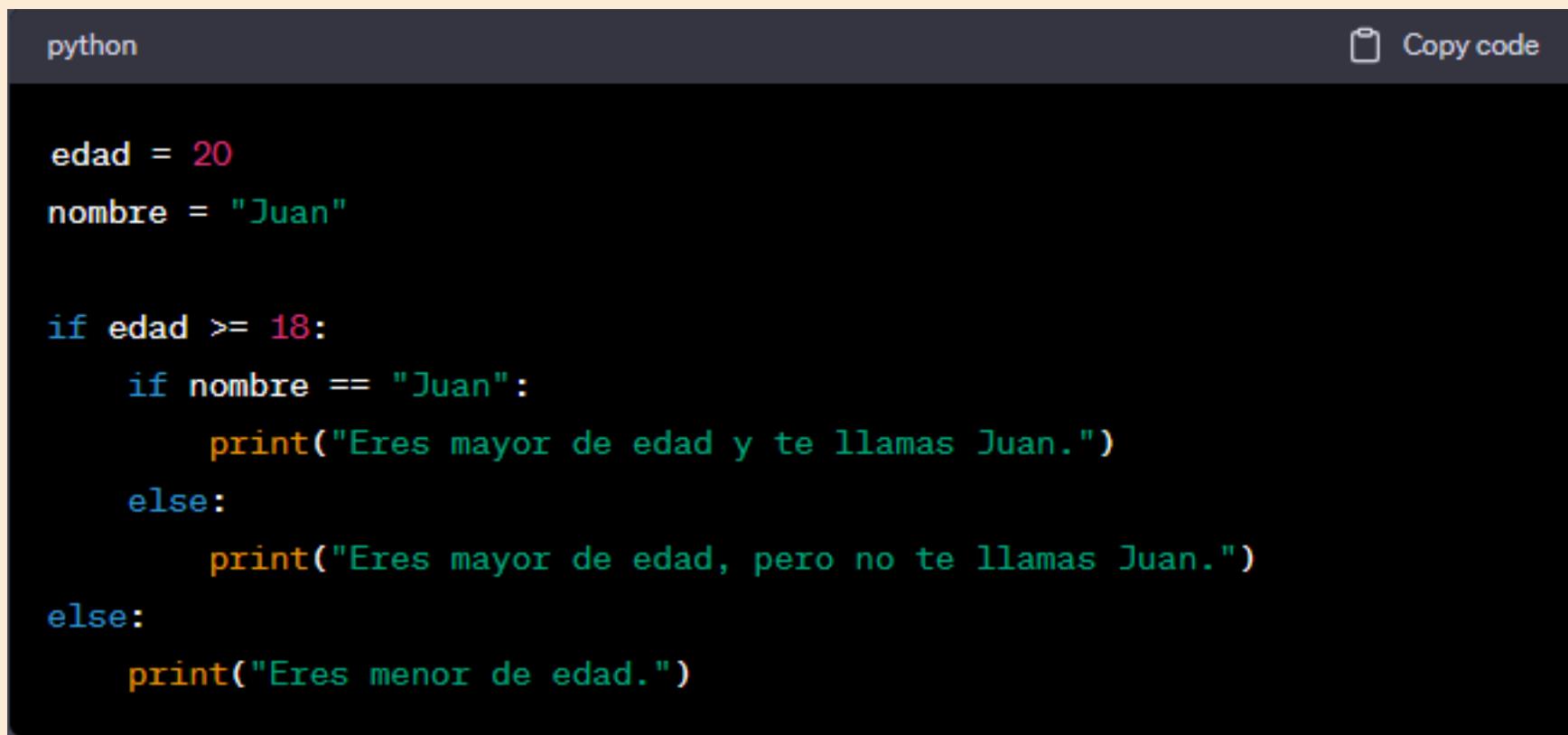
Explicación:

- La variable `edad` se inicializa con un valor de 20.
- La declaración if se evalúa como True si la condición `edad >= 18` es verdadera.
- En este caso, como la edad es igual a 20 (mayor o igual a 18), la condición es verdadera y se ejecuta el bloque de código dentro del if.
- La instrucción `print` imprime en la consola el mensaje "Eres mayor de edad".

CONDICIONALES

NESTED IF STATEMENTS (DECLARACIONES IF ANIDADAS)

- Las declaraciones if anidadas, también conocidas como if dentro de if, se utilizan para realizar evaluaciones condicionales más complejas. Aquí tienes una explicación breve y un ejemplo de cómo funcionan:



A screenshot of a Python code editor window titled "python". The code is as follows:

```
python

edad = 20
nombre = "Juan"

if edad >= 18:
    if nombre == "Juan":
        print("Eres mayor de edad y te llamas Juan.")
    else:
        print("Eres mayor de edad, pero no te llamas Juan.")
else:
    print("Eres menor de edad.")

Copy code
```

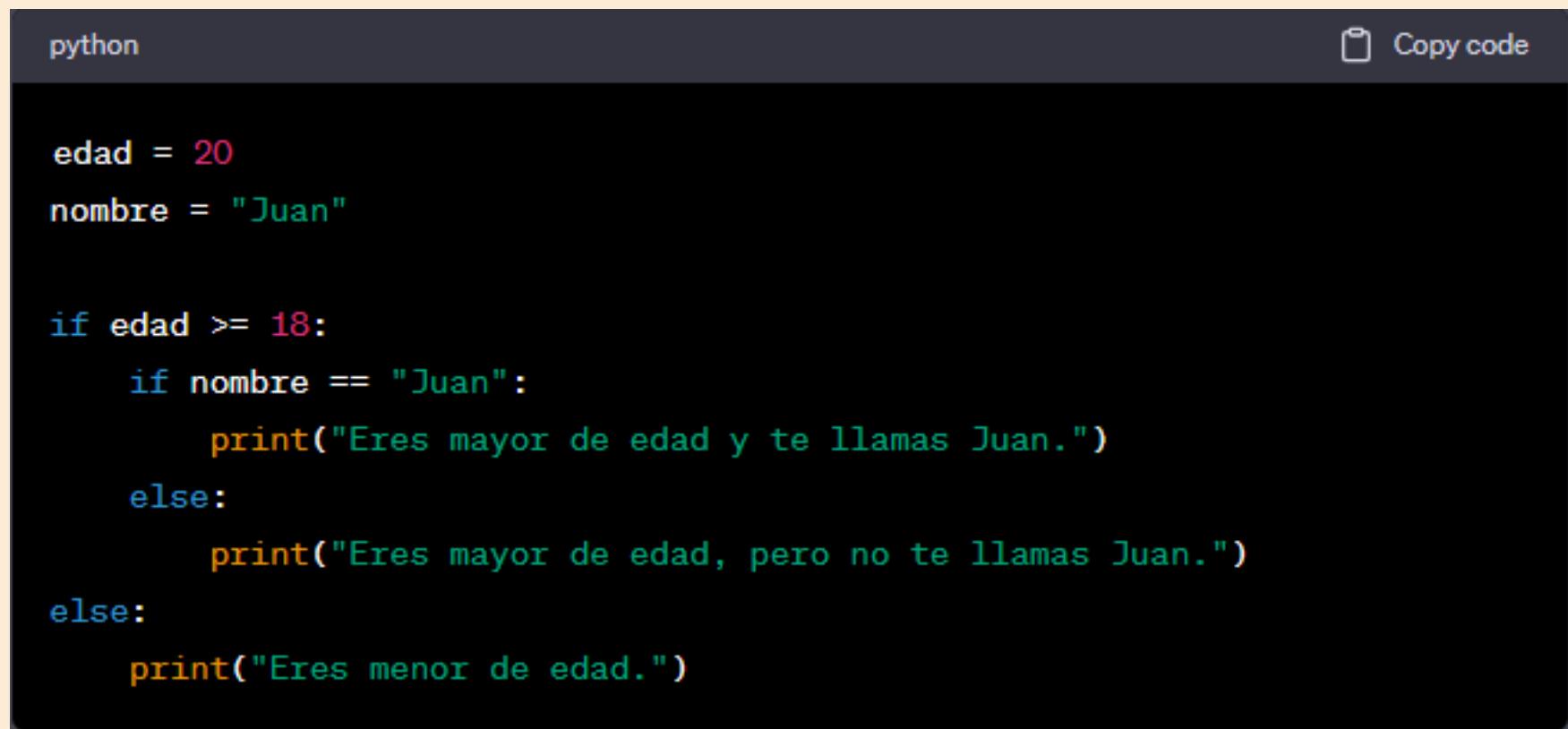
Explicación:

- La variable `edad` se inicializa con un valor de 20 y la variable `nombre` se inicializa con el valor "Juan".
- La primera declaración if se evalúa como True si la condición `edad >= 18` es verdadera.
- Si la condición del primer if es verdadera, se evalúa la siguiente declaración if dentro del bloque de código del primer if.
- En este caso, como la edad es igual a 20 (mayor o igual a 18) y el nombre es "Juan", ambas condiciones son verdaderas y se ejecuta el primer bloque de código dentro del segundo if, que imprime "Eres mayor de edad y te llamas Juan".
- Si alguna de las condiciones anteriores es falsa, se ejecuta el bloque de código del else más cercano.
- Si la primera condición del if es falsa, se ejecuta el bloque de código del else más externo, que imprime "Eres menor de edad".

CONDICIONALES

ELIF STATEMENTS (DECLARACIONES ELIF)

- Las declaraciones if anidadas, también conocidas como if dentro de if, se utilizan para realizar evaluaciones condicionales más complejas. Aquí tienes una explicación breve y un ejemplo de cómo funcionan:



A screenshot of a Python code editor window titled "python". The code is as follows:

```
python

edad = 20
nombre = "Juan"

if edad >= 18:
    if nombre == "Juan":
        print("Eres mayor de edad y te llamas Juan.")
    else:
        print("Eres mayor de edad, pero no te llamas Juan.")
else:
    print("Eres menor de edad.")

Copy code
```

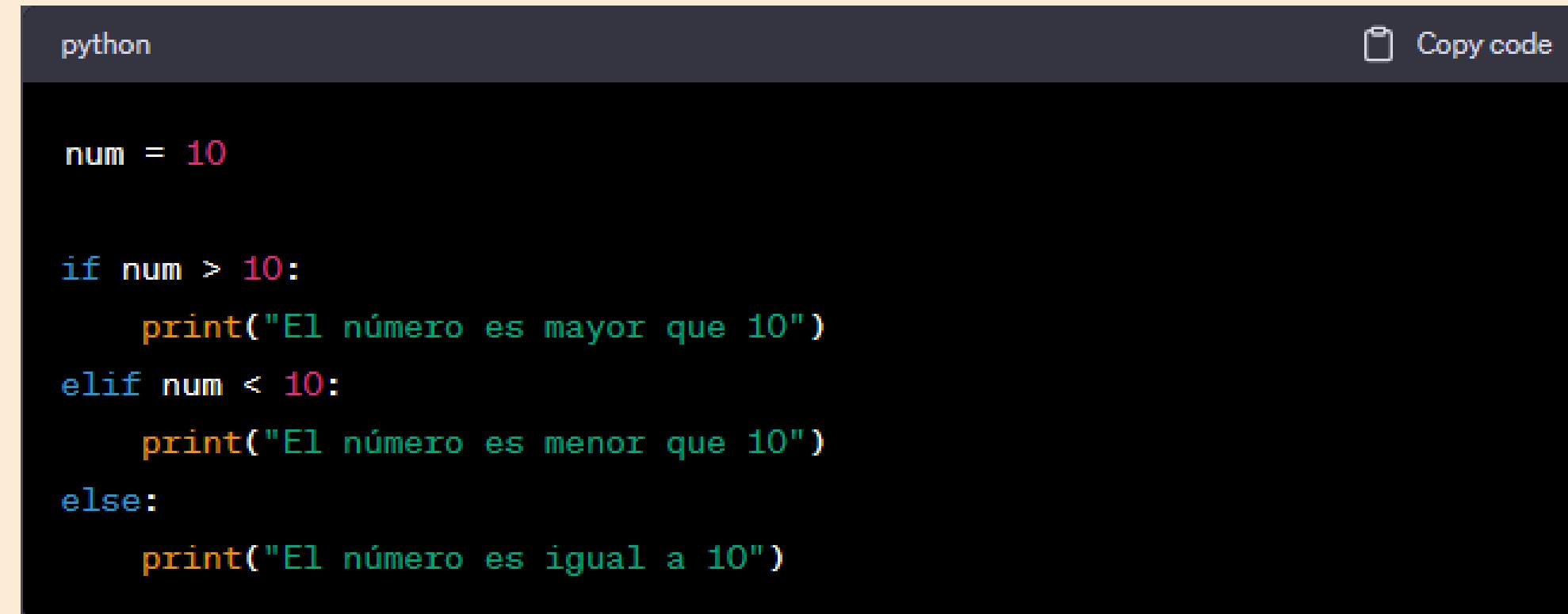
Explicación:

- La variable `edad` se inicializa con un valor de 20 y la variable `nombre` se inicializa con el valor "Juan".
- La primera declaración `if` se evalúa como True si la condición `edad >= 18` es verdadera.
- Si la condición del primer `if` es verdadera, se evalúa la siguiente declaración `if` dentro del bloque de código del primer `if`.
- En este caso, como la edad es igual a 20 (mayor o igual a 18) y el nombre es "Juan", ambas condiciones son verdaderas y se ejecuta el primer bloque de código dentro del segundo `if`, que imprime "Eres mayor de edad y te llamas Juan".
- Si alguna de las condiciones anteriores es falsa, se ejecuta el bloque de código del `else` más cercano.
- Si la primera condición del `if` es falsa, se ejecuta el bloque de código del `else` más externo, que imprime "Eres menor de edad".

CONDICIONALES

IF-ELSE STATEMENTS (DECLARACIONES IF-ELSE)

- En Python, el `elif` se utiliza como parte de las declaraciones condicionales para verificar múltiples condiciones en secuencia. A continuación, te explico cómo se utiliza de manera resumida:
- El `elif` se coloca después de un bloque `if` y antes de un bloque `else`. Permite evaluar una condición adicional si la condición en el `if` anterior resulta ser falsa. Si la condición en el `elif` es verdadera, se ejecuta el bloque de código correspondiente. Si la condición en el `elif` es falsa, se pasa a la siguiente declaración condicional.
- Aquí tienes un ejemplo que muestra el uso de `elif`:
-



A screenshot of a Python code editor window. The title bar says "python". The code area contains the following Python code:

```
python

num = 10

if num > 10:
    print("El número es mayor que 10")
elif num < 10:
    print("El número es menor que 10")
else:
    print("El número es igual a 10")
```

On the right side of the code area, there is a "Copy code" button with a clipboard icon.

- En este ejemplo, se evalúa la variable `num` en diferentes condiciones. Si `num` es mayor que 10, se imprime "El número es mayor que 10". Si `num` es menor que 10, se imprime "El número es menor que 10". Si ninguna de las condiciones anteriores se cumple, se imprime "El número es igual a 10".

CONDICIONALES

COMPARISONS (COMPARACIONES)

- En Python, las comparaciones se utilizan para evaluar si una expresión o variable cumple ciertas condiciones. Aquí tienes una explicación resumida de las comparaciones más comunes:
- Igualdad (==): Compara si dos valores son iguales. Devuelve True si son iguales y False en caso contrario.
-

```
python
x = 5
y = 7
print(x == y) # False
```

Desigualdad (!=): Compara si dos valores son diferentes. Devuelve True si son diferentes y False en caso contrario.

```
python
x = 5
y = 7
print(x != y) # True
```

CONDICIONALES

COMPARISONS (COMPARACIONES)

- Mayor que (`>`), menor que (`<`), mayor o igual que (`>=`), menor o igual que (`<=`): Estas comparaciones se utilizan para verificar el orden relativo entre dos valores. Devuelven `True` si la condición se cumple y `False` en caso contrario.

```
python

x = 5
y = 7
print(x > y)    # False
print(x < y)    # True
print(x >= y)   # False
print(x <= y)   # True
```

- Operadores de pertenencia: Los operadores `in` y `not in` se utilizan para verificar si un valor está presente en una secuencia, como una lista, una tupla o una cadena.

```
python

fruits = ['apple', 'banana', 'orange']
print('apple' in fruits)      # True
print('grape' not in fruits) # True
```

CONDICIONALES

LOGICAL OPERATORS (OPERADORES LÓGICOS)

- En Python, los operadores lógicos se utilizan para combinar o modificar las condiciones booleanas. Aquí tienes una explicación resumida de los operadores lógicos más comunes:
- Operador AND (and): Devuelve True si ambas condiciones son verdaderas, y False en caso contrario.

```
python

x = 5
y = 7
print(x > 0 and y < 10) # True
```

- Operador OR (or): Devuelve True si al menos una de las condiciones es verdadera, y False si ambas condiciones son falsas.

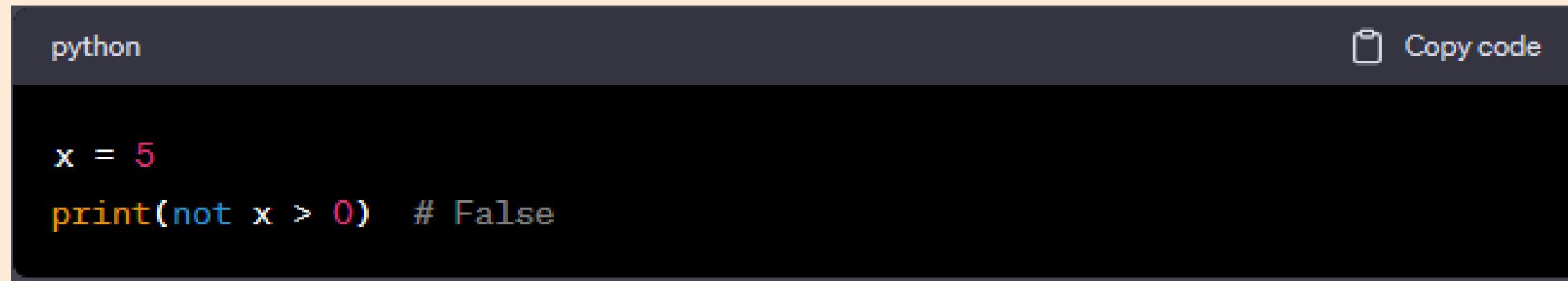
```
python

x = 5
y = 7
print(x > 0 or y < 0) # True
```

CONDICIONALES

LOGICAL OPERATORS (OPERADORES LÓGICOS)

- Operador NOT (not): Devuelve el valor opuesto de una condición. Si la condición es verdadera, devuelve False, y si la condición es falsa, devuelve True.



A screenshot of a code editor window titled "python". The code is as follows:

```
python

x = 5
print(not x > 0) # False
```

The "Copy code" button is visible in the top right corner of the editor window.

- Estos operadores se utilizan principalmente en combinación con las declaraciones condicionales (if, elif, else) para evaluar condiciones más complejas. También se pueden utilizar para modificar o invertir el resultado de una condición.

CONDICIONALES

TRUTHINESS AND FALSENESS (VERACIDAD Y FALSEDAD)

- En Python, los conceptos de "veracidad" (truthiness) y "falsedad" (falsiness) se refieren a cómo se evalúan los valores en contextos booleanos, es decir, cuando se utilizan en expresiones condicionales o como operandos de los operadores lógicos.
- En términos generales, se considera que un valor es "verdadero" si tiene algún contenido o significado, y "falso" si carece de contenido o significado. Aquí tienes una explicación
 - Valores verdaderos (Truthy): Los siguientes valores se consideran verdaderos en Python:
 - Cualquier número distinto de cero (ejemplo: 1, -1, 3.14)
 - Cualquier cadena de texto no vacía (ejemplo: "Hola", "Python")
 - Cualquier lista, tupla, conjunto o diccionario no vacío
 - El valor especial True
 - Cualquier objeto definido por el usuario que no implemente el método especial `__bool__()` o `__len__()` de forma explícita para devolver False
 -
 - Valores falsos (Falsy): Los siguientes valores se consideran falsos en Python:
 - El número cero (0)
 - El valor especial None
 - Cadenas de texto vacías ("", "")
 - Listas, tuplas, conjuntos o diccionarios vacíos
 - El valor especial False
 - Cualquier objeto definido por el usuario que implemente el método especial `__bool__()` o `__len__()` de forma explícita para devolver False
 -

CONDICIONALES

SHORT-CIRCUIT EVALUATION (EVALUACIÓN DE CIRCUITO CORTO)

- Evaluación cortocircuitada con and:
- Cuando se utiliza el operador and, la evaluación se realiza de izquierda a derecha.
- Si el primer operando es falso, el resultado se determina inmediatamente como falso sin evaluar el segundo operando. En este caso, el segundo operando se omite por completo.
- Si el primer operando es verdadero, se evalúa el segundo operando y el resultado final es el valor del segundo operando.
- Ejemplo:
- o significado, y "falso" si carece de contenido o significado. Aquí tienes una explicación

```
python

    a = 5
    b = 10
    if a > 0 and b > 0:
        print("Ambos números son positivos")
```

CONDICIONALES

SHORT-CIRCUIT EVALUATION (EVALUACIÓN DE CIRCUITO CORTO)

- Evaluación cortocircuitada con or:
- Cuando se utiliza el operador or, la evaluación también se realiza de izquierda a derecha.
- Si el primer operando es verdadero, el resultado se determina inmediatamente como verdadero sin evaluar el segundo operando. En este caso, el segundo operando se omite por completo.
- Si el primer operando es falso, se evalúa el segundo operando y el resultado final es el valor del segundo operando.
- Ejemplo:
-

```
python

a = 0
b = 10
if a == 0 or b == 0:
    print("Al menos uno de los números es cero")
```

CONDICIONALES

TERNARY OPERATOR (OPERADOR TERNARIO)

- En Python, el operador ternario es una forma concisa de escribir una expresión condicional en una sola línea. También se conoce como operador condicional.
- La sintaxis general del operador ternario es la siguiente:

```
Enseñando.py > ...
1 #condicionales
2 edad = 20
3 mensaje = "Eres mayor de edad" if edad >= 18 else "Eres menor de edad"
4 print(mensaje) # Imprime "Eres mayor de edad" si la edad es 18 o mayor, de lo contrario imprime "Eres menor de edad"
```

En este ejemplo, la condición `edad >= 18` se evalúa. Si es verdadera, se asigna el valor "Eres mayor de edad" a la variable `mensaje`; de lo contrario, se asigna el valor "Eres menor de edad". El operador ternario puede ser útil para simplificar expresiones condicionales simples y evitar el uso de declaraciones `if-else` más extensas cuando se necesita asignar un valor basado en una condición.

BUCLES

BUCLES WHILE (WHILE LOOPS)

Los bucles while en Python permiten repetir un bloque de código mientras se cumpla una determinada condición. La estructura general de un bucle while es la siguiente:

```
python
while condicion:
    # Código a ejecutar mientras se cumpla la condición
    # Aquí se pueden incluir varias instrucciones
    Copy code
```

La condición es una expresión que se evalúa antes de cada iteración del bucle. Si la condición es verdadera, el bloque de código dentro del bucle se ejecuta. Después de cada iteración, la condición se vuelve a evaluar. Si la condición sigue siendo verdadera, el bucle continúa ejecutándose. Si la condición se vuelve falsa, el bucle se detiene y la ejecución del programa continúa con la siguiente instrucción después del bucle.

Aquí tienes un ejemplo para ilustrar su uso:

BUCLES

BUCLES WHILE (WHILE LOOPS)

En este ejemplo, el bucle while se ejecuta mientras la variable contador sea menor que 5. En cada iteración, se imprime el valor actual del contador y luego se incrementa en 1. El bucle se detiene cuando el contador alcanza el valor de 5. Los bucles while son útiles cuando no se sabe de antemano cuántas veces se va a repetir un bloque de código, ya que se basan en una condición que se evalúa en cada iteración. Sin embargo, es importante tener cuidado de no crear bucles infinitos, donde la condición nunca se vuelve falsa y el bucle continúa ejecutándose indefinidamente.

```
python
contador = 0
while contador < 5:
    print("El contador es:", contador)
    contador += 1
Copy code
```

BUCLES

BUCLES WHILE (WHILE LOOPS)

ANALIZAREMOS el while , Lo mas importante de while es saber que while necesita un contador siempre, la funcion del contador es indicarle cuando tiene que parar por ejemplo:

nuestro contador sera "num_gatitos =", puede ser el nombre que tu quieras lo importante es que se usara como un contador!

while se ejecutara reiteradas veces hasta que la condicion que le pongamos sea False

while significa "mientras" y se leeria asi , "mientras num_gatitos sea menor que 5 sigue imprimiendo hasta que sea False"

```
pytho... Copy code
num_gatitos = 0

while num_gatitos < 5:
    print("iApareció un gatito!")
    num_gatitos += 1

print("Ya hay suficientes gatitos")
```

nuestro contador empezara en 0. y diremos "mientras num_gatitos sea menor que 5" imprimeme lo que este dentro del while hasta que la condicion sea False

siempre hay que poner el contador al ultimo de toda nuestras operaciones que hagamos , el contador cada que while vuelva a pasar por todo el codigo sumara 1

y que significa esto? , que ahora se leeria asi "num_gatitos tiene 1, ya no es mas 0 y sera asi hasta que llegue a 5 donde el while parara su lectura"

BUCLES

BUCLES WHILE (WHILE LOOPS)

```
#bucle
num_gatitos = 0
while num_gatitos < 5:
    print("¡Apareció un gatito!")
    num_gatitos += 1
print("Ya hay suficientes gatitos.")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PYTHON & C:/Users/Miguel-PC/AppData/Local/folders/KALI/PYTHON/Enseñando.py

¡Apareció un gatito!
Ya hay suficientes gatitos.

Miguel-PC PYTHON ✓

cuando sea la primera vuelta se sumara 1 y esto pasara hasta que se cumpla la condicion de num_gatitos sea mayor que 5

esta flecha lo que quiere decir es que . "mientras que num_gatitos sea menor que 5 sigue ejecutando el codigo una y otra y otra y otra vez :D"

En este ejemplo, el bucle while se ejecutará mientras num_gatitos sea menor que 5. En cada iteración del bucle, se imprimirá el mensaje "¡Apareció un gatito!" y luego se incrementará el valor de num_gatitos en 1. El bucle se detendrá cuando num_gatitos alcance el valor de 5. Después del bucle, se imprimirá el mensaje "Ya hay suficientes gatitos."

BUCLES

BUCLES FOR (FOR LOOPS)

Los bucles for se utilizan para iterar sobre una secuencia de elementos, como una lista, una tupla, un conjunto o una cadena de caracteres. La sintaxis básica de un bucle for es la siguiente:

```
python

for elemento in secuencia:
    # Cuerpo del bucle
    # Realizar acciones con cada elemento
```

En cada iteración del bucle, el valor actual de elemento se toma de la secuencia y se ejecutan las instrucciones dentro del cuerpo del bucle. Después de cada iteración, el siguiente elemento de la secuencia se asigna a elemento y el ciclo continúa hasta que se hayan recorrido todos los elementos de la secuencia.

Aquí tienes un ejemplo sencillo que muestra cómo usar un bucle for para imprimir cada elemento de una lista de gatitos:

```
python

gatitos = ["Mimi", "Tom", "Luna", "Simba"]

for gatito in gatitos:
    print(gatito)
```

BUCLES

BUCLES FOR (FOR LOOPS)

En este caso, el bucle for recorre la lista gatitos y en cada iteración, el valor actual se asigna a la variable gatito. Luego se imprime ese valor. En cada iteración, el bucle imprimirá el nombre de un gatito diferente de la lista.

Los bucles for también se pueden utilizar junto con la función range() para generar secuencias numéricas. Por ejemplo, para imprimir los números del 1 al 5, se puede hacer lo siguiente:

```
python
for i in range(1, 6):
    print(i)

```

En este caso, range(1, 6) genera una secuencia de números desde 1 hasta 5 (excluyendo el límite superior). El bucle for itera sobre estos números y los imprime uno por uno. Esto es solo una introducción básica a los bucles for en Python. Puedes realizar diferentes acciones dentro del cuerpo del bucle, como realizar cálculos, modificar listas, acceder a elementos de una tupla, etc. Los bucles for son una herramienta poderosa para iterar sobre elementos y realizar operaciones repetitivas de manera eficiente.

BUCLES

BUCLES ANIDADOS (NESTED LOOPS)

Los bucles anidados, también conocidos como bucles dentro de bucles, son estructuras que contienen un bucle dentro de otro bucle. Esto nos permite realizar iteraciones más complejas y realizar acciones repetitivas en múltiples niveles.

La sintaxis básica de los bucles anidados en Python es la siguiente:

```
python
for elemento_exterior in secuencia_exterior:
    # Código del bucle exterior

    for elemento_interior in secuencia_interior:
        # Código del bucle interior
        # Realizar acciones con cada elemento
```

En cada iteración del bucle exterior, se ejecuta el bucle interior completo. Esto significa que el bucle interior se repite completamente para cada iteración del bucle exterior.

BUCLES

BUCLES ANIDADOS (NESTED LOOPS)

Aquí tienes un ejemplo sencillo que muestra cómo utilizar bucles anidados:

```
python

gatitos = ['Tom', 'Garfield', 'Whiskers']
acciones = ['jugar', 'comer', 'dormir']

for gatito in gatitos:
    for accion in acciones:
        print(gatito, 'está', accion)

Copy code
```

En este caso, tenemos una lista de nombres de gatitos (`gatitos`) y una lista de acciones (`acciones`). El bucle exterior recorre cada nombre de gatito, mientras que el bucle interior recorre cada acción. Dentro del bucle anidado, imprimimos el nombre del gatito y la acción que está realizando.

BUCLES

BUCLES ANIDADOS (NESTED LOOPS)

El resultado de este código sería:



A screenshot of a code editor window. At the top right, there is a "Copy code" button with a clipboard icon. The main area contains the following text output:

```
Tom está jugando
Tom está comiendo
Tom está durmiendo
Garfield está jugando
Garfield está comiendo
Garfield está durmiendo
Whiskers está jugando
Whiskers está comiendo
Whiskers está durmiendo
```

Este ejemplo ilustra cómo se pueden combinar elementos de diferentes listas utilizando bucles anidados para generar diferentes combinaciones de acciones para cada gatito.

BUCLES

SENTENCIAS DE CONTROL DE BUCLES (LOOP CONTROL STATEMENTS)

a. Sentencia Break (Break Statement):

La sentencia `break` se utiliza para interrumpir la ejecución de un bucle de forma anticipada. Cuando se encuentra un `break` dentro de un bucle, el programa salta inmediatamente fuera del bucle y continúa con la ejecución del código que se encuentra después del bucle.

```
python

for i in range(1, 6):
    if i == 3:
        break
    print(i)

Copy code
```

En este ejemplo, el bucle `for` recorre los números del 1 al 5. Sin embargo, cuando se cumple la condición `i == 3`, se encuentra el `break` y se sale del bucle de forma prematura. Como resultado, solo se imprimirán los números 1 y 2.

BUCLES

SENTENCIAS DE CONTROL DE BUCLES (LOOP CONTROL STATEMENTS)

b. Sentencia Continue (Continue Statement):

La sentencia continue se utiliza para saltar a la siguiente iteración de un bucle sin ejecutar el resto del código dentro del bucle. Cuando se encuentra un continue, el programa ignora el resto del bloque de código dentro del bucle y continúa con la siguiente iteración.



A screenshot of a Python code editor. The code is as follows:

```
python

for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

The code uses a for loop to iterate from 1 to 5. Inside the loop, it checks if the value of i is 3. If it is, it executes the continue statement, which skips the rest of the loop body for that iteration. The print statement is only executed for i values 1, 2, 4, and 5. There is a "Copy code" button in the top right corner of the code editor window.

En este ejemplo, cuando i es igual a 3, se encuentra el continue y se salta la impresión del número 3. El bucle continúa con la siguiente iteración y se imprimen los números restantes del 1 al 5, excepto el 3.

BUCLES

SENTENCIAS DE CONTROL DE BUCLES (LOOP CONTROL STATEMENTS)

c. Sentencia Pass (Pass Statement):

La sentencia pass se utiliza como un marcador de posición o como un código de relleno cuando no se desea ejecutar ninguna acción en un bloque de código. Simplemente se omite y no tiene ningún efecto en la ejecución del programa.



A screenshot of a Python code editor. The title bar says "python". The code area contains the following Python code:

```
for i in range(1, 6):
    if i == 3:
        pass
    print(i)
```

On the right side of the code area, there is a "Copy code" button with a clipboard icon.

En este ejemplo, cuando i es igual a 3, se encuentra el pass, pero no se realiza ninguna acción. El bucle continúa su ejecución normal y se imprimen todos los números del 1 al 5.

Estas sentencias de control de bucles (break, continue y pass) proporcionan flexibilidad en el control del flujo de ejecución dentro de los bucles y permiten ajustar el comportamiento según las condiciones deseadas.

BUCLES

SENTENCIAS DE CONTROL DE BUCLES (LOOP CONTROL STATEMENTS)

La función `range(1, 6)` crea un objeto de tipo rango que representa una secuencia de números enteros. En este caso, el rango comienza desde el número 1 y va hasta el número anterior al 6 (es decir, 5). El primer número especificado (1) es el límite inferior del rango y el segundo número (6) es el límite superior.
La sintaxis general de la función `range()` es `range(start, stop, step)`, donde:

- `start` es el valor inicial del rango (incluido).
- `stop` es el valor final del rango (excluido).
- `step` es el tamaño del incremento entre los valores del rango (opcional).

En el caso de `range(1, 6)`, el rango comienza en 1 y termina en 5 (el 6 se excluye). El valor predeterminado para `step` es 1, lo que significa que el rango avanza de uno en uno. Por lo tanto, la secuencia generada será 1, 2, 3, 4, 5.
Esta función es comúnmente utilizada en bucles `for` para iterar sobre una secuencia de números en Python.

BUCLES

FUNCIÓN RANGE (RANGE FUNCTION)

La función range() en Python es una función incorporada que se utiliza para generar una secuencia de números en un rango especificado. La sintaxis básica de la función range() es la siguiente:

```
range(start, stop, step)
```

- **start (opcional):** Especifica el valor inicial de la secuencia. Si no se proporciona, el valor predeterminado es 0.
- **stop:** Especifica el valor final de la secuencia. Este valor no se incluirá en la secuencia generada. Es necesario proporcionar este argumento.
- **step (opcional):** Especifica la diferencia entre los números consecutivos en la secuencia. Si no se proporciona, el valor predeterminado es 1.

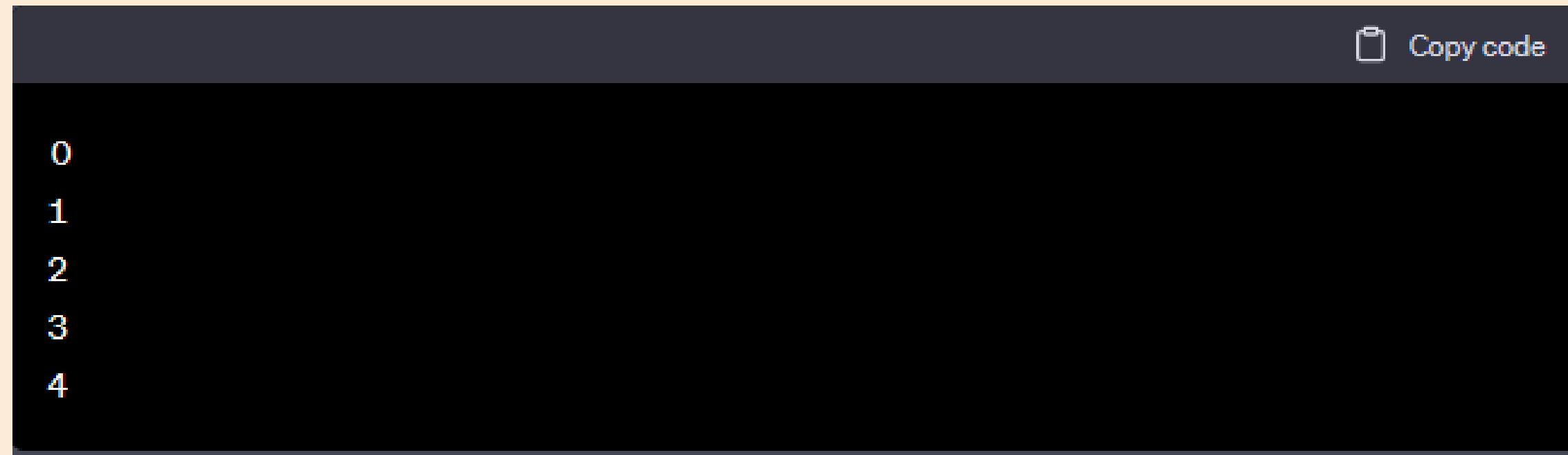
Aquí hay algunos ejemplos para ilustrar cómo se utiliza la función range():

I. Generar una secuencia de números desde 0 hasta 4:

```
python
for num in range(5):
    print(num)
```

BUCLES

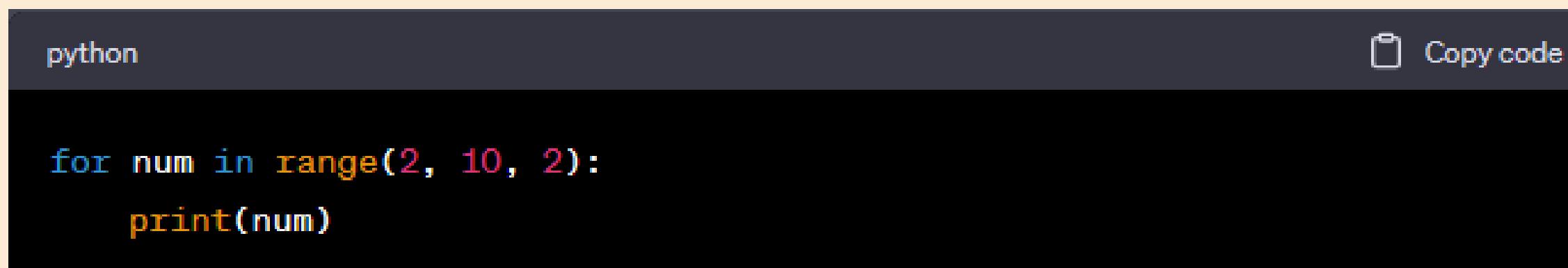
FUNCIÓN RANGE (RANGE FUNCTION)



A terminal window with a dark background and light text. In the top right corner, there is a "Copy code" button with a clipboard icon. The output of the code consists of five numbers: 0, 1, 2, 3, and 4, each on a new line.

```
0
1
2
3
4
```

Generar una secuencia de números desde 2 hasta 10 (excluyendo el 10) con un paso de 2:



A terminal window with a dark background and light text. In the top right corner, there is a "Copy code" button with a clipboard icon. The code is written in Python and prints the numbers 2, 4, 6, 8, and 10, each on a new line. The code uses the range function with parameters 2, 10, and 2.

```
python

for num in range(2, 10, 2):
    print(num)
```

BUCLES

FUNCIÓN RANGE (RANGE FUNCTION)



A terminal window with a dark background and light text. It displays the numbers 2, 4, 6, and 8, each on a new line. In the top right corner, there is a small icon followed by the text "Copy code".

```
2
4
6
8
```

Crear una lista de números desde 1 hasta 5:



A terminal window with a dark background and light text. It shows a Python script with two lines of code: "numbers = list(range(1, 6))" and "print(numbers)". In the top right corner, there is a small icon followed by the text "Copy code".

```
python

numbers = list(range(1, 6))
print(numbers)
```

BUCLES

FUNCIÓN RANGE (RANGE FUNCTION)



A screenshot of a code editor window. The title bar says "csharp". The code area contains the following C# code:

```
[1, 2, 3, 4, 5]
```

On the right side of the code area, there is a "Copy code" button with a clipboard icon.

Es importante tener en cuenta que el objeto range no almacena todos los números en memoria, sino que genera los números a medida que se necesitan, lo que lo hace eficiente para trabajar con grandes secuencias de números.

BUCLES

FUNCIÓN ENUMERATE (ENUMERATE FUNCTION)

La función `enumerate()` en Python es una función incorporada que se utiliza para iterar sobre una secuencia mientras se lleva un seguimiento del índice de cada elemento. Proporciona una forma conveniente de obtener tanto el índice como el valor de cada elemento en un bucle. Aquí tienes una explicación detallada:
La sintaxis básica de la función `enumerate()` es la siguiente:

```
python
enumerate(iterable, start=0)
Copy code
```

- **iterable:** La secuencia (como una lista, tupla, cadena, etc.) sobre la cual se va a iterar.
- **start (opcional):** El valor inicial del índice. Si no se proporciona, el valor predeterminado es 0.

BUCLES

FUNCIÓN ENUMERATE (ENUMERATE FUNCTION)

La función enumerate() devuelve un objeto de tipo enumeración que genera tuplas que contienen un contador (el índice) y los valores correspondientes de la secuencia. Puedes utilizar este objeto en un bucle for para iterar sobre los elementos con sus respectivos índices.
Aquí tienes un ejemplo para ilustrar cómo se utiliza la función enumerate():

```
python

fruits = ['apple', 'banana', 'orange']

for index, fruit in enumerate(fruits):
    print(index, fruit)
```

```
0 apple
1 banana
2 orange
```

BUCLES

FUNCIÓN ENUMERATE (ENUMERATE FUNCTION)

En este ejemplo, la función `enumerate(fruits)` devuelve un objeto enumeración que genera tuplas con el índice y el valor de cada fruta en la lista `fruits`. Luego, en el bucle `for`, desempaquetamos las tuplas en las variables `index` y `fruit`, y las imprimimos en cada iteración.

La función `enumerate()` es útil cuando necesitas acceder tanto al índice como al valor de los elementos de una secuencia mientras iteras sobre ella. Puedes utilizar esta función en diversas situaciones, como recorrer listas, cadenas u otras secuencias donde necesitas realizar un seguimiento del índice correspondiente.

BUCLES

COMPRENSIONES DE LISTAS (LIST COMPREHENSIONS)

Las comprensiones de listas, o list comprehensions en inglés, son una forma concisa y poderosa de crear listas en Python. Proporcionan una sintaxis compacta para generar una nueva lista basada en otra secuencia o mediante la aplicación de una expresión a cada elemento de la secuencia. Aquí tienes una explicación detallada:
La sintaxis básica de una comprensión de listas es la siguiente:

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

- **expresión:** La expresión que se va a aplicar a cada elemento de la secuencia para generar un nuevo valor en la lista resultante.
- **elemento:** La variable que representa cada elemento de la secuencia.
- **secuencia:** La secuencia de la cual se tomarán los elementos para generar la lista resultante.
- **condición (opcional):** Una condición opcional que filtra los elementos de la secuencia. Solo los elementos que cumplan con esta condición se incluirán en la lista resultante.

Aquí tienes algunos ejemplos para ilustrar cómo se utilizan las comprensiones de listas:

I. Generar una lista de los cuadrados de los números del 1 al 5:

```
python
```

 Copy code

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

BUCLES

COMPRENSIONES DE LISTAS (LIST COMPREHENSIONS)

```
[1, 4, 9, 16, 25]
```

Generar una lista de los números pares del 0 al 10:

```
python
even_numbers = [x for x in range(11) if x % 2 == 0]
print(even_numbers)
```

 Copy code

```
[0, 2, 4, 6, 8, 10]
```

BUCLES

COMPRENSIONES DE LISTAS (LIST COMPREHENSIONS)

Generar una lista de las letras en mayúscula de una cadena:

```
python
string = "Hello, World!"
uppercase_letters = [char for char in string if char.isupper()]
print(uppercase_letters)
```

```
['H', 'W']
```

Las comprensiones de listas son una forma elegante y eficiente de generar listas en Python. Permiten combinar la generación de elementos y la aplicación de una expresión en una sola línea de código, lo que puede hacer que el código sea más legible y conciso. Puedes utilizar comprensiones de listas en una variedad de situaciones donde necesites crear una lista basada en una secuencia existente o aplicar una expresión a cada elemento de la secuencia.

BUCLES

USO DE BUCLES ANIDADOS:

Cuando anidamos bucles, colocamos uno dentro del otro, lo que nos permite iterar sobre múltiples dimensiones o estructuras de datos anidadas. Esto es útil cuando trabajamos con listas, matrices, diccionarios u otras estructuras de datos que contienen elementos anidados.

Aquí tienes un ejemplo para ilustrar cómo se utilizan los bucles anidados en Python:

```
python
fruits = ["apple", "banana", "orange"]
colors = ["red", "yellow", "orange"]

for fruit in fruits:
    for color in colors:
        print(fruit, color)
```

```
apple red
apple yellow
apple orange
banana red
banana yellow
banana orange
orange red
orange yellow
orange orange
```

BUCLES

USO DE BUCLES ANIDADOS:

En este ejemplo, tenemos dos listas: fruits y colors. Utilizamos un bucle for anidado para iterar sobre cada elemento de fruits y luego, dentro de ese bucle, iteramos sobre cada elemento de colors. Imprimimos la combinación de cada fruta y cada color en cada iteración.

El uso de bucles anidados es especialmente útil cuando trabajamos con estructuras de datos multidimensionales, como matrices. Por ejemplo:

```
python
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for row in matrix:
    for element in row:
        print(element)
```

```
1
2
3
4
5
6
7
8
9
```

BUCLES

USO DE BUCLES ANIDADOS:

En este caso, `matrix` es una lista que contiene listas anidadas, que representan filas y elementos dentro de una matriz. Mediante bucles anidados, podemos recorrer cada elemento individual en la matriz y realizar operaciones o tareas específicas.

Recuerda que al anidar bucles, debes tener en cuenta la estructura de tus datos y cómo deseas iterar sobre ellos para asegurarte de obtener los resultados deseados. Los bucles anidados son una herramienta poderosa para trabajar con estructuras de datos anidadas y te permiten realizar operaciones en múltiples niveles.

BUALES

EJERCICIOS :

Problemática: Registro de gatitos

Descripción: Eres el encargado de registrar información sobre los gatitos en un refugio de animales. Cada gatito tiene un nombre, edad y color de pelaje. Tu tarea es procesar la información y realizar diferentes acciones según las condiciones dadas.

Instrucciones:

1. Crea una lista llamada "gatitos" que contenga tuplas de la siguiente forma: (nombre, edad, color). Puedes agregar al menos 5 gatitos a la lista.
2. Utiliza un bucle for para recorrer la lista de gatitos y mostrar la información de cada gatito en el siguiente formato: "Nombre: [nombre], Edad: [edad], Color: [color]".
3. Utiliza un bucle for y una variable para contar cuántos gatitos tienen un color específico, por ejemplo, "naranja". Imprime el resultado.
4. Crea un set llamado "colores_distintos" y utiliza un bucle for para iterar sobre la lista de gatitos y agregar los colores de pelaje al set. Imprime el set resultante.
5. Utiliza un bucle for y una condición if para imprimir el nombre de los gatitos que tienen menos de 2 años.

BUCLES

EJERCICIOS : SOLUCION

```
# Paso 1: Crear la lista de gatitos
gatitos = [
    ("Mimi", 2, "gris"),
    ("Pelusa", 1, "blanco"),
    ("Luna", 3, "negro"),
    ("Tom", 2, "naranja"),
    ("Simba", 4, "marrón")
]
```

```
# Paso 2: Mostrar la información de cada gatito
print("Información de los gatitos:")
for gatito in gatitos:
    nombre, edad, color = gatito
    print(f"Nombre: {nombre}, Edad: {edad}, Color: {color}")
```

```
# Paso 3: Contar gatitos con un color específico
color_busqueda = "naranja"
contador_color = 0
for gatito in gatitos:
    nombre, edad, color = gatito
    if color == color_busqueda:
        contador_color += 1

print(f"Cantidad de gatitos con color {color_busqueda}: {contador_color}")
```

BUCLES

EJERCICIOS : SOLUCION

```
# Paso 4: Obtener los colores distintos de pelaje
colores_distintos = set()
for gatito in gatitos:
    nombre, edad, color = gatito
    colores_distintos.add(color)

print("Colores distintos de pelaje:")
for color in colores_distintos:
    print(color)
```

```
# Paso 5: Imprimir los gatitos con menos de 2 años
print("Gatitos con menos de 2 años:")
for gatito in gatitos:
    nombre, edad, color = gatito
    if edad < 2:
        print(nombre)
```



FUNCIONES

PYTHON

¿QUÉ SON LAS FUNCIONES EN PYTHON?

1.1 DEFINICIÓN Y UTILIDAD.

1.2 SINTAXIS BÁSICA DE UNA
FUNCIÓN EN PYTHON.

ESTRUCTURA DE UNA FUNCIÓN EN PYTHON.

2.1 DECLARACIÓN DE LA FUNCIÓN.

2.2 PARÁMETROS DE ENTRADA.

2.3 CUERPO DE LA FUNCIÓN.

2.4 VALOR DE RETORNO.

DECLARACIÓN Y USO DE FUNCIONES EN PYTHON.

3.1 CREACIÓN DE FUNCIONES PERSONALIZADAS.

3.2 LLAMADO DE FUNCIONES Y USO DE PARÁMETROS.

3.3 EJEMPLOS PRÁCTICOS DE FUNCIONES EN PYTHON.

TIPOS DE VARIABLES DENTRO DE LAS FUNCIONES EN PYTHON.

4.1 VARIABLES LOCALES.

4.2 VARIABLES GLOBALES.

FUNCIONES ANIDADAS Y RECURSIVIDAD EN PYTHON.

5.1 FUNCIONES ANIDADAS.

5.2 FUNCIONES RECURSIVAS.

FUNCIONES PREDEFINIDAS DE PYTHON.



6.1 USO DE FUNCIONES
PREDEFINIDAS.

BIBLIOTECAS Y MÓDULOS EN PYTHON.

7.1 UTILIZACIÓN DE BIBLIOTECAS
Y MÓDULOS.

7.2 EJEMPLOS PRÁCTICOS DE
BIBLIOTECAS Y MÓDULOS EN
PYTHON.

BUENAS PRÁCTICAS EN LA PROGRAMACIÓN DE FUNCIONES EN PYTHON.

8.1 NOMENCLATURA DE
FUNCIONES.

8.2 DOCUMENTACIÓN DE
FUNCIONES.

CONCLUSIONES Y RECOMENDACIONES FINALES.

9.2 RECOMENDACIONES PARA EL
USO DE FUNCIONES EN PYTHON.

¿QUÉ SON LAS FUNCIONES EN PYTHON?

1.1 DEFINICIÓN Y UTILIDAD.

Una función en Python es un bloque de código que **hace una tarea específica** y se puede usar en cualquier momento. Es como una herramienta que te ayuda a **dividir tu programa en partes más pequeñas** y fáciles de entender.

La mejor parte de las funciones es que te permiten **escribir código una vez y usarlo muchas veces**, lo que ahorra tiempo y esfuerzo. Además, **las funciones pueden ser compartidas entre diferentes programas**, lo que las hace muy útiles para los programadores. En resumen, las funciones **hacen que tu código sea más organizado, legible y fácil de mantener**.

En inglés, "def" es una abreviatura de "define" que significa "definir" en español

1.2 SINTAXIS BÁSICA DE UNA FUNCIÓN EN PYTHON.

La sintaxis básica de una función en Python comienza con la palabra clave "def" seguida del nombre de la función, paréntesis y dos puntos.

Dentro de la función, se escribe el código que realiza la tarea deseada. La función puede o no recibir argumentos y puede devolver un valor usando la palabra clave "return". Es importante nombrar a la función de una manera clara y descriptiva, para que su uso sea fácil de entender.

ESTRUCTURA DE UNA FUNCIÓN EN PYTHON.

2.1 DECLARACIÓN DE LA FUNCIÓN.

La **declaración** de una función en Python **es la forma en que se define la función**. Esta declaración comienza con la palabra clave "**def**", seguida del nombre de la función, y **los paréntesis que pueden o no contener argumentos**. La sintaxis básica es la siguiente:

```
python
def nombre_funcion(argumento1, argumento2, ...):
    # cuerpo de la función
    
```

Es importante en python el tener cuidado con los espacios

2.1 DECLARACIÓN DE LA FUNCIÓN.

El nombre de la función debe seguir las mismas reglas que las variables en Python. Los argumentos son variables **opcionales** que pueden ser pasadas a la función para ser utilizadas en su cuerpo. Si no se necesita ningún argumento, los paréntesis todavía deben ser incluidos. El cuerpo de la función es donde se coloca el código que se ejecutará cuando se llame a la función.

Es importante destacar que la definición de una función no la ejecuta, sino que simplemente la crea para ser llamada posteriormente. La ejecución de la función se realiza mediante una llamada a la función.

```
def saludar():
    print("Hola, bienvenido!")
```

2.1 DECLARACIÓN DE LA FUNCIÓN.

En este ejemplo, la función `saludar()` no tiene ningún argumento. Simplemente imprime el mensaje "Hola, bienvenido!" cuando se llama a la función. Para llamar a esta función, simplemente escribiríamos `saludar()` en nuestro programa principal.

Recordando los espacios! , No llamaras a la funcion dentro de la funcion si no fuera de ella.

¿Como me doy cuenta si estoy fuera o no de la funcion?.

```
# Definición de la función
def saludar():
    print("Hola, bienvenido!")

# Llamando a la función
saludar()
```

2.1 DECLARACIÓN DE LA FUNCIÓN.

Como puedes observar, `saludar()` no esta a la misma distancia que "print" esta alineado a `def saludar()` dentro de la funcion, ahora para llamar a la funcion creada, esta alineado con "`def`", Esto significa que donde esta `def` es la linea de tu codigo principal.

no te preocupes el editor de codigo que estes usando te ayudara con los espacios.

2.2 PARÁMETROS DE ENTRADA.

Los parámetros de entrada en Python **son valores que se pasan a una función para que esta realice una tarea específica**. Los parámetros **se definen dentro de los paréntesis de la definición de la función** y se utilizan dentro del cuerpo de la función para realizar cálculos o realizar operaciones en ellos.

Los parámetros de entrada en Python **son opcionales**, lo que significa que una función puede tener cero o varios parámetros de entrada. Además, los parámetros **pueden tener valores por defecto o ser opcionales**, lo que significa que pueden ser ignorados al llamar la función si se proporciona un valor por defecto.

2.2 PARÁMETROS DE ENTRADA.

```
def sumar(numero1, numero2):  
    resultado = numero1 + numero2  
    return resultado
```

En este caso, la función se llama "sumar" y toma dos parámetros de entrada: "numero1" y "numero2". Dentro del cuerpo de la función, se realiza una operación para sumar los dos números y se devuelve el resultado.

Luego, puedes llamar a esta función de la siguiente manera:

```
resultado_suma = sumar(2, 3)  
print(resultado_suma) # 5
```

2.2 PARÁMETROS DE ENTRADA.

```
def sumar(numero1, numero2):  
    resultado = numero1 + numero2  
    return resultado
```

```
resultado_suma = sumar(2, 3)  
print(resultado_suma) # 5
```

En este ejemplo, se llama a la función "sumar" proporcionando dos valores como argumentos de entrada: 2 y 3. La función realiza la suma de los dos números y devuelve el resultado, que se almacena en la variable "resultado_suma".

Este ejemplo ilustra cómo los parámetros de entrada en Python se definen dentro de los paréntesis de la definición de la función y se utilizan dentro del cuerpo de la función para realizar cálculos o realizar operaciones en ellos.

2.3 CUERPO DE LA FUNCIÓN.

Dentro del cuerpo de la función, se pueden incluir declaraciones de variables, estructuras de control de flujo como "if" o "for", operaciones aritméticas, llamadas a otras funciones, entre otras cosas.

Además, es importante recordar que las funciones pueden devolver valores mediante la declaración "return".

IMPORTANTE, el return sirve para casi darle finalidad a tu función , ¿por que casi? por que si usas condicionales en este caso el que cerraria tu función seria un else.

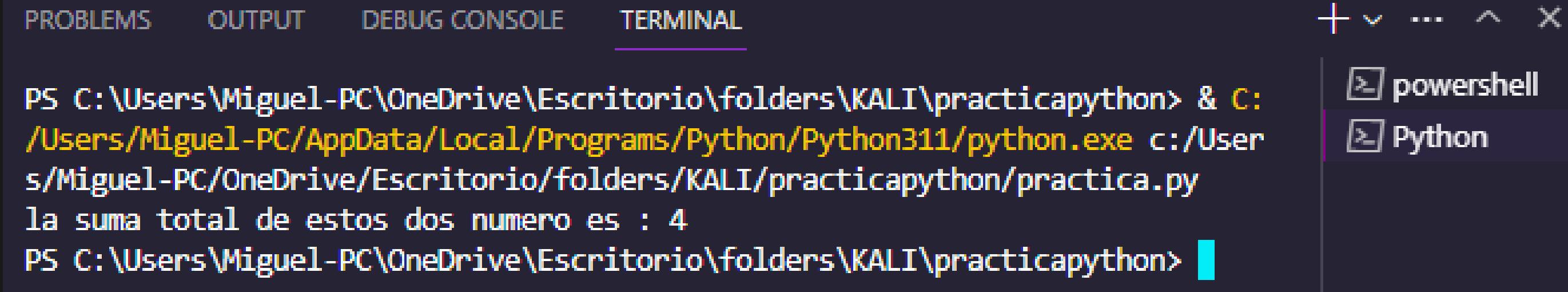
```
def dividir(a, b):
    if b == 0:
        return "Error: no se puede dividir entre cero"
    else:
        return a / b
```

2.4 VALOR DE RETORNO.

Aqui les dejo un ejemplo de VALOR DE RETORNO que pasaria si una funcion no tiene un return a comparacion de que si tenga un return.

```
def suma(num1,num2):
    resultado = num1 + num2
    return resultado

suma_total = suma(2,2)
print(f"la suma total de estos dos numero es : {suma_total}")
```



The screenshot shows a terminal window with the following content:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL + v ... ^ x
PS C:\Users\Miguel-PC\OneDrive\Escritorio\folders\KALI\practicapython> & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python311/python.exe c:/Users/Miguel-PC/OneDrive/Escritorio/folders/KALI/practicapython/practica.py
la suma total de estos dos numero es : 4
PS C:\Users\Miguel-PC\OneDrive\Escritorio\folders\KALI\practicapython>
```

The terminal interface includes tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL, with the TERMINAL tab currently selected. To the right of the terminal window, there is a dropdown menu for selecting a terminal profile, with "Python" highlighted.

Usamos el return para retornar la operacion que hicimos dentro de la funcion , esto nos da el resultado esperado.

2.4 VALOR DE RETORNO.

```
def suma(num1,num2):  
    resultado = num1 + num2  
  
    return resultado  
  
suma_total = suma(2,2)  
print(f"la suma total de estos dos numero es : {suma_total}")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\Miguel-PC\OneDrive\Escritorio\folders\KALI\practicapython> & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python311/python.exe c:/Users/Miguel-PC/OneDrive/Escritorio/folders/KALI/practicapython/practica.py  
la suma total de estos dos numero es : None  
PS C:\Users\Miguel-PC\OneDrive\Escritorio\folders\KALI\practicapython> □
```

Si no usamos el return , no habra nada para retornar,devolver.

DECLARACIÓN Y USO DE FUNCIONES EN PYTHON.

3.1 CREACIÓN DE FUNCIONES PERSONALIZADAS.

Crear funciones personalizadas es una **técnica importante** para escribir código de manera **eficiente**, modular y reutilizable en proyectos de programación. En Python, las funciones personalizadas son útiles para agrupar bloques de código que hacen una tarea específica y se pueden reutilizar en diferentes partes del programa. Al crear una función personalizada, se le da un nombre y un conjunto de parámetros, lo que hace que el código sea más fácil de leer y mantener.

Aquí te muestro un ejemplo sencillo de una función personalizada que calcula la suma de dos números y devuelve el resultado:

3.1 CREACIÓN DE FUNCIONES PERSONALIZADAS.

```
def sumar(a, b):  
    resultado = a + b  
    return resultado  
  
# Llamado a la función y asignación del resultado a una variable  
resultado_suma = sumar(5, 3)  
  
# Impresión del resultado  
print(resultado_suma)
```

En este ejemplo, la función sumar toma dos parámetros (a y b) y los suma para obtener un resultado, que se almacena en la variable resultado. Luego, la función devuelve este resultado usando la declaración return. Finalmente, la función es llamada en el programa principal pasándole los valores de los parámetros 5 y 3, y el resultado se almacena en la variable resultado_suma, que luego se imprime en la consola.

3.1 CREACIÓN DE FUNCIONES PERSONALIZADAS.

Este es un ejemplo muy simple, pero las funciones personalizadas pueden ser mucho más complejas y realizar tareas más sofisticadas. Al crear funciones personalizadas, se puede aumentar la modularidad y la legibilidad del código, lo que facilita la resolución de problemas y el mantenimiento del programa a medida que crece en complejidad.

3.3 EJEMPLOS PRÁCTICOS DE FUNCIONES EN PYTHON.

Ejemplos prácticos

1-Cálculo del área de un rectángulo:

```
def calcular_area_rectangulo(base, altura):  
    return base * altura  
  
area = calcular_area_rectangulo(4, 5)  
print(area) # Imprime 20
```

2-Para calcular el área de un rectángulo, debes multiplicar su base por su altura. La fórmula para el área de un rectángulo es:

$$\text{Área} = \text{base} \times \text{altura}$$

```
def convertir_celsius_a_fahrenheit(celsius):  
    fahrenheit = (celsius * 9/5) + 32  
    return fahrenheit  
  
temperatura_celsius = 25  
temperatura_fahrenheit = convertir_celsius_a_fahrenheit(temperatura_celsius)  
print(temperatura_fahrenheit) # Imprime 77.0
```

3.3 EJEMPLOS PRÁCTICOS DE FUNCIONES EN PYTHON.

Para convertir grados Celsius a grados Fahrenheit, se puede usar la siguiente fórmula:

$$F = (C * 9/5) + 32$$

donde "C" representa la temperatura en grados Celsius, y "F" representa la temperatura en grados Fahrenheit.

3-Función que calcula el área de un círculo dado su radio:

```
def area_circulo(radio):
    area = 3.1416 * radio ** 2
    return area
```

Para calcular el área de un círculo dado su radio, se puede utilizar la siguiente fórmula matemática:

$$\text{área} = \pi * \text{radio}^2$$

Donde "pi" es una constante matemática que se aproxima a 3.14159 y "radio" es la longitud del radio del círculo.

TIPOS DE VARIABLES DENTRO DE LAS FUNCIONES EN PYTHON

4.1 VARIABLES LOCALES.

Las variables locales en una función de Python son aquellas que se definen dentro de la función y solo existen dentro de ella. Esto significa que no se pueden acceder a ellas fuera de la función y no afectan a otras variables con el mismo nombre que puedan existir fuera de la función.

Veamos un ejemplo:

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

4.1 VARIABLES LOCALES.

En esta función, resultado es una variable local ya que se define dentro de la función suma y solo existe dentro de ella. Cuando se llama a la función suma con dos valores para a y b, se realiza la operación de suma y se almacena el resultado en resultado. Luego, la función devuelve este valor y la variable resultado deja de existir.

Es importante tener en cuenta que, si se intenta acceder a la variable resultado fuera de la función suma, se producirá un error ya que no existe fuera de la función.

Las variables locales son útiles para limitar el alcance de una variable y evitar posibles conflictos con variables que puedan tener el mismo nombre en otras partes del programa.

4.2 VARIABLES GLOBALES.

Las variables globales en Python son variables que se definen fuera de una función y pueden ser accedidas desde cualquier parte del programa. Estas variables son útiles para almacenar valores que necesitan ser compartidos entre diferentes funciones.

Es importante tener en cuenta que si se modifica una variable global dentro de una función, la modificación será reflejada en todas las partes del programa que usen esa variable global. Esto puede ser conveniente en algunas situaciones, pero también puede llevar a errores si no se manejan adecuadamente.

Por esta razón, se recomienda tener cuidado al usar variables globales y tratar de mantener su uso al mínimo necesario. En su lugar, se deben utilizar variables locales dentro de las funciones siempre que sea posible, para evitar conflictos y errores en el código.

4.2 VARIABLES GLOBALES.

```
x = 10

def mi_funcion():
    global x
    x += 5
    print("El valor de x dentro de la funcion es :", x)

mi_funcion() #el valor de x dentro de la funcion es : 15
```

En este ejemplo, definimos una variable global x con un valor inicial de 10. Luego, definimos una función mi_funcion() que utiliza la variable global x. Para poder modificar el valor de la variable global dentro de la función, usamos la palabra clave global. Dentro de la función, aumentamos el valor de x en 5 y luego imprimimos el valor actualizado. Finalmente, llamamos a la función y vemos el valor actualizado de x. Después de llamar a la función, imprimimos el valor de x fuera de la función para demostrar que ha sido actualizado.

4.2 VARIABLES GLOBALES.

En resumen, una variable global en una función de Python es una variable que se define fuera de la función y se puede acceder desde cualquier lugar dentro del programa. Cuando se modifica el valor de una variable global dentro de una función, el cambio se reflejará en todas las partes del programa que utilizan esa variable. Es importante tener cuidado al usar variables globales para evitar errores y asegurarse de que el código sea fácil de entender y mantener.

FUNCIONES ANIDADAS Y RECURSIVIDAD EN PYTHON.

5.1 FUNCIONES ANIDADAS.

Las funciones anidadas son simplemente una función dentro de otra función. Esto puede ser útil para separar la lógica del código en módulos más pequeños y manejables. Por ejemplo:

En este ejemplo, `funcion_secundaria` es una función anidada dentro de `funcion_principal`. Al llamar a `funcion_principal`, también se ejecutará la función `funcion_secundaria`.

```
def funcion_principal():
    def funcion_secundaria():
        print("Esta es una función anidada")
    funcion_secundaria()
```

5.1 FUNCIONES ANIDADAS.

En este ejemplo, la función calcular_edad contiene dos funciones anidadas: obtener_anio_actual y calcular_edad_actual.

```
def calcular_edad(nombre, anio_nacimiento):

    def obtener_anio_actual():
        return 2023

    def calcular_edad_actual(anio_actual):
        return anio_actual - anio_nacimiento

    edad = calcular_edad_actual(obtener_anio_actual())
    return f"{nombre} tiene {edad} años"

yo = calcular_edad("juan",1999)
print(yo)
```

La función obtener_anio_actual simplemente retorna el año actual, mientras que la función calcular_edad_actual calcula la edad actual restando el año de nacimiento del año actual.

La función principal, calcular_edad, llama a ambas funciones anidadas para calcular la edad de la persona y devuelve un mensaje con el nombre y la edad.

5.2 FUNCIONES RECURSIVAS.

La recursion en funciones se manda a llamar la misma funcion adentro de la funcion.

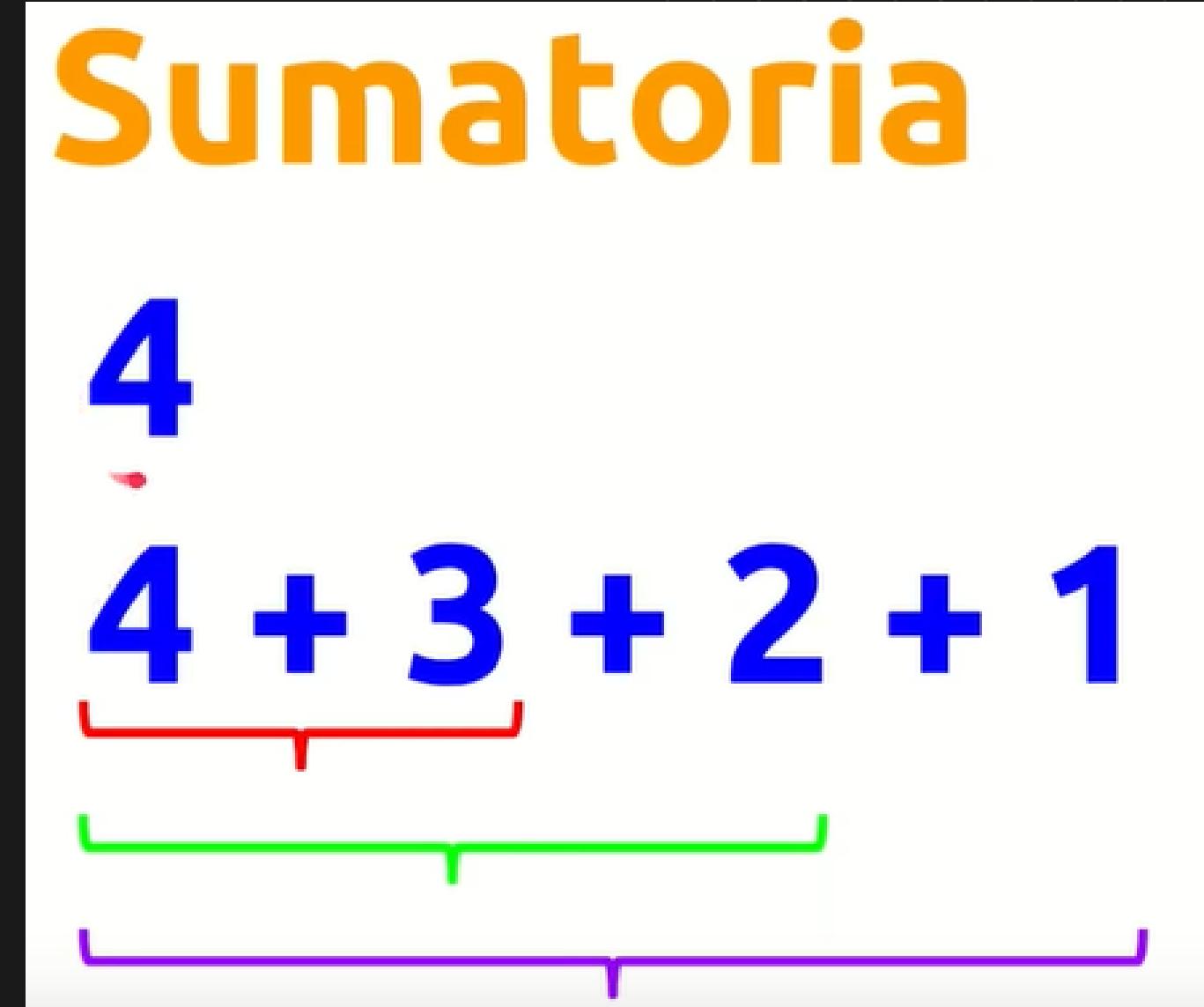
Vamos a trabajar el ejemplo de una sumatoria

Comenzamos escribiendo el 4 y le vamos a ir sumando hacia atras hasta llegar al 1 , este uno sera el encargado de detener la funcion

```
#comprendiendo La recursividad

def sumatoria(num):
    if num==1:
        return 1
    else:
        return num+sumatoria(num-1)

num =int(input("numero de la sumatoria"))
print(sumatoria(num))
```



5.2 FUNCIONES RECURSIVAS.

Explicaremos paso a paso que estamos haciendo.

1-definimos nuestra funcion , en este caso def sumatoria(num): y va a depender de la variable que la llamaremos num.

2-Vamos a poner la condicion que anteriormente mencione,

If ==1:

 return 1

cuando se llegue al 1 la funcion terminara regresara en 1.

3-Despues en caso de que no sea 1 , lo que hara es regresar el numero + la funcion que nombramos como sumatoria y pondremos que numero es -1
Esta resta lo que hara es que vayamos hacia atras sumando los valores , entonces en cuanto llegue al 1 terminara la funcion.

 return num + sumatoria(num-1)

Esto es lo que pareceria como funciona un ciclo while

5.2 FUNCIONES RECURSIVAS.

EJEMPLO PRACTICO

Si yo te preguntara como harias para contar cuantas paginas tienen todos tus libros juntos ,y te pido que me definas todos los pasos de ese proceso.
Tal vez tu solucion seria algo asi como en lo siguiente.

Para encontrar el total de paginas en estos libros

1-voy al primer libro y tomo cuantas paginas tiene "50" y lo guardo en una variable total. Total = 50

2-Ahora voy al segundo y sumo sus paginas al total, que nos quedaria asi
TOTAL = 150 y asi sucesivamente.

Esto se podria hacer
simplemente con un ciclo for.

```
1 libros = [50, 100, 150, 70, 250]
2
3 total = 0
4 for libro in libros:
5     total += libro
6
7 print(total)
```

5.2 FUNCIONES RECURSIVAS.

Pero hay una manera mucho mas facil de hacer mas eficiente, usando la recursividad , por que utilizando el for tenemos que iterar encambio la recursividad lo hace automaticamente. [#comprendiendo La recursividad](#)

```
def suma_naturales(lista):
    # Caso base:
    if len(lista) == 1:
        return lista[0]
    else:
        return lista[0] + suma_naturales(lista[1:])

numero = [50,100,150,200,250]
resultado = suma_naturales(numero)
print(resultado)
```

PROFI

OUTPUT

DEBUG CONSOLE

TERMINAL

```
PS C:\Users\Miguel-PC\OneDrive\Escritorio\folders\KALI\practicapython> & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python311/python.exe c:/Users/Miguel-PC/OneDrive/Escritorio/folders/KALI/practicapython/practica.py  
750  
PS C:\Users\Miguel-PC\OneDrive\Escritorio\folders\KALI\practicapython>
```

FUNCIONES PREDEFINIDAS DE PYTHON.

6.1 USO DE FUNCIONES PREDEFINIDAS.

Las funciones predefinidas en Python son funciones integradas en el lenguaje que puedes utilizar sin tener que definirlas previamente. Estas funciones están disponibles en Python sin necesidad de importar bibliotecas externas. Por ejemplo, la función `len()` se utiliza para obtener la longitud de un objeto, como una cadena de caracteres o una lista. Para utilizar esta función, simplemente la llamas y le pasas el objeto como argumento:

```
cadena = "Hola, mundo!"  
longitud = len(cadena)  
print(longitud)
```

6.1 USO DE FUNCIONES PREDEFINIDAS.

En este ejemplo, la función len() devuelve la longitud de la cadena de caracteres "Hola, mundo!" y la guarda en la variable longitud. Luego, se imprime el valor de longitud en la consola.

Otro ejemplo de función predefinida es sum(), que se utiliza para sumar los elementos de una lista:

```
numeros = [1, 2, 3, 4, 5]
suma = sum(numeros)
print(suma)
```

6.1 USO DE FUNCIONES PREDEFINIDAS.

En este caso, la función `sum()` recibe la lista `numeros` como argumento y devuelve la suma de sus elementos, que se guarda en la variable `suma`. Luego, se imprime el valor de `suma` en la consola.

Existen muchas funciones predefinidas en Python que puedes utilizar para simplificar tus programas. Algunas de las más comunes son `print()`, `input()`, `str()`, `int()`, `float()`, `bool()`, `max()`, `min()`, `range()`, `abs()`, `round()`, entre otras. Es importante mencionar que también puedes definir tus propias funciones en Python para realizar tareas específicas que necesites y poder reutilizarlas en tus programas.

FUNCIONES PREDEFINIDAS DE PYTHON.

7.1 UTILIZACIÓN DE BIBLIOTECAS Y MÓDULOS.

En Python, una biblioteca o módulo es un conjunto de funciones y objetos que se pueden importar en un programa para realizar tareas específicas. Estas bibliotecas están diseñadas para hacer que la programación en Python sea más fácil y eficiente.

Para utilizar una biblioteca o módulo en un programa de Python, primero debe importarse en el archivo de código fuente. Hay varias formas de hacer esto, pero la forma más común es usar la instrucción `import`. Por ejemplo, para utilizar el módulo de matemáticas de Python, puede importarlo en el siguiente código:

```
import math

print(math.pi) # Imprime el valor de pi
```

7.1 UTILIZACIÓN DE BIBLIOTECAS Y MÓDULOS.

En este ejemplo, la biblioteca de matemáticas de Python se importa en el programa con la instrucción "import math". A continuación, se puede acceder a las funciones y constantes del módulo utilizando el prefijo "math.". Hay muchas bibliotecas y módulos disponibles en Python que pueden ser utilizados para realizar una amplia gama de tareas. Algunas bibliotecas populares incluyen:

- Numpy: biblioteca de cálculo numérico que se utiliza para trabajar con matrices y vectores.
- Pandas: biblioteca utilizada para el análisis de datos y la manipulación de datos tabulares.
- Matplotlib: biblioteca de visualización de datos para la creación de gráficos y diagramas.
- Scikit-learn: biblioteca utilizada para el aprendizaje automático y la minería de datos.

7.2 EJEMPLOS PRÁCTICOS DE BIBLIOTECAS Y MÓDULOS EN PYTHON.

En este ejemplo, importamos numpy como np. Luego creamos dos matrices x y y utilizando la función array de numpy. Por último, sumamos ambas matrices utilizando el operador + de numpy y guardamos el resultado en la matriz z. Finalmente, imprimimos z en la consola.

```
import numpy as np  
x = np.array([1, 2, 3])  
y = np.array([4, 5, 6])  
z = x + y  
print(z)
```

En este ejemplo, importamos numpy como np. Luego creamos dos matrices x y y utilizando la función array de numpy. Por último, sumamos ambas matrices utilizando el operador + de numpy y guardamos el resultado en la matriz z. Finalmente, imprimimos z en la consola.

Este es solo un ejemplo muy simple, pero numpy tiene muchas más funciones y características que te permiten realizar cálculos más complejos y sofisticados.

BUENAS PRÁCTICAS EN LA PROGRAMACIÓN DE FUNCIONES EN PYTHON.

8.1 NOMENCLATURA DE FUNCIONES.

La nomenclatura de funciones en Python se refiere a las convenciones de nombrar funciones de manera que sean fáciles de leer y entender para otros programadores.

Algunas reglas generales de la nomenclatura de funciones en Python incluyen:

- Las funciones deben tener nombres en minúsculas y, si el nombre consta de múltiples palabras, estas deben separarse por guiones bajos.
- El nombre de la función debe ser lo más descriptivo posible para indicar su función.
- Se recomienda que el nombre de la función comience con un verbo, ya que las funciones en Python son acciones que realizan una tarea específica.
- Si la función es parte de un módulo, puede ser útil incluir el nombre del módulo como prefijo en el nombre de la función, separado por un guión bajo.

8.1 NOMENCLATURA DE FUNCIONES.

Un ejemplo de función con una buena nomenclatura sería "calcular_precio_total()" que indica claramente la tarea que realiza la función. Es importante seguir estas convenciones para que otros programadores puedan leer y entender fácilmente el código que escribimos.

8.2 DOCUMENTACIÓN DE FUNCIONES.

La documentación de funciones en Python se refiere a la práctica de incluir información adicional sobre la función en el código, para ayudar a los programadores que la utilizan a entender su uso y comportamiento. Esta información adicional puede incluir una descripción de lo que hace la función, los parámetros que acepta, el tipo de valores que espera y los valores que devuelve. La documentación de funciones en Python se hace utilizando "docstrings", que son cadenas de texto que se colocan inmediatamente después de la definición de la función y antes del cuerpo de la función. Estos docstrings pueden ser accesibles mediante el uso de la función `help()` o leyendo el código fuente. Es una buena práctica documentar las funciones en Python para que otros programadores puedan entender cómo se usan y para ayudar en el mantenimiento del código. Además, muchos editores de código y entornos de desarrollo integrados (IDE) pueden mostrar la documentación de la función en una ventana emergente o como parte de una función de autocompletado, lo que facilita el trabajo con las funciones.

A continuación, se presenta un ejemplo sencillo de cómo documentar una función en Python utilizando docstrings:

8.2 DOCUMENTACIÓN DE FUNCIONES.

```
def suma(a, b):
    """
    Esta función toma dos números como argumentos y devuelve su suma.
    Los argumentos deben ser números enteros o de punto flotante.
    """
    return a + b
```

En este ejemplo, se utiliza un docstring de varias líneas para proporcionar información sobre la función. El docstring incluye una descripción de lo que hace la función, los parámetros que acepta y el tipo de valores que espera.

CONCLUSIONES Y RECOMENDACIONES FINALES.

9.1 RECOMENDACIONES PARA EL USO DE FUNCIONES EN PYTHON.

hay varios consejos de programadores profesionales sobre el uso de funciones en Python. Algunos de los más importantes son:

1. Mantener las funciones cortas y simples: las funciones deben hacer una sola cosa y hacerla bien. Si una función es demasiado larga o compleja, puede ser difícil de entender y mantener.
2. Usar nombres de funciones descriptivos: los nombres de las funciones deben ser descriptivos y explicar claramente lo que hace la función.
3. Documentar las funciones: es importante documentar las funciones para que otras personas puedan entender lo que hace la función y cómo se usa.
4. Usar argumentos por defecto: si una función tiene argumentos opcionales, es una buena práctica proporcionar valores predeterminados para estos argumentos. Esto hace que sea más fácil usar la función y reduce la cantidad de código necesario.

9.1 RECOMENDACIONES PARA EL USO DE FUNCIONES EN PYTHON.

- 7.Evitar el uso de variables globales: las variables globales pueden causar problemas de legibilidad y mantenimiento. Es mejor pasar cualquier variable necesaria como argumento a la función.
- 8.Usar el retorno de valores: es importante que las funciones devuelvan un valor si es posible, para que se puedan utilizar en otras partes del programa.
- 9.Usar la modularidad: las funciones deben ser escritas de manera que puedan ser utilizadas en diferentes partes del programa. Esto hace que el código sea más modular y fácil de mantener.
Estos son solo algunos consejos básicos

CLASES



CLASES

INTRODUCCIÓN A LAS CLASES:

Definición de una clase: En Python, una clase es una plantilla para crear objetos. Se define utilizando la palabra clave `class`, seguida del nombre de la clase. Dentro de la clase, se pueden definir atributos (variables) y métodos (funciones) que describen las características y comportamientos de los objetos que se crearán a partir de esa clase.



A screenshot of a code editor window. The title bar says "python". The code area contains the following Python code:

```
class MiClase:  
    pass
```

On the right side of the code area, there is a "Copy code" button with a clipboard icon.

En este caso, `MiClase` es el nombre de la clase que se define utilizando la palabra clave `class`, y `pass` en el ejemplo que te proporcioné, `pass` se utiliza dentro de la clase `MiClase` como un marcador de posición donde normalmente se espera que haya código. Es decir, si no tienes atributos o métodos para agregar en ese momento, puedes utilizar `pass` para evitar un error de sintaxis.

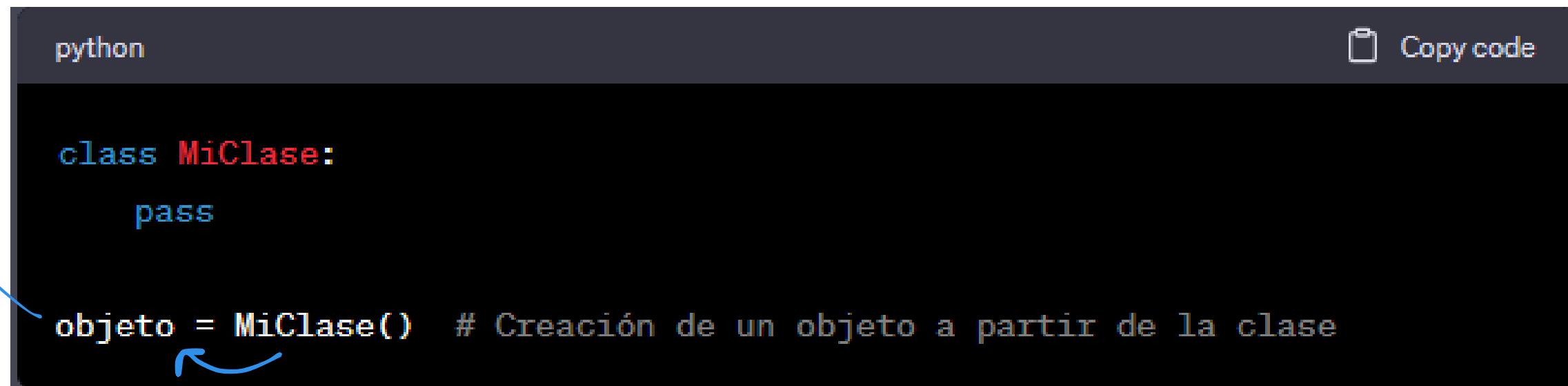
CLASES

INTRODUCCIÓN A LAS CLASES:

Aquí tienes un ejemplo más completo donde se utiliza pass para evitar errores de sintaxis al definir una clase sin atributos o métodos:

objeto

Para crear un objeto simplemente tenemos que crear una variable simple("con el nombre que quieran en este caso le pusimos objeto") como cualquier otra , pero el asignarle una clase ESO lo convierte en un objeto :D



```
python
class MiClase:
    pass

objeto = MiClase() # Creación de un objeto a partir de la clase
```

The screenshot shows a Python code editor window. At the top, it says "python". Below that is a code block. The first line is "class MiClase:". The second line is "pass". The third line is "objeto = MiClase() # Creación de un objeto a partir de la clase". In the top right corner of the code block, there is a "Copy code" button. To the left of the word "objeto" in the main text, there is a blue arrow pointing towards the "objeto" variable in the code block. Another blue arrow points from the explanatory text below to the "objeto" variable in the code block.

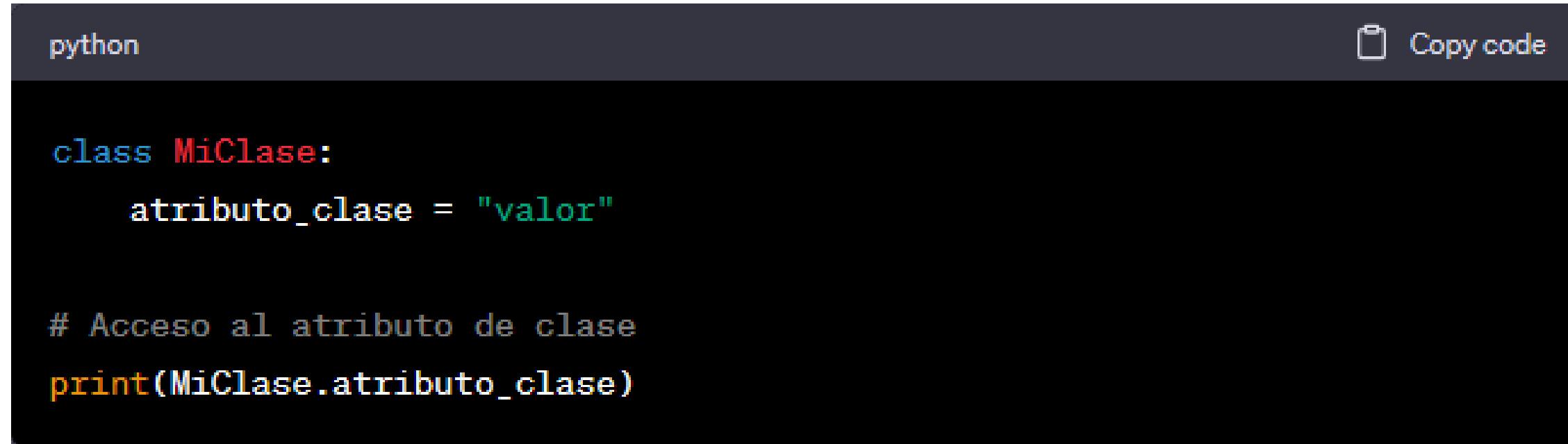
¿Que es eso de crear un objeto?

En este caso, pass se coloca dentro de la clase MiClase para indicar que no hay atributos ni métodos definidos aún. Puedes eliminar la instrucción pass y agregar tus propios atributos y métodos a la clase según sea necesario. En resumen, pass se utiliza para evitar errores de sintaxis cuando necesitas una declaración válida pero no tienes código para escribir en ese momento. No tiene un impacto funcional en el programa, simplemente permite que el código se ejecute sin errores de sintaxis.

CLASES

ATRIBUTOS Y MÉTODOS DE CLASE:

Atributos de clase: Son variables que se definen dentro de una clase y son compartidas por todas las instancias de esa clase. Se definen directamente dentro de la clase, pero fuera de cualquier método. Los atributos de clase son comunes a todos los objetos creados a partir de esa clase y se acceden utilizando la sintaxis `nombre_clase.atributo`.



The image shows a screenshot of a Python code editor. The code is as follows:

```
python

class MiClase:
    atributo_clase = "valor"

# Acceso al atributo de clase
print(MiClase.atributo_clase)
```

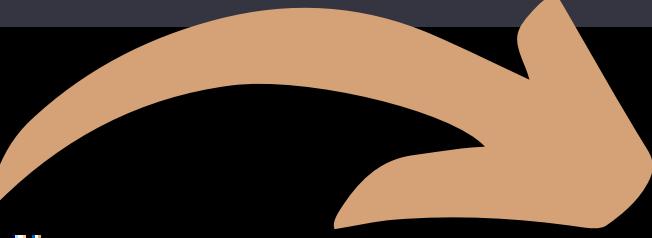
The code defines a class `MiClase` with a class attribute `atributo_clase` set to the value "valor". It then demonstrates accessing this class attribute via the class name `MiClase.atributo_clase`. The code is displayed in a dark-themed editor with syntax highlighting.

En este caso, `atributo_clase` es un atributo de clase que se define dentro de la clase `MiClase`. Puedes acceder a él utilizando la sintaxis `MiClase.atributo_clase`.

CLASES

ATRIBUTOS Y MÉTODOS DE CLASE:

Métodos de clase: Son funciones definidas dentro de una clase y están asociadas a la clase en lugar de a una instancia específica. Aunque normalmente se utiliza self en los métodos de instancia para referirse al objeto actual



```
python
class MiClase:
    @classmethod
    def metodo_clase(self):
        # Código del método de clase
        pass

    # Llamada al método de clase
    MiClase.metodo_clase()

Copy code
```

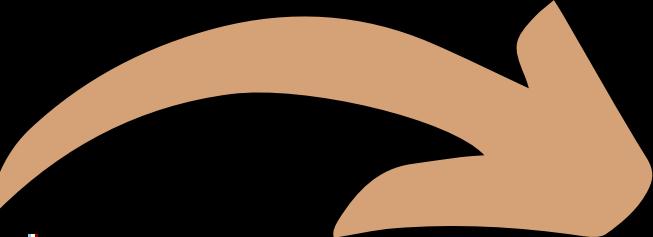
siempre va a ser self ("pero como ejemplo veces ponemos las siglas de classmethod cls")

Métodos de clase: Son funciones definidas dentro de una clase y están asociadas a la clase en lugar de a una instancia específica. Aunque normalmente se utiliza self en los métodos de instancia para referirse al objeto actual

CLASES

ATRIBUTOS Y MÉTODOS DE CLASE:

Llamada a métodos de clase: Los métodos de clase se llaman directamente desde la clase, en lugar de llamarlos desde una instancia o un objeto específico. Se utiliza la sintaxis `nombre_clase.nombre_metodo()` para invocar un método de clase.



```
python
class MiClase:
    @classmethod
    def metodo_clase(cls):
        # Código del método de clase
        pass

    # Llamada al método de clase
    MiClase.metodo_clase()

Copy code
```

cls es para hacer referencia a @classmethod que así se llama la estructura para hacer un método pero siempre como predeterminado se pone self ("no quiero que se confundan")

En este caso, `metodo_clase` es un método de clase definido dentro de la clase `MiClase`. Para llamar al método, utilizamos la sintaxis `MiClase.metodo_clase()`, donde `MiClase` es el nombre de la clase y `metodo_clase` es el nombre del método de clase que queremos invocar.

CLASES

CONSTRUCTOR Y DESTRUCTOR DE CLASE:

Constructor de clase: El constructor de clase es un método especial que se llama automáticamente cuando se crea un objeto de esa clase. Se utiliza para inicializar los atributos de la clase. El constructor de clase se define utilizando el método `_init_()`.

python

Copy code

```
class MiClase:
    def __init__(self, parametro1, parametro2):
        self.atributo1 = parametro1
        self.atributo2 = parametro2

# Creación de un objeto y llamada al constructor
objeto = MiClase(valor1, valor2)
```

En este caso, `_init_()` es el constructor de clase definido dentro de la clase `MiClase`. Toma dos parámetros, `parametro1` y `parametro2`, que se utilizan para inicializar los atributos `atributo1` y `atributo2` respectivamente. Dentro del constructor, utilizamos `self.atributo` para asignar los valores de los parámetros a los atributos de la clase.

Cuando se crea un objeto de la clase utilizando `objeto = MiClase(valor1, valor2)`, se llama automáticamente al constructor de clase y se inicializan los atributos con los valores proporcionados.

Recuerda que el constructor de clase es opcional, pero se utiliza comúnmente para realizar tareas de inicialización necesarias al crear objetos de una clase.

CLASES

CONSTRUCTOR Y DESTRUCTOR DE CLASE:

El constructor de clase no es obligatorio y no todas las clases necesitan tener uno. Sin embargo, se utiliza comúnmente para realizar tareas de inicialización, como asignar valores predeterminados a los atributos, establecer conexiones con bases de datos, abrir archivos, entre otras acciones necesarias para preparar el objeto antes de su uso.

Es importante destacar que el constructor de clase se ejecuta automáticamente cada vez que se crea un objeto de esa clase, y es una parte clave en el proceso de creación de objetos en la programación orientada a objetos.

Cuando se crea una instancia u objeto de una clase utilizando la sintaxis `objeto = Clase()`, Python automáticamente busca el método `_init_()` dentro de la clase y lo ejecuta, pasando el objeto recién creado (`self`) como el primer parámetro implícito. El resto de los parámetros que se pasen al crear el objeto se pasan como argumentos al método `_init_()`.

CLASES

HERENCIA DE CLASES:

Herencia: La herencia es un concepto fundamental en la programación orientada a objetos que permite crear nuevas clases basadas en clases existentes. Permite la creación de una relación de "es-un" entre las clases, donde una clase derivada hereda los atributos y métodos de una clase base.

Clase base y clase derivada: En la herencia, la clase original se conoce como "clase base" o "clase padre", mientras que la nueva clase creada se llama "clase derivada" o "clase hija". La clase derivada hereda todos los atributos y métodos de la clase base y puede agregar nuevos atributos y métodos o modificar los existentes.

Jerarquía de clases: La herencia puede formar una jerarquía de clases, donde las clases derivadas pueden a su vez ser utilizadas como clases base para crear nuevas clases. Esto permite la creación de una estructura de clases organizada y modular.

```
class ClaseBase:
    def metodo_base(self):
        # Implementación del método base

class ClaseDerivada(ClaseBase):
    def metodo_derivado(self):
        # Implementación del método derivado

# Creación de objetos
objeto_base = ClaseBase()
objeto_derivado = ClaseDerivada()

# Llamada a métodos
objeto_base.metodo_base()
objeto_derivado.metodo_base()
objeto_derivado.metodo_derivado()
```

CLASES

HERENCIA DE CLASES:

En este caso, la clase ClaseDerivada hereda de la clase ClaseBase, lo que significa que obtiene el método `metodo_base()` de la clase base. La clase derivada también puede agregar su propio método, en este caso `metodo_derivado()`.

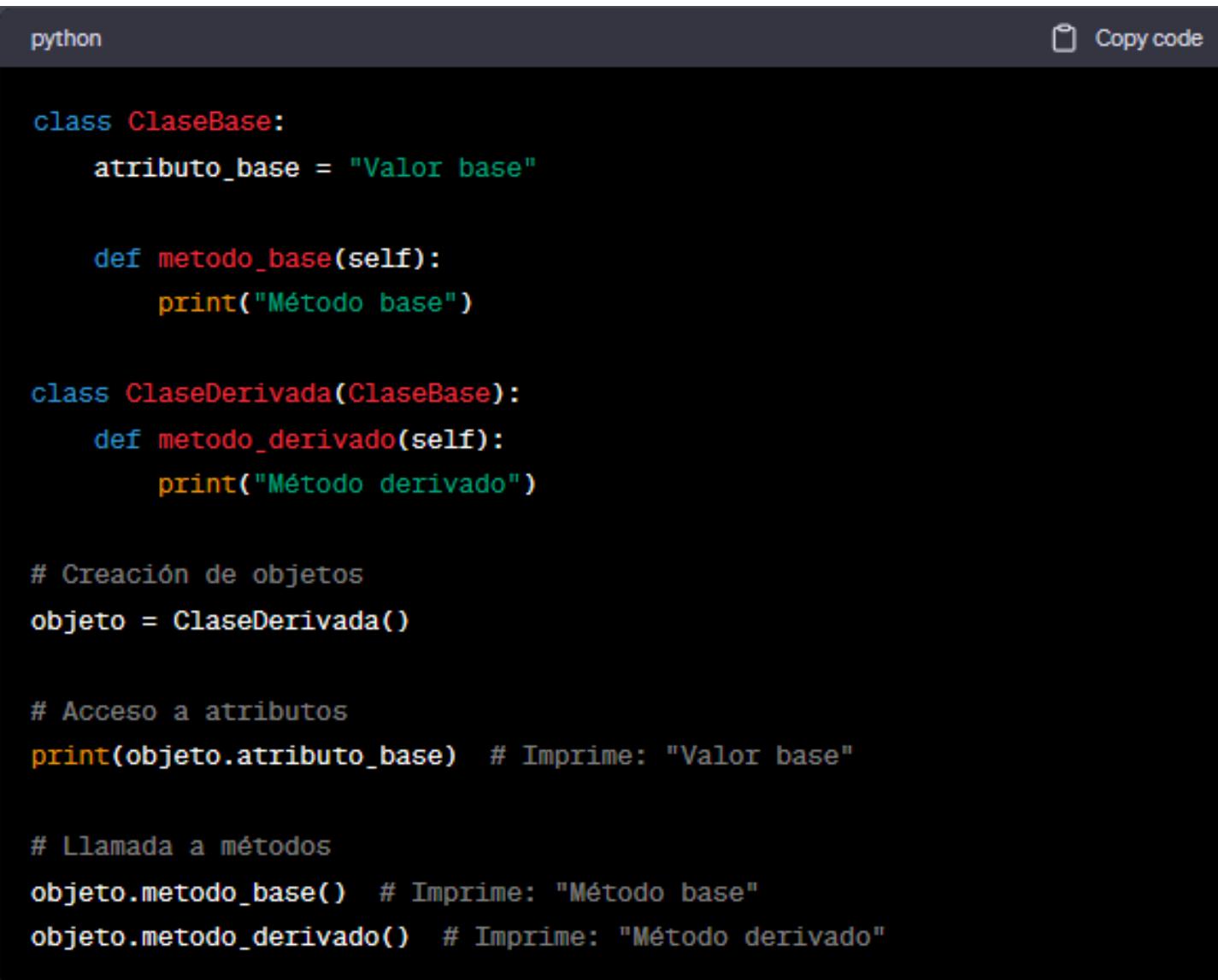
La jerarquía de clases puede ser más compleja, con múltiples niveles de herencia y clases derivadas que se convierten en clases base para nuevas clases.

La herencia y la jerarquía de clases permiten la reutilización de código, la organización de las clases y la implementación de relaciones lógicas entre ellas.

CLASES

HERENCIA DE CLASES:

- **Herencia de atributos:** Al utilizar la herencia en Python, una clase derivada hereda los atributos de la clase base. Esto significa que la clase derivada puede acceder y utilizar los mismos atributos que la clase base, sin necesidad de volver a definirlos. Los atributos de la clase base se convierten en atributos de la clase derivada.
- **Herencia de métodos:** De manera similar, la herencia también permite a la clase derivada heredar los métodos de la clase base. Esto significa que la clase derivada puede utilizar y llamar a los métodos definidos en la clase base, sin necesidad de volver a escribir su implementación.



A screenshot of a Python code editor window titled "python". The code defines two classes: ClaseBase and ClaseDerivada. ClaseBase has an attribute "atributo_base" and a method "metodo_base". ClaseDerivada inherits from ClaseBase and adds its own method "metodo_derivado". An object of ClaseDerivada is created and its attributes and methods are demonstrated.

```
python

class ClaseBase:
    atributo_base = "Valor base"

    def metodo_base(self):
        print("Método base")

class ClaseDerivada(ClaseBase):
    def metodo_derivado(self):
        print("Método derivado")

# Creación de objetos
objeto = ClaseDerivada()

# Acceso a atributos
print(objeto.atributo_base) # Imprime: "Valor base"

# Llamada a métodos
objeto.metodo_base() # Imprime: "Método base"
objeto.metodo_derivado() # Imprime: "Método derivado"
```

En este ejemplo, la clase ClaseDerivada hereda el atributo `atributo_base` y el método `metodo_base` de la clase ClaseBase. Al crear un objeto de la clase derivada (objeto), podemos acceder al atributo y llamar a los métodos tanto de la clase base como de la clase derivada. La herencia de atributos y métodos permite la reutilización de código y facilita la implementación de relaciones lógicas entre clases.

CLASES

HERENCIA DE CLASES:

- **Uso de super()**: En la programación orientada a objetos, super() es una función incorporada que se utiliza para llamar y acceder a los métodos de la clase padre (clase base) desde una clase derivada (clase hija).
- **Acceso a métodos de la clase padre**: Al utilizar super(), puedes llamar a los métodos de la clase padre y utilizarlos en la implementación de la clase derivada. Esto es útil cuando deseas agregar o modificar el comportamiento de un método de la clase padre sin tener que volver a escribir todo su código.

```
python
Copy code

class ClasePadre:
    def metodo_padre(self):
        print("Método de la clase padre")

class ClaseHija(ClasePadre):
    def metodo_hijo(self):
        super().metodo_padre() # Llamada al método de la clase padre
        print("Método de la clase hija")

# Creación de un objeto de la clase hija
objeto = ClaseHija()

# Llamada a métodos
objeto.metodo_hijo()
```

En este ejemplo, la clase ClaseHija hereda de la clase ClasePadre y define su propio método metodo_hijo(). Dentro de metodo_hijo(), se utiliza super().metodo_padre() para llamar al método metodo_padre() de la clase padre y se imprime su mensaje. Luego, se imprime el mensaje del método metodo_hijo(). El uso de super() te permite aprovechar la implementación existente de la clase padre y extenderla o modificarla según sea necesario en la clase hija.

CLASES

MÉTODOS ESPECIALES (MÉTODOS MÁGICOS)

Métodos especiales: En Python, los métodos especiales son aquellos que tienen nombres predefinidos y se utilizan para realizar operaciones específicas en una clase. Estos métodos son invocados automáticamente en ciertas situaciones y proporcionan una funcionalidad especial a la clase.

Sintaxis: Los métodos especiales se definen utilizando una sintaxis especial con doble guion bajo (`_`). Por ejemplo, `__init__()` es el método especial utilizado para la inicialización de objetos, y se llama automáticamente al crear un nuevo objeto de la clase.

- **Funcionalidades:** Algunos ejemplos de métodos especiales comunes son:
- `__init__()`: Método de inicialización, se ejecuta al crear un objeto.
- `__str__()`: Método que devuelve una representación en cadena del objeto cuando se llama a la función `str(objeto)`.
- `__len__()`: Método que devuelve la longitud del objeto cuando se llama a la función `len(objeto)`.

```
python

class MiClase:
    def __init__(self, valor):
        self.valor = valor

    def __str__(self):
        return f"MiClase({self.valor})"

# Creación de un objeto
objeto = MiClase(42)

# Llamada al método especial __str__()
print(objeto) # Imprime: MiClase(42)
```

En este ejemplo, se define el método especial `__str__()` en la clase `MiClase`. Este método se ejecuta cuando se llama a la función `str()` en un objeto de la clase. El método devuelve una cadena que representa el objeto en un formato específico.

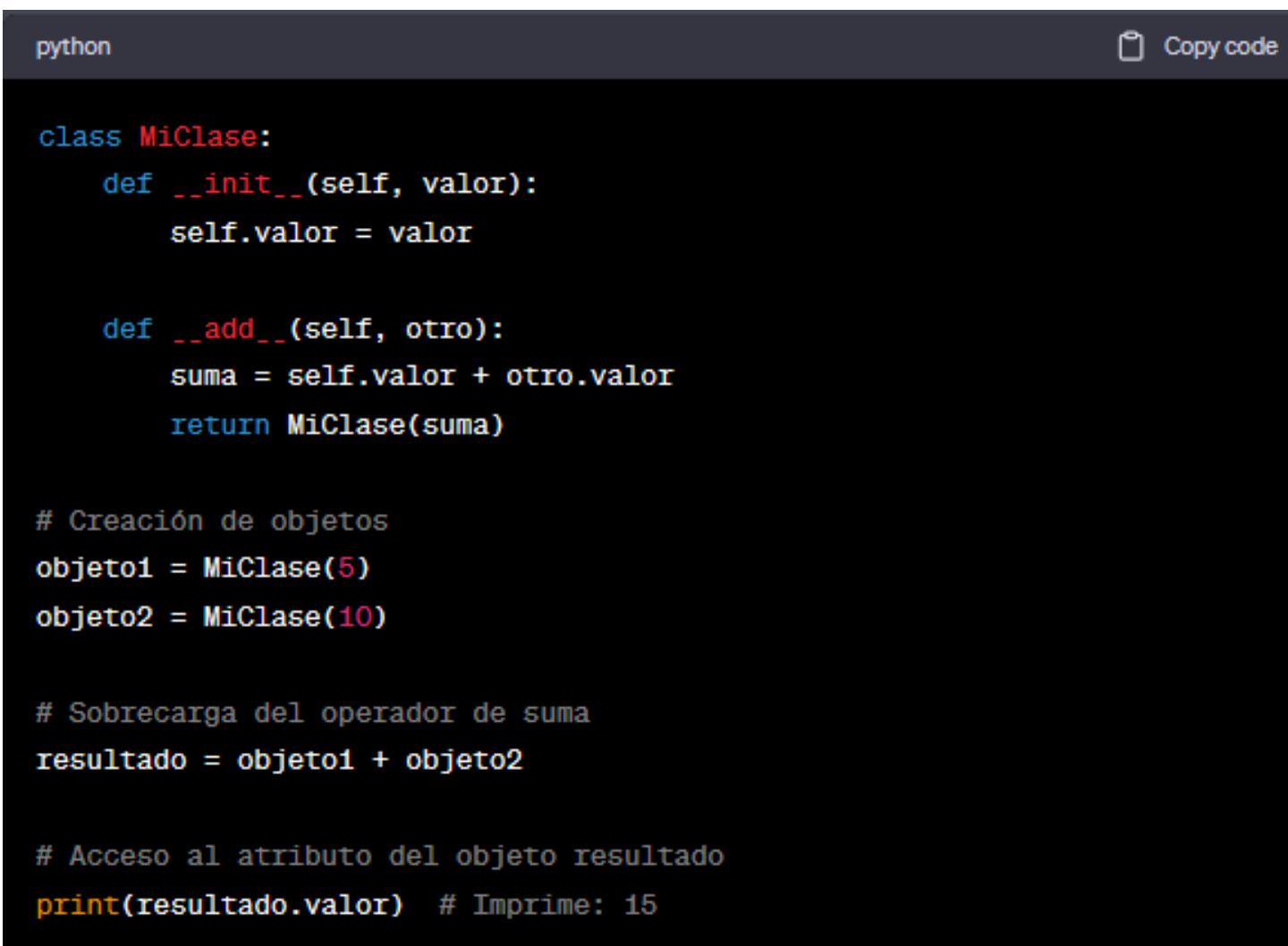
Los métodos especiales te permiten personalizar el comportamiento de las operaciones básicas en una clase y mejorar la usabilidad y legibilidad del código.

CLASES

MÉTODOS ESPECIALES (MÉTODOS MÁGICOS)

Sobrecarga de operadores: En Python, puedes sobrecargar los operadores incorporados, como la suma (+), resta (-), multiplicación (*), etc., utilizando métodos especiales. Estos métodos especiales permiten definir el comportamiento de los operadores cuando se aplican a objetos de una clase personalizada.

Modificación de comportamientos: Al sobrecargar los operadores utilizando métodos especiales, puedes modificar los comportamientos predeterminados y definir cómo los objetos de tu clase responden a los operadores. Por ejemplo, puedes definir la suma de dos objetos de tu clase para que realice una operación específica.



```
python

class MiClase:
    def __init__(self, valor):
        self.valor = valor

    def __add__(self, otro):
        suma = self.valor + otro.valor
        return MiClase(suma)

# Creación de objetos
objeto1 = MiClase(5)
objeto2 = MiClase(10)

# Sobrecarga del operador de suma
resultado = objeto1 + objeto2

# Acceso al atributo del objeto resultado
print(resultado.valor) # Imprime: 15
```

En este ejemplo, se define el método especial `__add__()` en la clase `MiClase` para sobrecargar el operador de suma (`+`). Este método se llama automáticamente cuando se aplica el operador de suma a objetos de la clase. En este caso, se realiza una suma de los atributos `valor` de los objetos y se devuelve un nuevo objeto de `MiClase` con el resultado.

Al sobrecargar los operadores con métodos especiales, puedes personalizar el comportamiento de tu clase y hacer que los objetos respondan a los operadores de la manera que deseas.

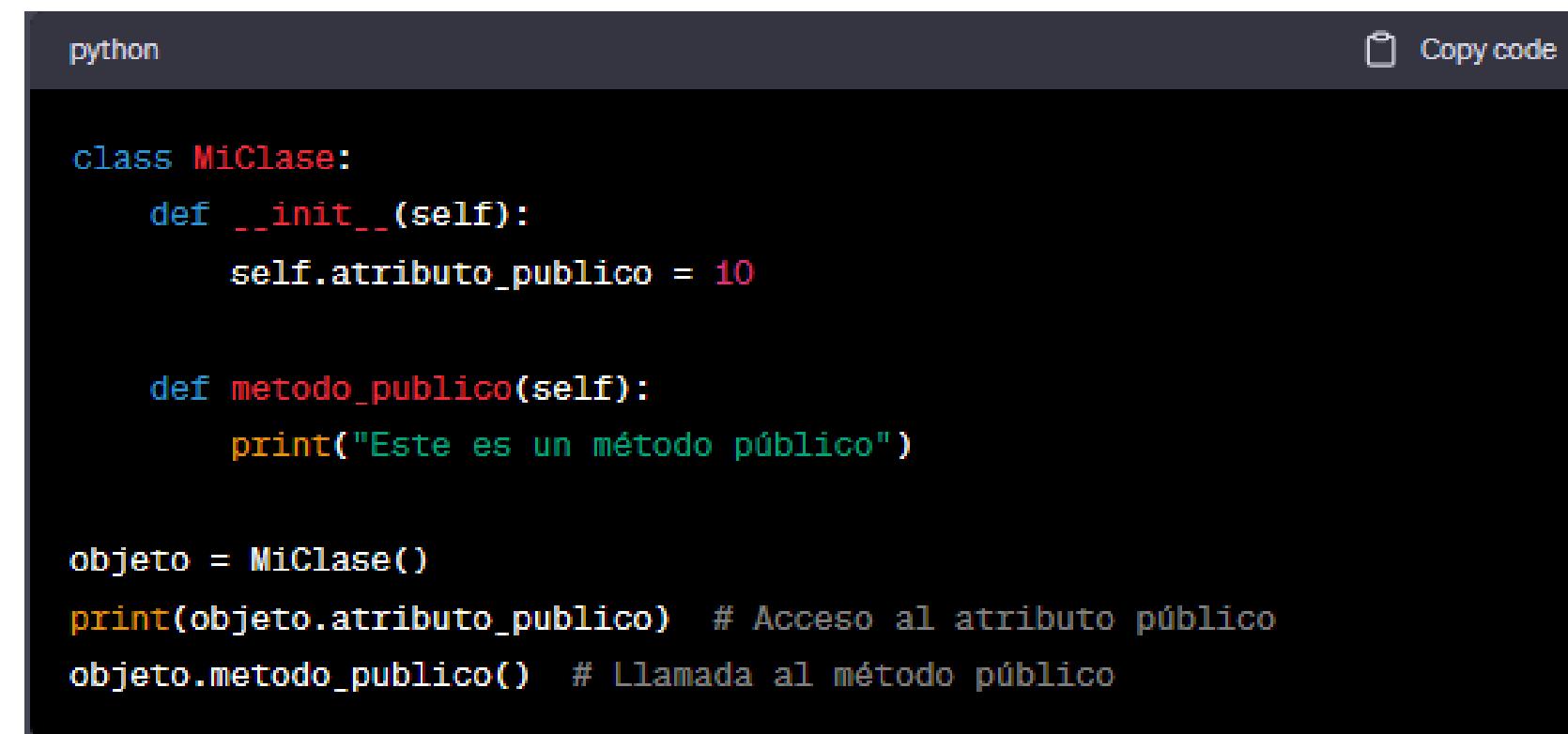
CLASES

ENCAPSULACIÓN Y CONTROL DE ACCESO:

Encapsulación: La encapsulación es un concepto de la programación orientada a objetos que consiste en agrupar los datos (atributos) y las operaciones (métodos) relacionadas en una sola entidad llamada clase. Esto permite ocultar la implementación interna de la clase y proporcionar una interfaz clara y definida para interactuar con ella.

Ocultación de información: La ocultación de información es un aspecto clave de la encapsulación. Significa que los detalles internos de una clase, como los atributos y métodos privados, no son accesibles desde fuera de la clase. Esto se logra utilizando la convención de nomenclatura y el control de acceso en Python.

Public: En Python, por convención, todos los atributos y métodos se consideran públicos por defecto. Esto significa que se pueden acceder y utilizar desde cualquier parte del programa, ya sea dentro de la clase o desde fuera de ella.



A screenshot of a Python code editor window titled "python". The code defines a class named "MiClase" with an __init__ method that initializes a public attribute "atributo_publico" to 10. It also contains a public method "metodo_publico" that prints a message. An object "objeto" is created from the class, and its attribute "atributo_publico" is printed, followed by a call to its method "metodo_publico". A "Copy code" button is visible in the top right corner of the editor.

```
python

class MiClase:
    def __init__(self):
        self.atributo_publico = 10

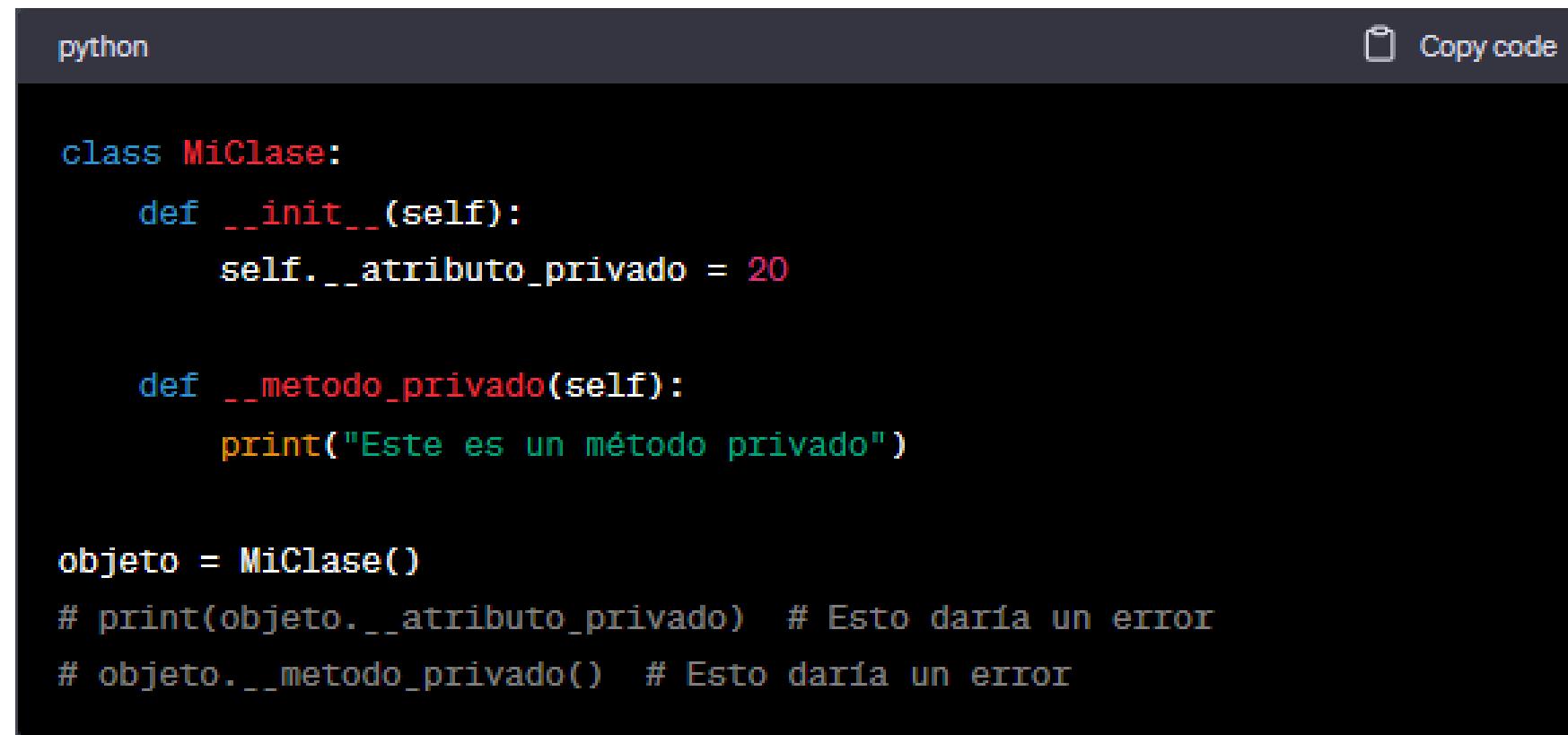
    def metodo_publico(self):
        print("Este es un método público")

objeto = MiClase()
print(objeto.atributo_publico) # Acceso al atributo público
objeto.metodo_publico() # Llamada al método público
```

CLASES

ENCAPSULACIÓN Y CONTROL DE ACCESO:

Private: En Python, se utiliza una convención de nomenclatura para indicar que un atributo o método es privado. Se nombra utilizando dos guiones bajos (`_`) al principio del nombre. Sin embargo, en realidad, la privacidad en Python es solo una convención y no una restricción real. Los atributos y métodos privados se pueden acceder desde fuera de la clase, aunque no se recomienda hacerlo.



The screenshot shows a code editor window with a dark theme. The title bar says "python". The code area contains the following Python code:

```
python

class MiClase:
    def __init__(self):
        self.__atributo_privado = 20

    def __metodo_privado(self):
        print("Este es un método privado")

objeto = MiClase()
# print(objeto.__atributo_privado) # Esto daría un error
# objeto.__metodo_privado() # Esto daría un error
```

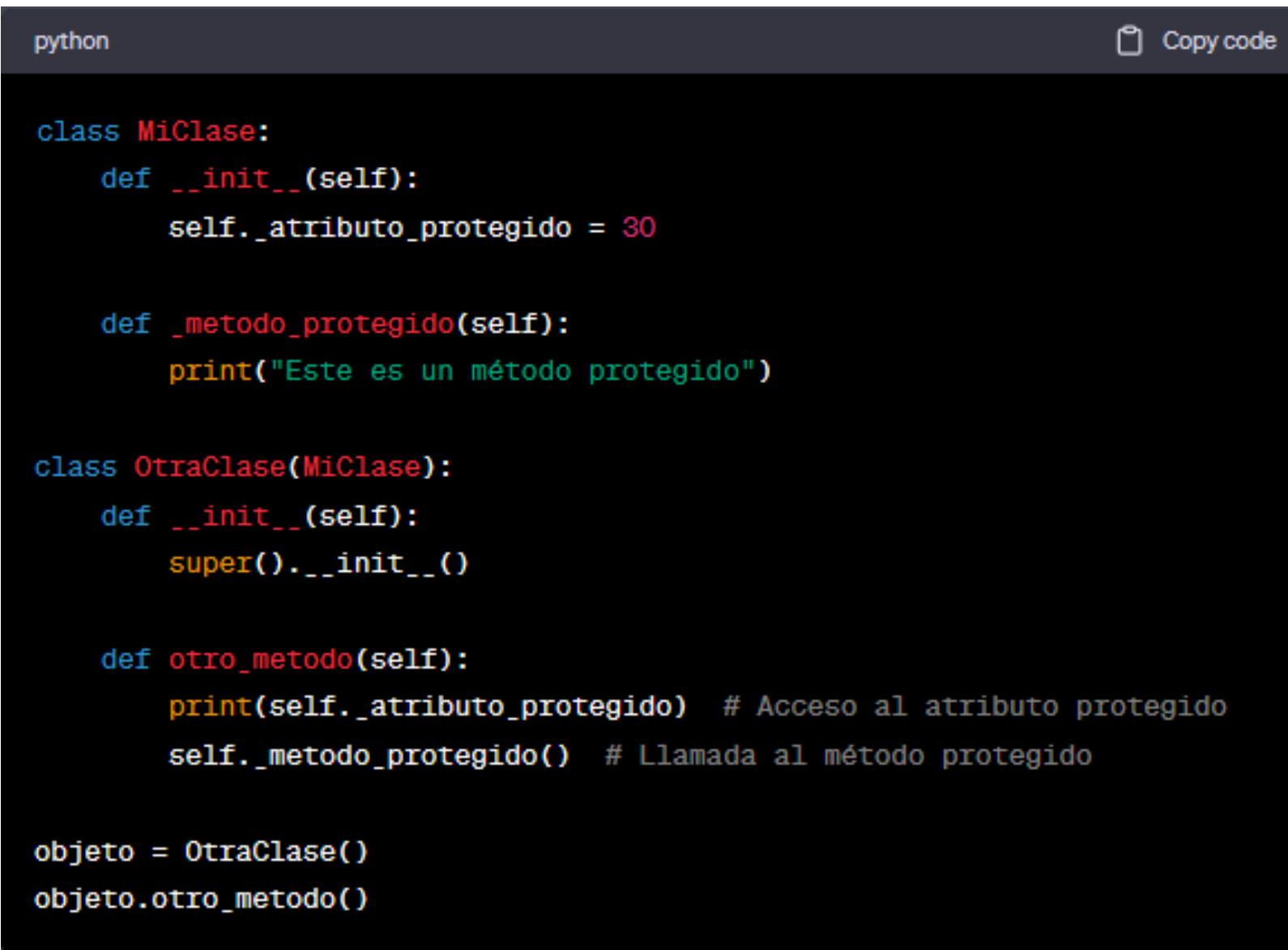
On the right side of the code area, there is a "Copy code" button with a clipboard icon.

Aunque es posible acceder a los atributos y métodos privados utilizando el nombre real, se recomienda respetar la convención y no hacerlo.

CLASES

ENCAPSULACIÓN Y CONTROL DE ACCESO:

Protected: En Python, se utiliza una convención de nomenclatura para indicar que un atributo o método es protegido. Se nombra utilizando un guion bajo (_) al principio del nombre. Al igual que con los atributos privados, los atributos protegidos también se pueden acceder desde fuera de la clase, pero se considera una buena práctica no hacerlo.



A screenshot of a Python code editor window titled "python". The code demonstrates encapsulation and access control. It defines two classes: "MiClase" and "OtraClase". "MiClase" has a protected attribute "_atributo_protegido" initialized to 30 in its constructor. It also has a protected method "_metodo_protegido" that prints a message. "OtraClase" inherits from "MiClase". Its constructor calls the constructor of "MiClase". It has a public method "otro_metodo" that prints the value of the protected attribute and calls the protected method. An instance of "OtraClase" is created and its "otro_metodo" method is called, demonstrating that protected members can be accessed from outside the class.

```
python

class MiClase:
    def __init__(self):
        self._atributo_protegido = 30

    def _metodo_protegido(self):
        print("Este es un método protegido")

class OtraClase(MiClase):
    def __init__(self):
        super().__init__()

    def otro_metodo(self):
        print(self._atributo_protegido) # Acceso al atributo protegido
        self._metodo_protegido() # Llamada al método protegido

objeto = OtraClase()
objeto.otro_metodo()
```

CLASES

ENCAPSULACIÓN Y CONTROL DE ACCESO:

Getter: Un getter es un método que se utiliza para obtener el valor de un atributo privado o protegido de una clase. Proporciona acceso de solo lectura al atributo. El nombre comúnmente utilizado para el getter es `get_nombre_atributo`. Aquí tienes un ejemplo:



```
Enseñando.py > ...
Enseñando.py > ...
1  class MiClase:
2      def __init__(self):
3          self.__atributo_privado = 10
4
5      def get_atributo_privado(self):
6          return self.__atributo_privado
7
8      objeto = MiClase()
9      valor = objeto.get_atributo_privado() # Acceso al atributo privado a través del getter
10     print(valor) # Imprime: 10
11

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
TERMINAL
● PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python311/python.exe c:/Users/Drive/Escritorio/folders/KALI/PYTHON/Enseñando.py
10
○ Miguel-PC ▶ □ PYTHON ▶ □ □ in pws
```

CLASES

ENCAPSULACIÓN Y CONTROL DE ACCESO:

Setter: Un setter es un método que se utiliza para modificar el valor de un atributo privado o protegido de una clase. Proporciona acceso de escritura al atributo. El nombre comúnmente utilizado para el setter es `set_nombre_atributo`. Aquí tienes un ejemplo:

```
Enseñando.py X
Enseñando.py > ...
1  class MiClase:
2      def __init__(self):
3          self.__atributo_privado = 10
4
5      def set_atributo_privado(self, nuevo_valor):
6          self.__atributo_privado = nuevo_valor
7
8      def get_atributo_privado(self):
9          return self.__atributo_privado
10
11 objeto = MiClase()
12 objeto.set_atributo_privado(20) # Modificación del atributo privado a través del setter
13 valor = objeto.get_atributo_privado() # Acceso al atributo privado actualizado
14 print(valor) # Imprime: 20
15
16
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python311/python.exe c:/Users/Drive/Escritorio/folders/KALI/PYTHON/Enseñando.py
20
- Miguel-PC PYTHON

CLASES

POLIMORFISMO Y ABSTRACCIÓN:

Polimorfismo: El polimorfismo es la capacidad de los objetos de una misma jerarquía de clases de responder de manera diferente a una misma llamada de método. Esto significa que los objetos de diferentes clases pueden implementar un método de manera diferente, pero con la misma interfaz. El polimorfismo permite escribir código genérico que puede trabajar con diferentes tipos de objetos sin conocer su tipo específico. En Python, el polimorfismo se logra mediante el uso de herencia y la sobrescritura de métodos.

Abstracción: La abstracción es un concepto que permite representar características y comportamientos esenciales de un objeto en forma de una clase base o abstracta. Una clase abstracta define la estructura y los métodos comunes que deben ser implementados por las clases derivadas. No se pueden crear instancias de una clase abstracta, sino que se utilizan como plantillas para crear nuevas clases. En Python, la abstracción se implementa utilizando la clase base abstracta y los métodos abstractos.

Implementación de polimorfismo en Python: En Python, el polimorfismo se logra cuando las clases derivadas implementan métodos con el mismo nombre que la clase base, pero con diferentes implementaciones. Esto permite que los objetos de diferentes clases respondan de manera diferente al mismo método. Por ejemplo:

CLASES

POLIMORFISMO Y ABSTRACCIÓN:

```
class Animal:
    def sonido(self):
        pass

class Perro(Animal):
    def sonido(self):
        print("Guau guau")

class Gato(Animal):
    def sonido(self):
        print("Miau miau")

def hacer_sonar(animal):
    animal.sonido()

perro = Perro()
gato = Gato()

hacer_sonar(perro) # Imprime: Guau guau
hacer_sonar(gato) # Imprime: Miau miau
```

CLASES

POLIMORFISMO Y ABSTRACCIÓN:

En Python, se puede utilizar el módulo abc (Abstract Base Classes) para crear clases abstractas y métodos abstractos. Una clase abstracta es aquella que no puede ser instanciada directamente y sirve como base para otras clases. Los métodos abstractos son aquellos que no tienen implementación en la clase abstracta, sino que deben ser implementados en las clases derivadas. Aquí tienes un ejemplo muy resumido de cómo crear una clase abstracta y un método abstracto utilizando el módulo abc:

```
python
from abc import ABC, abstractmethod

class Figura(ABC):
    @abstractmethod
    def calcular_area(self):
        pass

class Rectangulo(Figura):
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

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

rectangulo = Rectangulo(5, 3)
area = rectangulo.calcular_area()
print(area) # Imprime: 15
```

En este ejemplo, la clase Figura es una clase abstracta que define el método abstracto calcular_area(). La clase Rectangulo hereda de la clase Figura y proporciona una implementación concreta del método calcular_area(). Al crear una instancia de la clase Rectangulo y llamar al método calcular_area(), se ejecutará la implementación específica de la clase Rectangulo. La creación de clases abstractas y métodos abstractos permite definir una interfaz común para varias clases relacionadas y garantizar que las clases derivadas implementen los métodos necesarios.

CLASES

CLASES Y MÓDULOS:

Organización de clases en módulos y paquetes: En Python, las clases se pueden organizar en módulos y paquetes para una mejor estructura y organización del código. Un módulo es un archivo Python que contiene definiciones de clases, funciones y variables. Un paquete es una carpeta que contiene uno o más módulos, y puede tener subpaquetes. Los módulos y paquetes ayudan a dividir el código en componentes lógicos y promueven la reutilización y la modularidad.

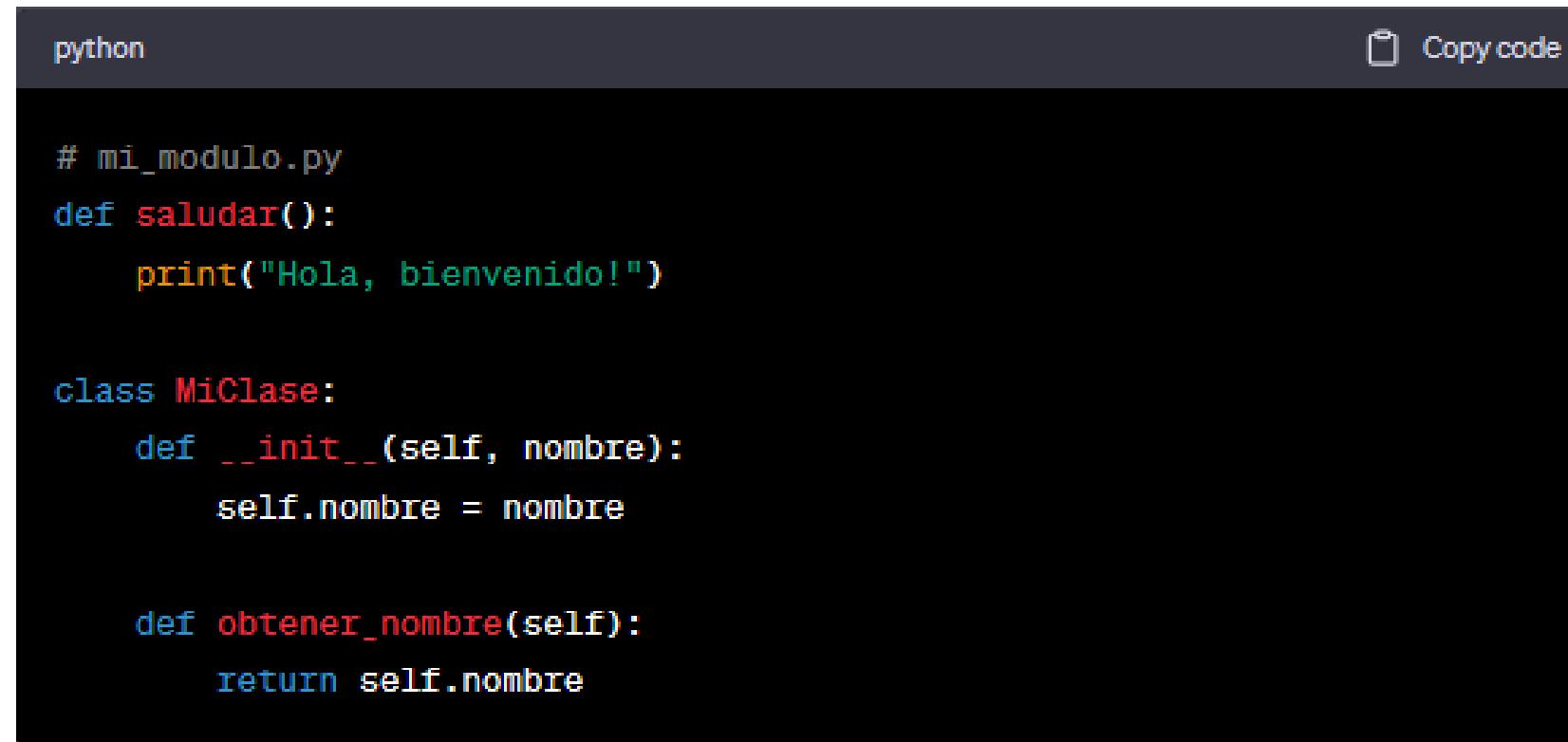
Importación de clases desde otros módulos: Para utilizar una clase definida en otro módulo, debes importarla en el módulo actual. Puedes importar todo el módulo o importar una clase específica. La sintaxis básica de importación es `import nombre_modulo` o `from nombre_modulo import nombre_clase`.

Uso de alias y importación selectiva de clases: Puedes utilizar un alias para darle a un módulo o clase un nombre diferente al importarlo. Esto es útil para evitar conflictos de nombres o para hacer el código más legible. Además, puedes realizar una importación selectiva para importar solo las clases o funciones que necesitas de un módulo, en lugar de importar todo el módulo. La sintaxis para importar con alias y selectivamente es `import nombre_modulo as alias` y `from nombre_modulo import nombre_clase`.

CLASES

CLASES Y MÓDULOS:

Importación del módulo completo: Supongamos que tienes un archivo llamado mi_modulo.py que contiene algunas definiciones de clase y funciones:



```
python
Copy code

# mi_modulo.py
def saludar():
    print("Hola, bienvenido!")

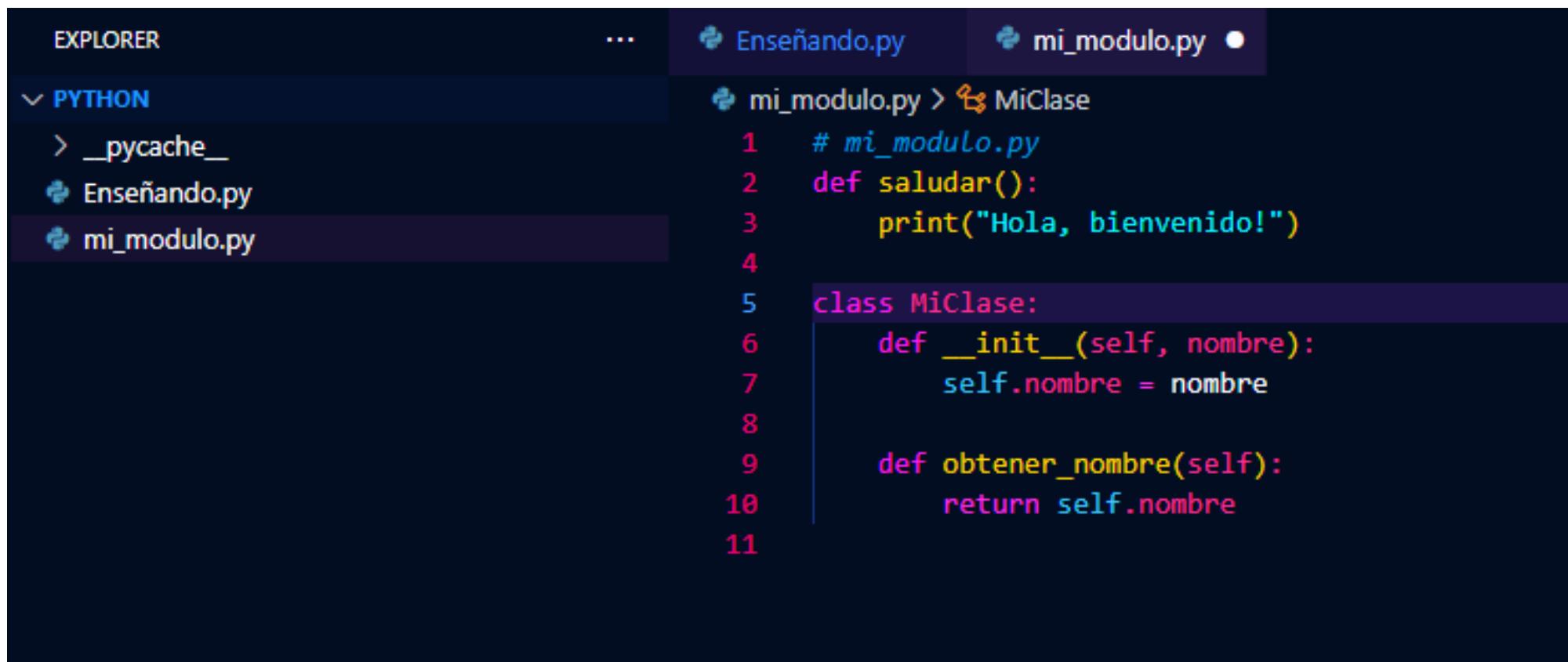
class MiClase:
    def __init__(self, nombre):
        self.nombre = nombre

    def obtener_nombre(self):
        return self.nombre
```

Para importar todo el módulo mi_modulo en otro archivo, puedes utilizar la siguiente sintaxis:

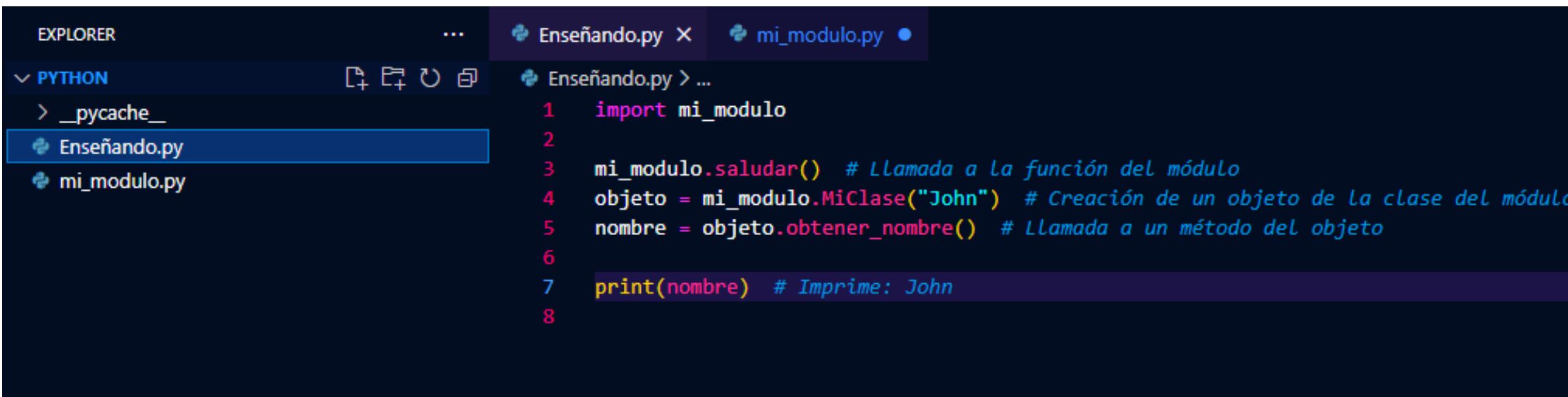
CLASES

CLASES Y MÓDULOS:



The screenshot shows a dark-themed code editor interface. In the top right, there are tabs for "Enseñando.py" and "mi_modulo.py". The "mi_modulo.py" tab is active. Below the tabs, the file content is displayed:

```
1 # mi_modulo.py
2 def saludar():
3     print("Hola, bienvenido!")
4
5 class MiClase:
6     def __init__(self, nombre):
7         self.nombre = nombre
8
9     def obtener_nombre(self):
10        return self.nombre
11
```



The screenshot shows a dark-themed code editor interface. In the top right, there are tabs for "Enseñando.py" and "mi_modulo.py". The "Enseñando.py" tab is active. Below the tabs, the file content is displayed:

```
1 import mi_modulo
2
3 mi_modulo.saludar() # Llamada a la función del módulo
4 objeto = mi_modulo.MiClase("John") # Creación de un objeto de la clase del módulo
5 nombre = objeto.obtener_nombre() # Llamada a un método del objeto
6
7 print(nombre) # Imprime: John
8
```

CLASES

CLASES Y MÓDULOS:

The screenshot shows a Python development environment in a dark-themed code editor. The left sidebar (EXPLORER) lists files: __pycache__, Enseñando.py (selected), and mi_modulo.py. The main area (EDITOR) contains the following code:

```
1 import mi_modulo
2
3 mi_modulo.saludar() # Llamada a la función del módulo
4 objeto = mi_modulo.MiClase("John") # Creación de un objeto de la clase del módulo
5 nombre = objeto.obtener_nombre() # Llamada a un método del objeto
6
7 print(nombre) # Imprime: John
```

The code imports a module named `mi_modulo`, calls its `saludar()` function, creates an object of its `MiClase` class with the name "John", and prints the object's `nombre` attribute.

Below the code editor is a terminal window showing the execution of the script:

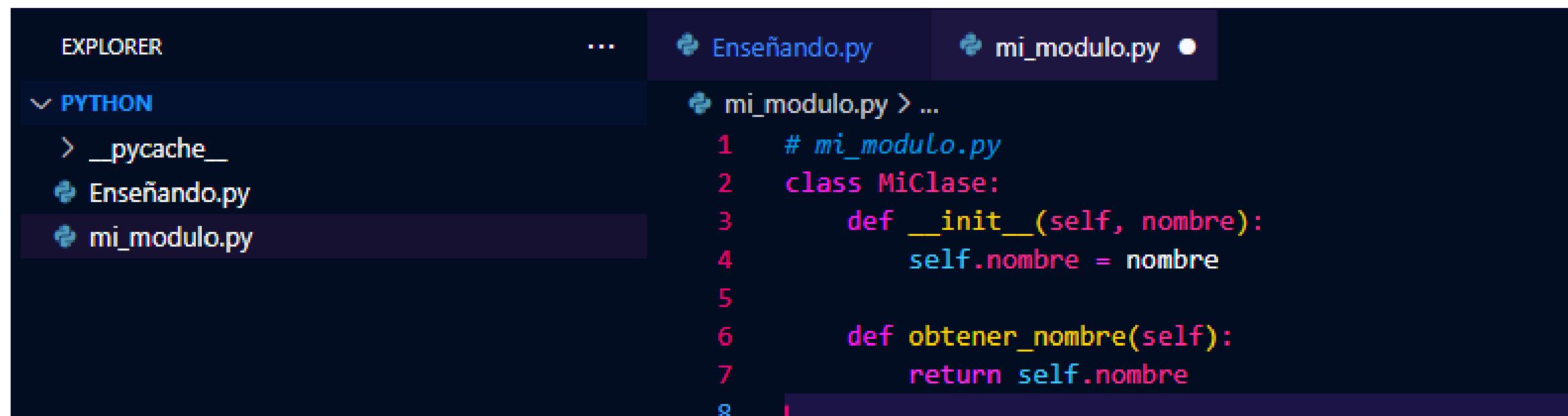
```
PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python311/python.exe c:/Drive/Escritorio/folders/KALI/PYTHON/Enseñando.py
Hola, bienvenido!
John
```

The terminal output shows the script running in a Python environment, printing "Hola, bienvenido!" and then "John".

CLASES

CLASES Y MÓDULOS:

Importación selectiva de una clase :Supongamos que tienes un archivo llamado mi_modulo.py que contiene la definición de una clase llamada MiClase:

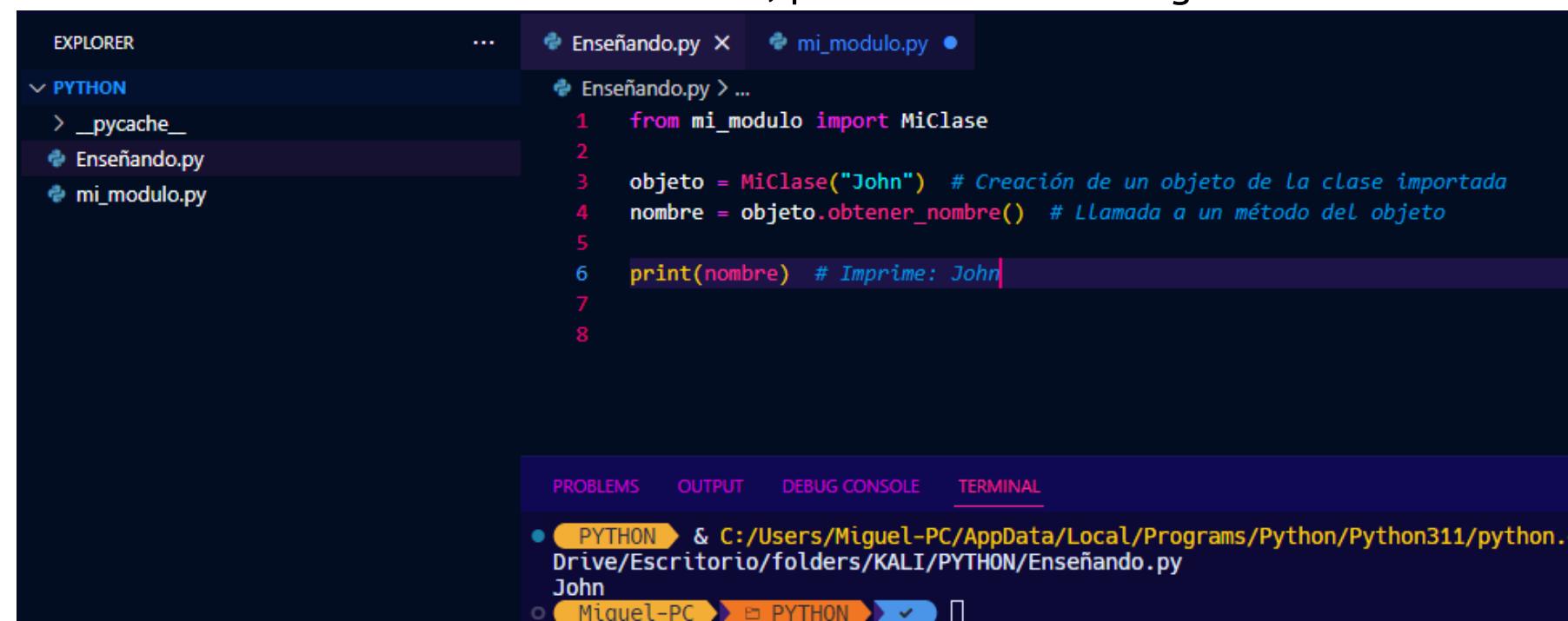


```
EXPLORER ... Enseñando.py mi_modulo.py ●
PYTHON > __pycache__ mi_modulo.py > ...
Enseñando.py
mi_modulo.py

# mi_modulo.py
class MiClase:
    def __init__(self, nombre):
        self.nombre = nombre

    def obtener_nombre(self):
        return self.nombre
```

Para importar selectivamente solo la clase MiClase en otro archivo, puedes utilizar la siguiente sintaxis:



```
EXPLORER ... Enseñando.py x mi_modulo.py ●
PYTHON > __pycache__ Enseñando.py > ...
mi_modulo.py

from mi_modulo import MiClase
objeto = MiClase("John") # Creación de un objeto de la clase importada
nombre = objeto.obtener_nombre() # Llamada a un método del objeto
print(nombre) # Imprime: John
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python311/python.exe
Drive/Escritorio/folders/KALI/PYTHON/Enseñando.py
John

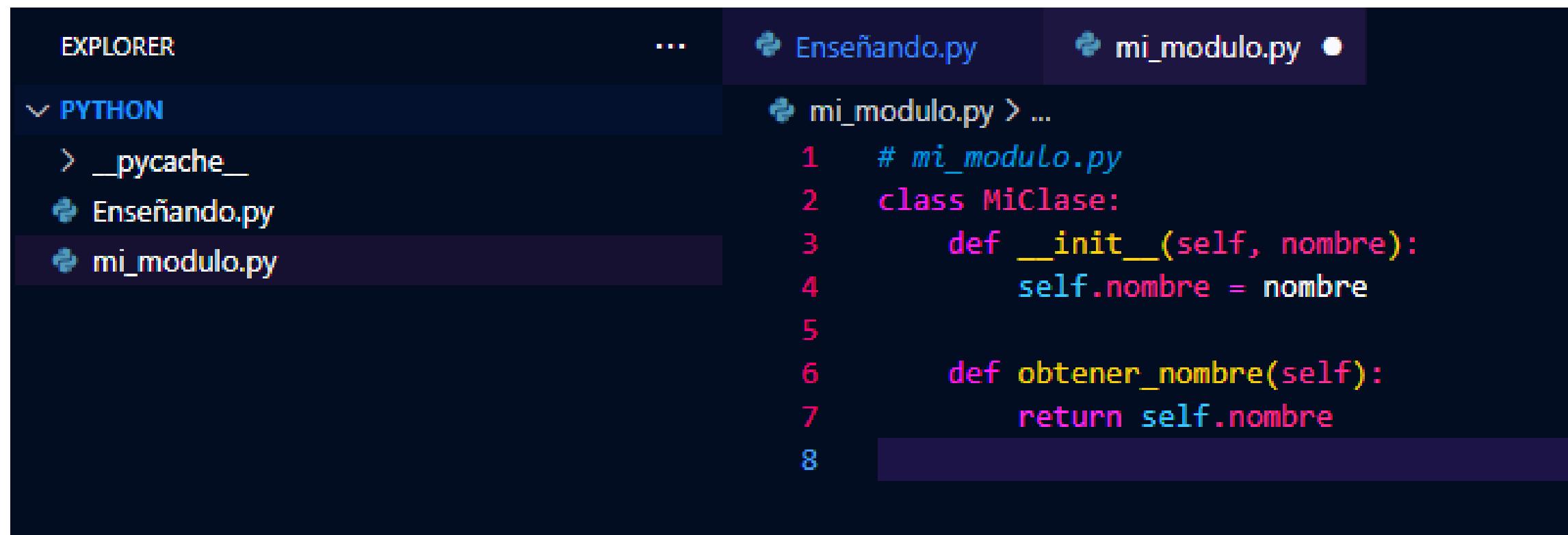
En este ejemplo, se utiliza la instrucción `from mi_modulo import MiClase` para importar solo la clase `MiClase` del módulo `mi_modulo`. Esto permite acceder directamente a la clase sin tener que especificar el nombre del módulo. Luego, se crea un objeto de la clase `MiClase` utilizando el constructor `MiClase()` y se llama al método `obtener_nombre()` del objeto. Finalmente, se imprime el resultado.

Ten en cuenta que al utilizar una importación selectiva, solo puedes acceder a la clase o función que has importado y no a otras definiciones dentro del módulo.

CLASES

CLASES Y MÓDULOS:

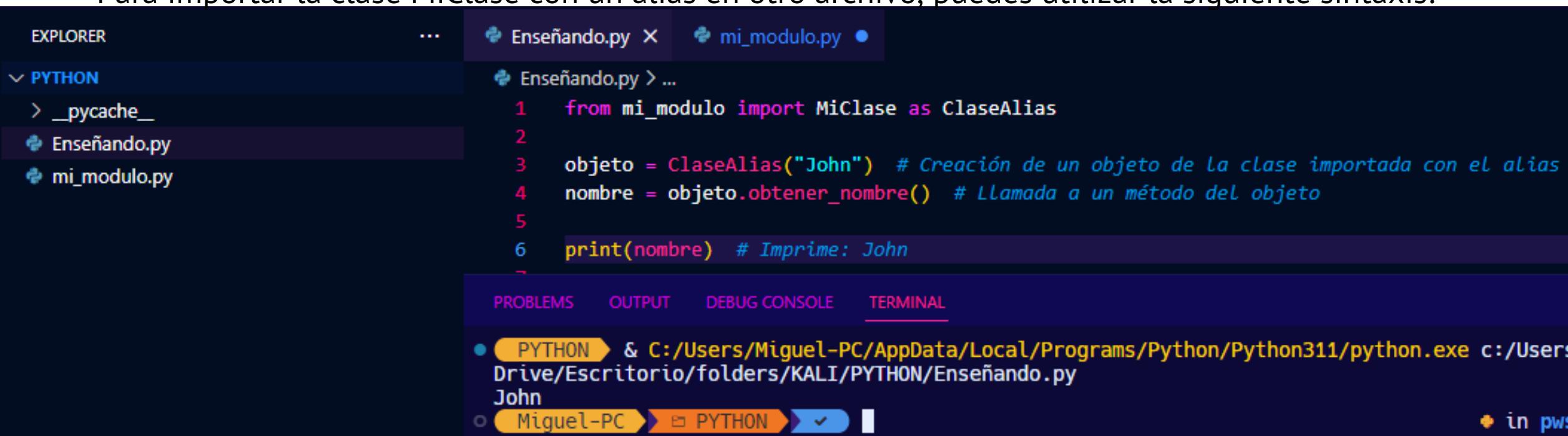
Importación con alias: Supongamos que tienes un archivo llamado mi_modulo.py que contiene la definición de una clase llamada MiClase:



The screenshot shows the VS Code interface. On the left, the Explorer sidebar shows a folder named 'PYTHON' containing '_pycache_'. Inside '_pycache_' are files 'Enseñando.py' and 'mi_modulo.py'. The 'mi_modulo.py' file is selected. On the right, the code editor displays the following Python code:

```
1 # mi_modulo.py
2 class MiClase:
3     def __init__(self, nombre):
4         self.nombre = nombre
5
6     def obtener_nombre(self):
7         return self.nombre
8
```

Para importar la clase MiClase con un alias en otro archivo, puedes utilizar la siguiente sintaxis:



The screenshot shows the VS Code interface. On the left, the Explorer sidebar shows a folder named 'PYTHON' containing '_pycache_'. Inside '_pycache_' are files 'Enseñando.py' and 'mi_modulo.py'. The 'Enseñando.py' file is selected. On the right, the code editor displays the following Python code:

```
1 from mi_modulo import MiClase as ClaseAlias
2
3 objeto = ClaseAlias("John") # Creación de un objeto de la clase importada con el alias
4 nombre = objeto.obtener_nombre() # Llamada a un método del objeto
5
6 print(nombre) # Imprime: John
```

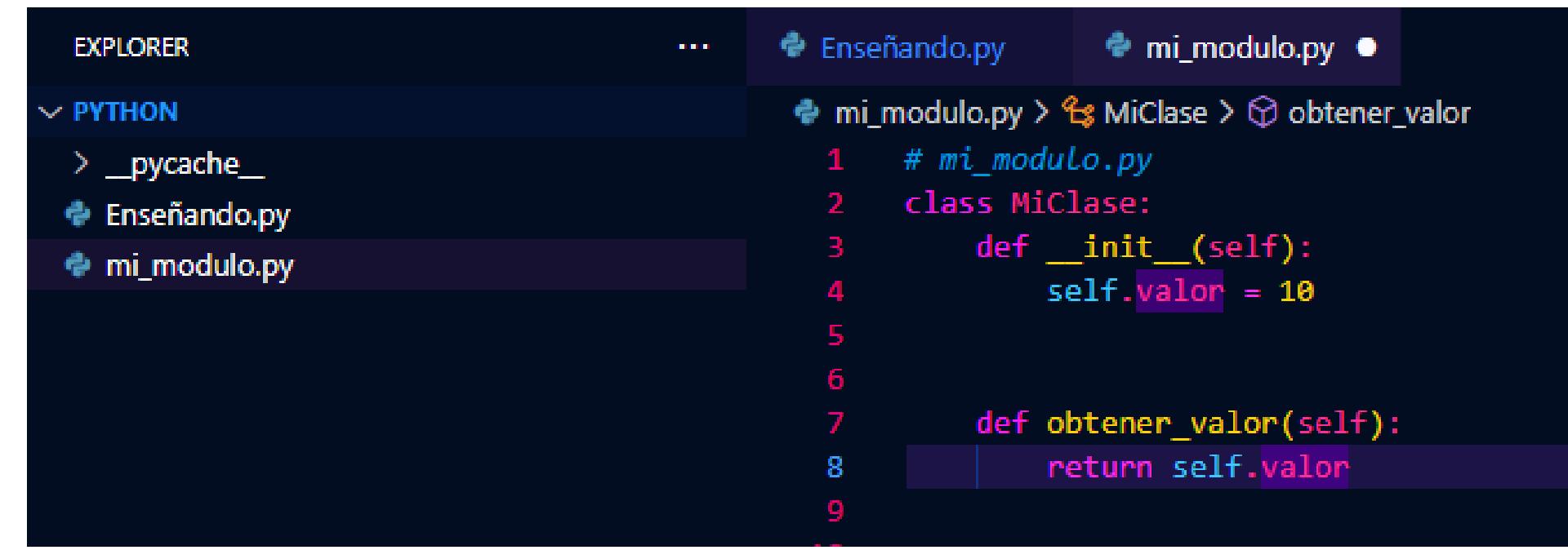
At the bottom, the terminal window shows the output of running the script:

```
PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python311/python.exe c:/Users/Drive/Escritorio/folders/KALI/PYTHON/Enseñando.py
John
Miguel-PC > PYTHON
```

CLASES

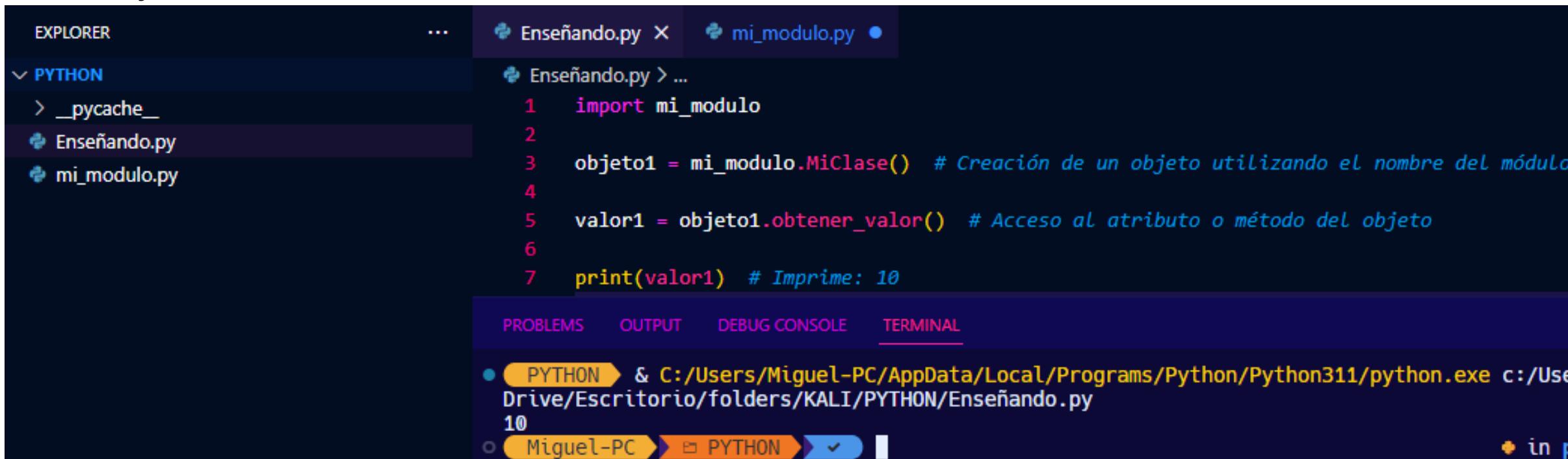
CREACION DE UN OBJETO(MODULO,SELECTIVO,ALIAS)

Supongamos que tienes un archivo llamado mi_modulo.py que contiene la definición de una clase llamada MiClase:



```
mi_modulo.py
1 # mi_modulo.py
2 class MiClase:
3     def __init__(self):
4         self.valor = 10
5
6
7     def obtener_valor(self):
8         return self.valor
9
```

Creación de un objeto utilizando el nombre del módulo:



```
mi_modulo.py
1 import mi_modulo
2
3 objeto1 = mi_modulo.MiClase() # Creación de un objeto utilizando el nombre del módulo
4
5 valor1 = objeto1.obtener_valor() # Acceso al atributo o método del objeto
6
7 print(valor1) # Imprime: 10
```

PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python311/python.exe c:/Users/Miguel-PC/Desktop/folders/KALI/PYTHON/Enseñando.py
10

En este caso, importamos el módulo completo mi_modulo y luego creamos un objeto de la clase MiClase utilizando la sintaxis mi_modulo.MiClase(). Luego, accedemos al atributo o método del objeto (obtener_valor()) y obtenemos el resultado.

CLASES

CREACION DE UN OBJETO(MODULO,SELECTIVO,ALIAS)

Creación de un objeto utilizando la clase importada selectivamente:



The screenshot shows a VS Code interface with the following details:

- EXPLORER**: Shows files: Enseñando.py (active), mi_modulo.py, and Enseñando.py > ...
- PYTHON**: Shows files: __pycache__, Enseñando.py (selected), and mi_modulo.py.
- Code Editor**: Displays the following Python code:

```
1  from mi_modulo import MiClase
2
3  objeto2 = MiClase() # Creación de un objeto utilizando la clase importada selectivamente
4
5  valor2 = objeto2.obtener_valor() # Acceso al atributo o método del objeto
6
7  print(valor2) # Imprime: 10
```
- TERMINAL**: Shows the command run in the terminal: & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python311/python.exe c:/Users/M... Drive/Escritorio/folders/KALI/PYTHON/Enseñando.py
- OUTPUT**: Shows the output: 10
- PROBLEMS**: Shows 0 problems.
- Status Bar**: Shows "in pwsh".

En este caso, importamos selectivamente solo la clase MiClase del módulo mi_modulo utilizando la sintaxis `from mi_modulo import MiClase`. Luego, creamos un objeto de la clase directamente con `MiClase()`. Después, accedemos al atributo o método del objeto y obtenemos el resultado.

CLASES

CREACION DE UN OBJETO(MODULO,SELECTIVO,ALIAS)

Creación de un objeto utilizando el alias:

The screenshot shows a Python code editor interface. In the Explorer sidebar, there are files named '_pycache_>', 'Enseñando.py', and 'mi_modulo.py'. The 'mi_modulo.py' file is currently selected. The main editor area contains the following code:

```
from mi_modulo import MiClase as ClaseAlias
objeto3 = ClaseAlias() # Creación de un objeto utilizando el alias
valor3 = objeto3.obtener_valor() # Acceso al atributo o método del objeto
print(valor3) # Imprime: 10
```

Below the editor, the terminal window shows the output of running the script:

```
PYTHON & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python311/python.
Drive/Escritorio/folders/KALI/PYTHON/Enseñando.py
10
```

En este caso, importamos la clase `MiClase` del módulo `mi_modulo` con el alias `ClaseAlias` utilizando la sintaxis `from mi_modulo import MiClase as ClaseAlias`. Luego, creamos un objeto de la clase utilizando el alias (`ClaseAlias()`). Después, accedemos al atributo o método del objeto y obtenemos el resultado.

En los tres casos, estamos creando objetos de la clase `MiClase` y accediendo a su atributo o método correspondiente.

CLASES

EJERCICIOS:

Problemática de los gatitos:

En una tienda de mascotas, el dueño desea llevar un registro de los gatitos que tiene disponibles para su adopción. Sin embargo, manejar toda la información manualmente se ha vuelto complicado y propenso a errores. El dueño necesita un sistema automatizado que le permita gestionar la información de los gatitos de manera eficiente.

Ejercicio que utiliza clases en Python:

Crea una clase llamada "Gatito" que represente a un gatito en la tienda de mascotas. La clase debe tener los siguientes atributos:

- **Nombre:** el nombre del gatito (cadena de texto).
- **Edad:** la edad del gatito en meses (entero).
- **Raza:** la raza del gatito (cadena de texto).
- **Color:** el color del pelaje del gatito (cadena de texto).

La clase también debe tener los siguientes métodos:

- **ObtenerInformacion():** este método debe imprimir por pantalla toda la información del gatito (nombre, edad, raza y color).
- **CalcularEdadEnAnios():** este método debe calcular y devolver la edad del gatito en años, asumiendo que un año equivale a 12 meses.

A continuación, crea una instancia de la clase "Gatito" y prueba los métodos para asegurarte de que funcionan correctamente. Por ejemplo:

CLASES

EJERCICIOS: SOLUCION

```
python Copy code

class Gatito:
    def __init__(self, nombre, edad, raza, color):
        self.nombre = nombre
        self.edad = edad
        self.raza = raza
        self.color = color

    def ObtenerInformacion(self):
        print("Nombre:", self.nombre)
        print("Edad:", self.edad, "meses")
        print("Raza:", self.raza)
        print("Color:", self.color)

    def CalcularEdadEnAnios(self):
        return self.edad // 12

# Crear una instancia de la clase Gatito
mi_gatito = Gatito("Mimi", 24, "Persa", "Blanco")

# Prueba de los métodos
mi_gatito.ObtenerInformacion()
print("Edad en años:", mi_gatito.CalcularEdadEnAnios())
```

CLASES

EJERCICIOS:SOLUCION

The screenshot shows a code editor window with a dark theme. The file is named 'Enseñando.py'. The code defines a class 'Gatito' with methods to initialize attributes and print information, and a method to calculate age in years. It then creates an instance of 'Gatito' and prints its details and calculated age.

```
❶ Enseñando.py X
❷ Enseñando.py > ...
❸
❹ class Gatito:
❺     def __init__(self, nombre, edad, raza, color):
❻         self.nombre = nombre
❼         self.edad = edad
❼         self.raza = raza
❼         self.color = color
❼
❼     def ObtenerInformacion(self):
❼         print("Nombre:", self.nombre)
❼         print("Edad:", self.edad, "meses")
❼         print("Raza:", self.raza)
❼         print("Color:", self.color)
❼
❼     def CalcularEdadEnAnios(self):
❼         return self.edad // 12
❼
❼ # Crear una instancia de la clase Gatito
❼ mi_gatito = Gatito("Mimi", 24, "Persa", "Blanco")
❼
❼ # Prueba de Los métodos
❼ mi_gatito.ObtenerInformacion()
❼ print("Edad en años:", mi_gatito.CalcularEdadEnAnios())
❽
```

Below the code editor, there is a terminal window showing the execution of the script and its output:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
● PYTHON > & C:/Users/Miguel-PC/AppData/Local/Programs/Python/Python38-32/Drive/Escritorio/folders/KALI/PYTHON/Enseñando.py
Nombre: Mimi
Edad: 24 meses
Raza: Persa
Color: Blanco
Edad en años: 2
```

EXEPCIONES

EXEPCIONES:

Las excepciones en Python son eventos que ocurren durante la ejecución de un programa y que interrumpen el flujo normal de ejecución. Estas excepciones pueden ser errores o situaciones inesperadas que el programa no puede manejar de manera predeterminada. Algunos ejemplos comunes de excepciones son el intento de dividir entre cero, acceder a un índice fuera de rango en una lista o intentar abrir un archivo que no existe.

Cómo se manejan las excepciones en el código: En Python, se utiliza un mecanismo llamado "manejo de excepciones" para tratar con estas situaciones inesperadas. El manejo de excepciones permite al programador anticiparse a los posibles errores y tomar acciones específicas cuando se producen.

El manejo de excepciones se realiza mediante bloques de código try-except. El código problemático se coloca dentro del bloque try, y si ocurre alguna excepción, se captura y se maneja en el bloque except correspondiente.

El formato básico del manejo de excepciones es el siguiente:

```
python
try:
    # Código problemático
    # ...
except TipoDeExcepcion:
    # Acciones a realizar cuando se captura la excepción
    # ...
```

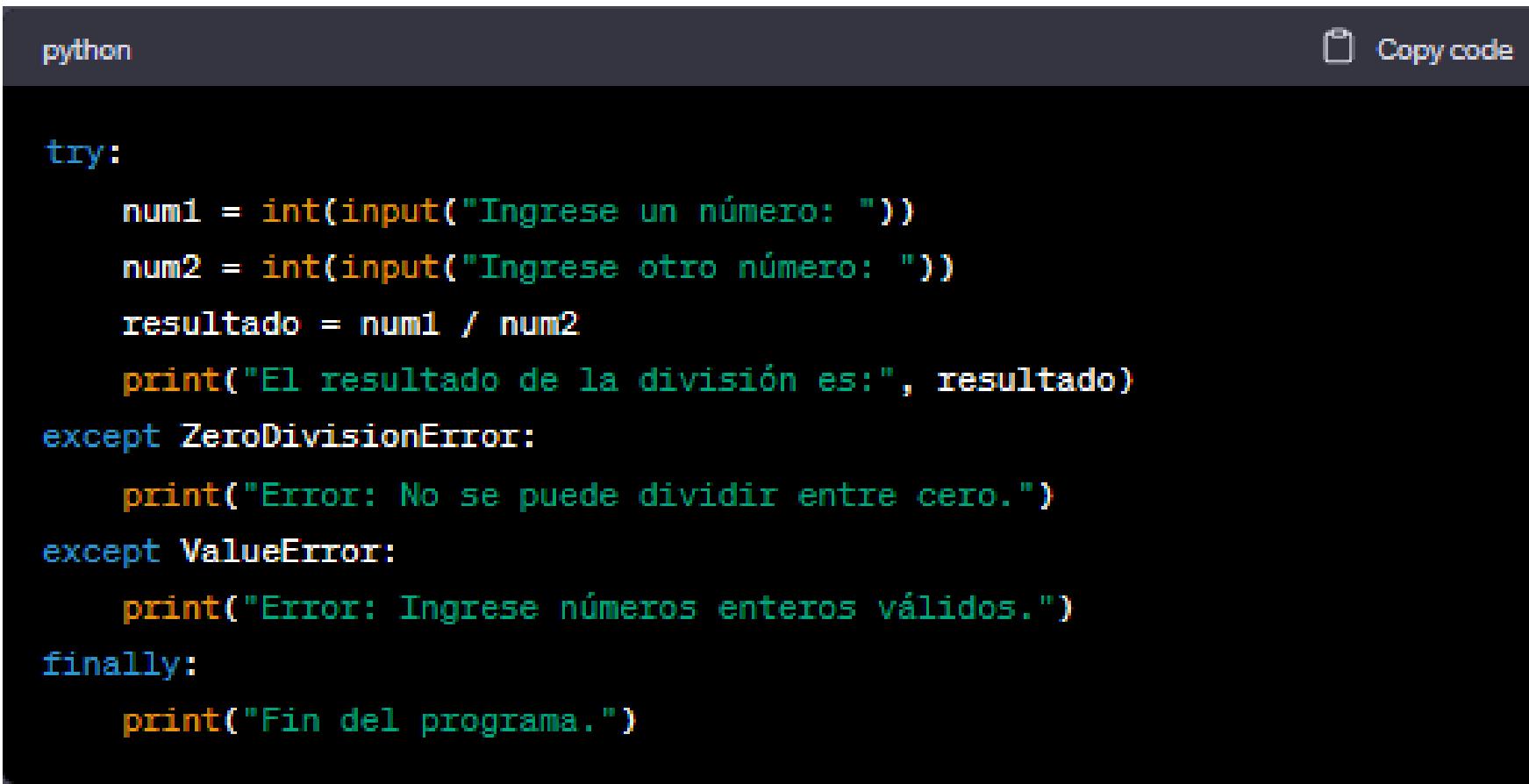
EXEPCIONES

EXEPCIONES:

El bloque try se ejecuta normalmente, pero si se produce una excepción del tipo especificado en el bloque except, el control se transfiere al bloque except y se ejecutan las acciones definidas allí. El bloque except puede tener múltiples cláusulas para manejar diferentes tipos de excepciones.

Además, se puede utilizar una cláusula finally opcional después del bloque except. El código dentro del bloque finally se ejecuta siempre, ya sea que se haya producido una excepción o no. Es útil para realizar acciones de limpieza o liberación de recursos.

Aquí hay un ejemplo para ilustrar el manejo de excepciones:



The screenshot shows a code editor window with a dark theme. The title bar says "python". The code itself is as follows:

```
python

try:
    num1 = int(input("Ingrese un número: "))
    num2 = int(input("Ingrese otro número: "))
    resultado = num1 / num2
    print("El resultado de la división es:", resultado)
except ZeroDivisionError:
    print("Error: No se puede dividir entre cero.")
except ValueError:
    print("Error: Ingrese números enteros válidos.")
finally:
    print("Fin del programa.")
```

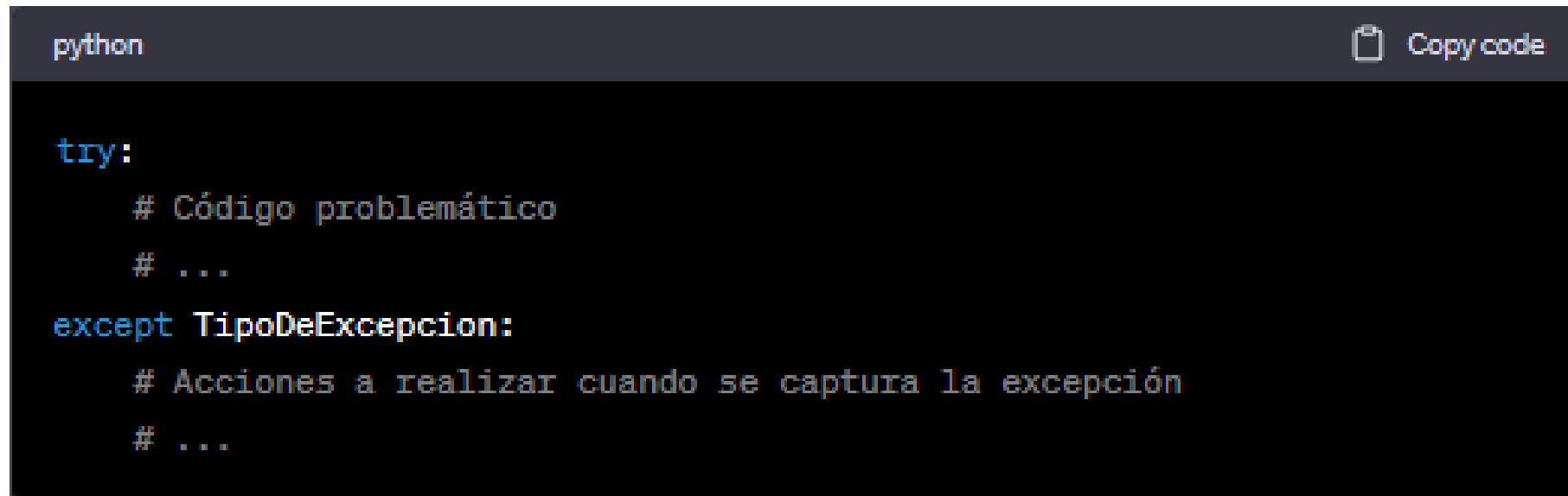
The code demonstrates a try-except block. It attempts to divide two integers input by the user. If a ZeroDivisionError occurs (division by zero), it prints an error message. If a ValueError occurs (non-integer input), it also prints an error message. Finally, it always prints "Fin del programa." regardless of any errors.

EXEPCIONES

BLOQUES TRY-EXCEPT:

El bloque try-except es una estructura en Python que permite manejar excepciones y controlar el flujo del programa en situaciones inesperadas. Se utiliza para proteger el código problemático y realizar acciones específicas cuando se producen excepciones.

Estructura básica del bloque try-except: La estructura básica del bloque try-except consiste en el bloque try, que contiene el código que podría generar una excepción, y uno o más bloques except que capturan y manejan las excepciones.



A screenshot of a code editor window titled "python". The code is written in Python and shows a basic try-except structure. The code is as follows:

```
try:  
    # Código problemático  
    # ...  
except TipoDeExcepcion:  
    # Acciones a realizar cuando se captura la excepción  
    # ...
```

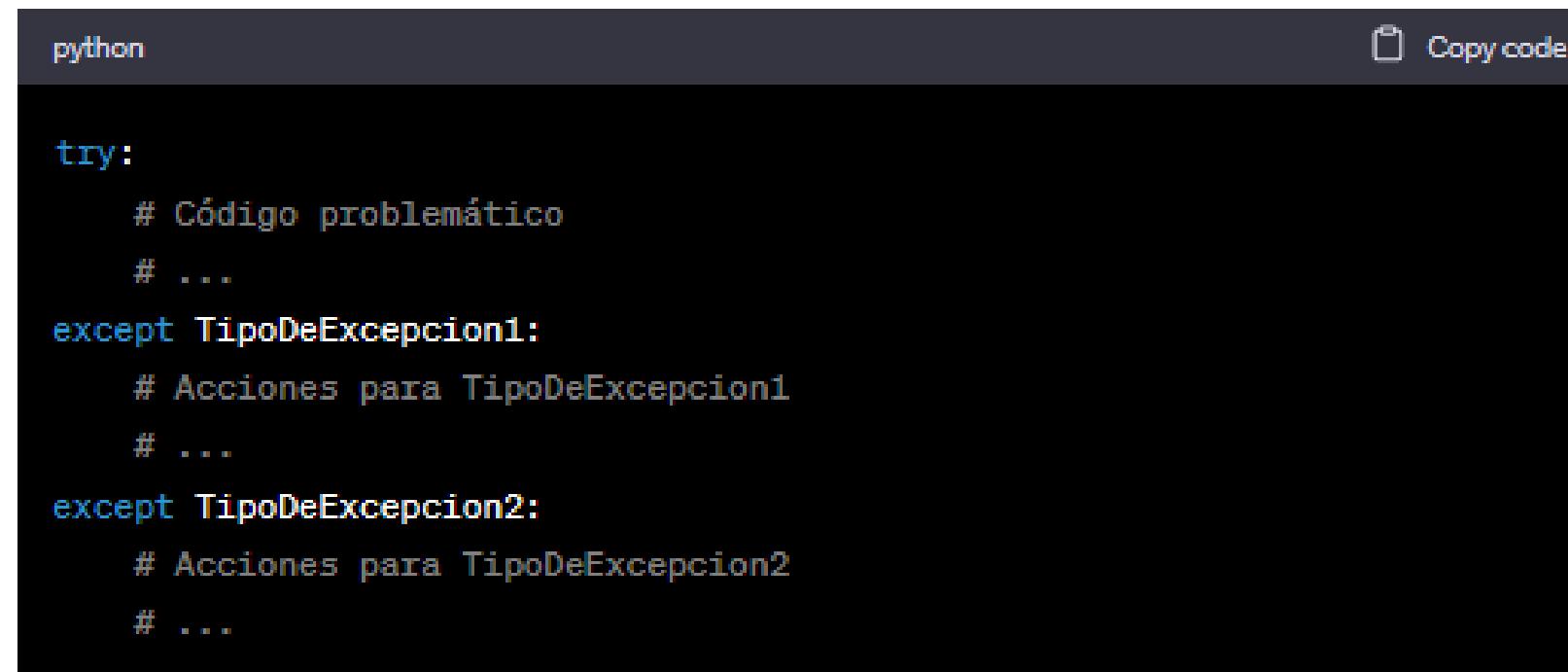
The code editor has a dark theme. The "Copy code" button is visible in the top right corner.

Cuando el código dentro del bloque try se ejecuta sin producir excepciones, el programa continúa normalmente y se omite el bloque except. Sin embargo, si se produce una excepción del tipo especificado, el control se transfiere al bloque except correspondiente, donde se ejecutan las acciones definidas para manejar la excepción.

EXEPCIONES

BLOQUES TRY-EXCEPT:

Captura de excepciones específicas: Es posible especificar el tipo de excepción que se desea capturar en el bloque except. Esto permite manejar diferentes tipos de excepciones de manera específica y tomar acciones personalizadas para cada caso.



A screenshot of a Python code editor window. The title bar says "python". The code area contains a try-except block:

```
python
try:
    # Código problemático
    # ...
except TipoDeExcepcion1:
    # Acciones para TipoDeExcepcion1
    # ...
except TipoDeExcepcion2:
    # Acciones para TipoDeExcepcion2
    # ...
```

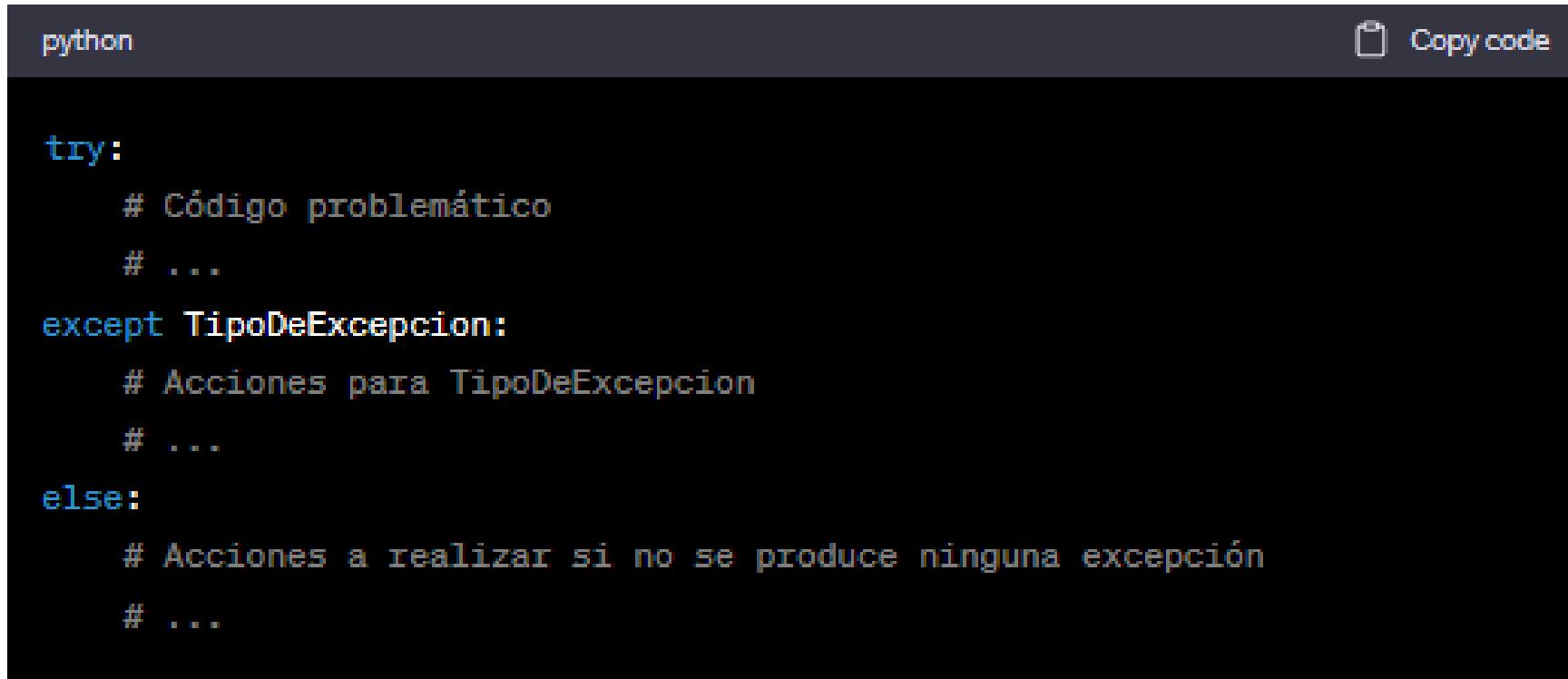
There is a "Copy code" button in the top right corner of the code editor.

En el ejemplo anterior, se capturarán y manejarán las excepciones del tipo TipoDeExcepcion1 y TipoDeExcepcion2. Cada bloque except puede contener acciones diferentes según el tipo de excepción capturada.

EXEPCIONES

BLOQUES TRY-EXCEPT:

Uso de la cláusula else en el bloque try-except: Además de los bloques try y except, es posible utilizar la cláusula else en un bloque try-except. La cláusula else se ejecuta si el bloque try no genera ninguna excepción. Es útil cuando se desea ejecutar cierto código solo cuando no se ha producido ninguna excepción.



The screenshot shows a code editor window with a dark theme. The title bar says "python". The code itself is:

```
python

try:
    # Código problemático
    # ...
except TipoDeExcepcion:
    # Acciones para TipoDeExcepcion
    # ...
else:
    # Acciones a realizar si no se produce ninguna excepción
    # ...
```

A "Copy code" button is visible in the top right corner of the editor window.

En el ejemplo anterior, si no se produce ninguna excepción en el bloque try, se ejecutarán las acciones definidas en el bloque else. Si se produce una excepción del tipo TipoDeExcepcion, el bloque except se ejecutará y el bloque else se omitirá.

EXEPCIONES

BLOQUES TRY-EXCEPT:

Uso de la cláusula finally en el bloque try-except: La cláusula finally es opcional y se utiliza para especificar un bloque de código que siempre se ejecutará, independientemente de si se ha producido una excepción o no. Es útil para realizar acciones de limpieza o liberación de recursos que deben realizarse sin importar las circunstancias.

```
Enseñando.py
1  try:
2      # Código problemático
3      # ...
4  except TipoDeExcepcion:
5      # Acciones para TipoDeExcepcion
6      # ...
7  finally:
8      # Acciones a realizar siempre, sin importar si se produce una excepción o no
9      # ...
10 
```

En el ejemplo anterior, el bloque finally se ejecutará siempre, ya sea que se haya producido una excepción o no. Puedes utilizarlo, por ejemplo, para cerrar archivos abiertos, liberar recursos o realizar cualquier otra tarea de limpieza necesaria.

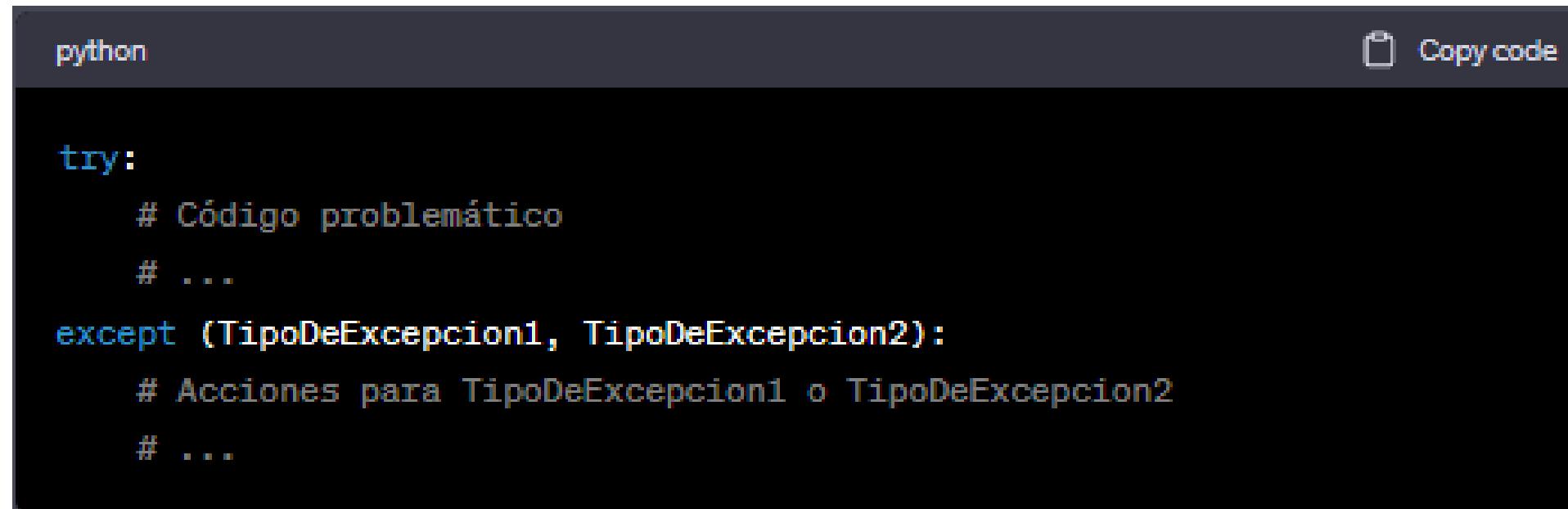
El uso de los bloques try-except, junto con las cláusulas else y finally, brinda un mayor control sobre el manejo de excepciones en Python, permitiéndote tomar acciones específicas para diferentes tipos de excepciones y garantizando que ciertas acciones se realicen siempre al finalizar la ejecución del bloque try.

EXCEPCIONES

CAPTURA DE MÚLTIPLES EXCEPCIONES:

En Python, es posible capturar y manejar diferentes tipos de excepciones en un solo bloque try-except. Esto nos permite gestionar distintos escenarios de error de manera específica en un solo lugar.

Capturar y manejar diferentes tipos de excepciones en un solo bloque try-except:



A screenshot of a code editor window titled "python". The code shown is:

```
try:  
    # Código problemático  
    # ...  
except (TipoDeExcepcion1, TipoDeExcepcion2):  
    # Acciones para TipoDeExcepcion1 o TipoDeExpcion2  
    # ...
```

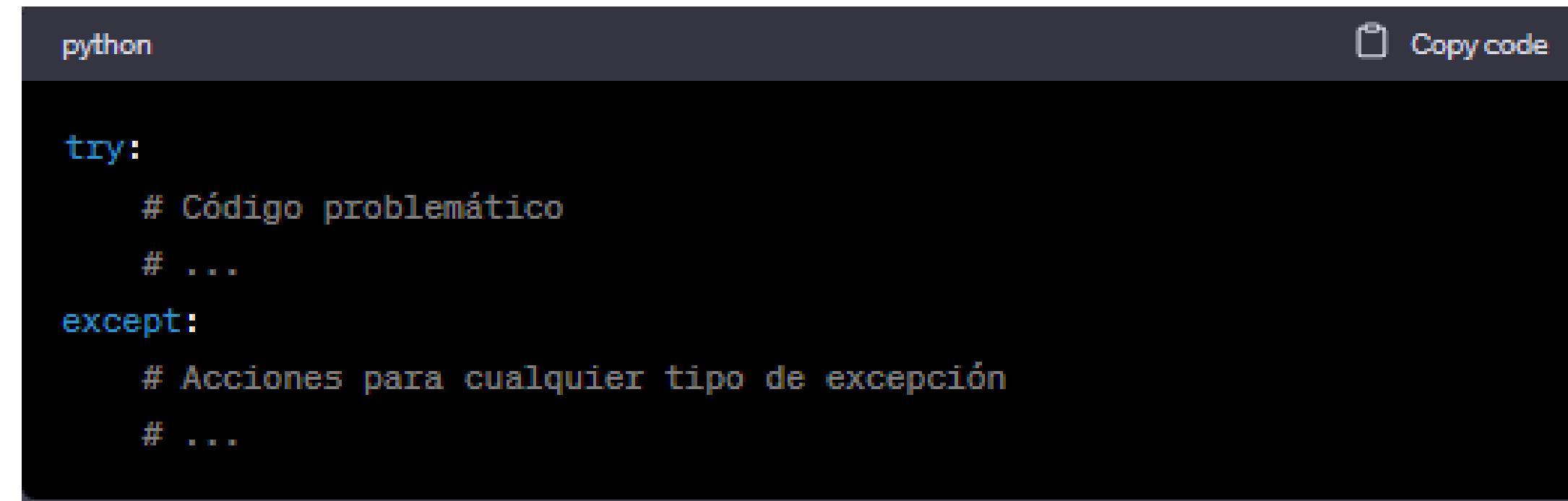
The code uses standard Python syntax for a try-except block. It includes a try block containing placeholder code, followed by an except block that catches two specific exception types: `TipoDeExcepcion1` and `TipoDeExcepcion2`. The except block also contains placeholder code. A "Copy code" button is visible in the top right corner of the editor window.

En este caso, se especifican los tipos de excepciones que se desean capturar y manejar dentro de paréntesis, separados por comas. Si se produce cualquiera de las excepciones especificadas, el bloque except correspondiente se ejecutará y se realizarán las acciones definidas para manejar esa excepción en particular.

EXEPCIONES

CAPTURA DE MÚLTIPLES EXCEPCIONES:

Uso de la cláusula except sin especificar el tipo de excepción: También es posible utilizar la cláusula except sin especificar el tipo de excepción. Esto capturará cualquier excepción que ocurra en el bloque try, permitiéndonos realizar acciones genéricas para cualquier tipo de excepción.



The screenshot shows a dark-themed code editor window with a Python script. The script contains a try block followed by an except block. The code is as follows:

```
python
try:
    # Código problemático
    # ...
except:
    # Acciones para cualquier tipo de excepción
    # ...
```

A "Copy code" button is visible in the top right corner of the editor window.

En este caso, el bloque except sin especificar el tipo de excepción capturará cualquier excepción que ocurra dentro del bloque try. Esto puede ser útil cuando deseamos realizar un manejo genérico de errores sin importar el tipo específico de excepción.

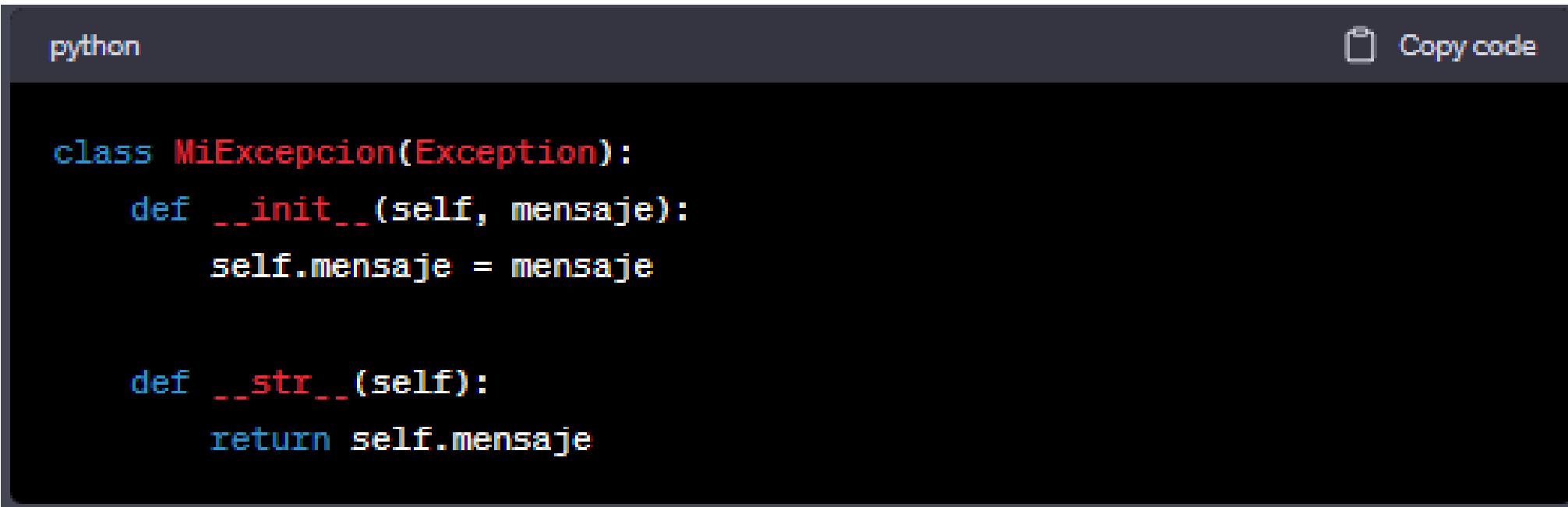
Es importante tener en cuenta que, al capturar múltiples excepciones o utilizar la cláusula except sin especificar el tipo de excepción, es posible que se pierda información sobre el tipo exacto de excepción que ocurrió. Por lo tanto, es recomendable capturar excepciones específicas cuando sea posible y utilizar la cláusula except sin tipo de excepción solo en casos donde un manejo genérico sea adecuado.

EXEPCIONES

EXCEPCIONES PERSONALIZADAS:

En Python, es posible crear y definir nuestras propias excepciones personalizadas para manejar situaciones específicas en nuestros programas. Las excepciones personalizadas nos permiten tener un control más preciso sobre los errores y nos brindan la capacidad de comunicar información específica sobre la excepción.

Creación y definición de excepciones personalizadas: Para crear una excepción personalizada, debemos definir una nueva clase que herede de la clase base "Exception" o de alguna de sus subclases. Podemos agregar atributos y métodos adicionales a nuestra excepción personalizada para proporcionar información específica sobre el error.



The screenshot shows a code editor window with a dark theme. The title bar says "python". The code area contains the following Python code:

```
python

class MiExcepcion(Exception):
    def __init__(self, mensaje):
        self.mensaje = mensaje

    def __str__(self):
        return self.mensaje
```

There is a "Copy code" button in the top right corner of the code editor.

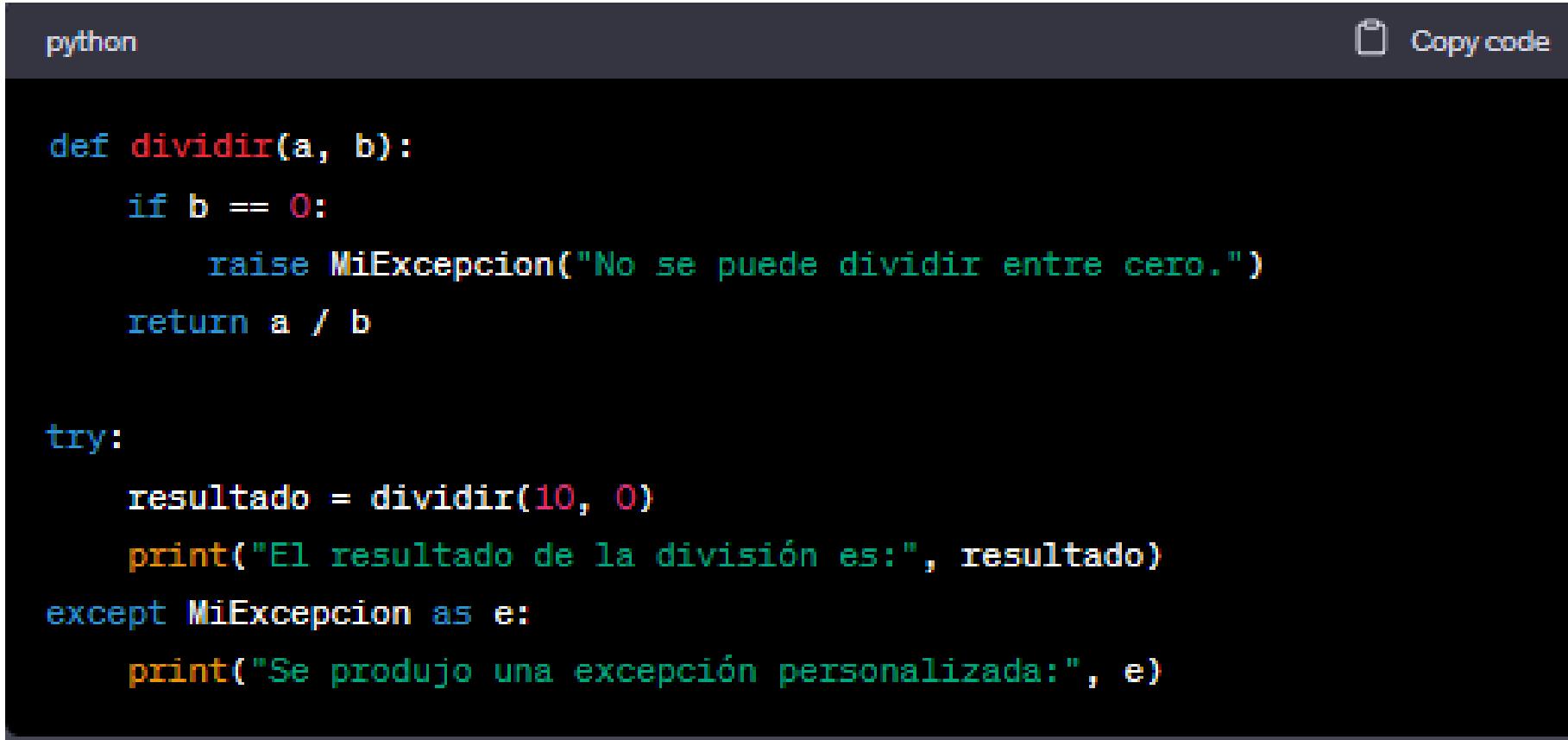
En este ejemplo, hemos creado la clase "MiExcepcion" que hereda de la clase "Exception". Hemos definido un constructor que recibe un mensaje como argumento y lo asigna al atributo "mensaje" de la excepción. También hemos sobrescrito el método "str" para que al imprimir la excepción, se muestre el mensaje definido.

EXEPCIONES

EXCEPCIONES PERSONALIZADAS:

Lanzamiento de excepciones personalizadas en el código:

Una vez que hemos definido nuestra excepción personalizada, podemos lanzarla en nuestro código cuando sea necesario. Para ello, utilizamos la palabra clave "raise" seguida de una instancia de nuestra excepción personalizada.



```
python

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

try:
    resultado = dividir(10, 0)
    print("El resultado de la división es:", resultado)
except MiExcepcion as e:
    print("Se produjo una excepción personalizada:", e)
```

En este ejemplo, hemos definido una función "dividir" que realiza una división, pero si el divisor es cero, lanzamos nuestra excepción personalizada "MiExcepcion". Luego, en el bloque try-except, capturamos la excepción y mostramos el mensaje de error.

El uso de excepciones personalizadas nos permite manejar situaciones específicas de error de una manera más clara y expresiva, y nos brinda la capacidad de comunicar información detallada sobre la excepción. Esto facilita la depuración y el mantenimiento de nuestro código.

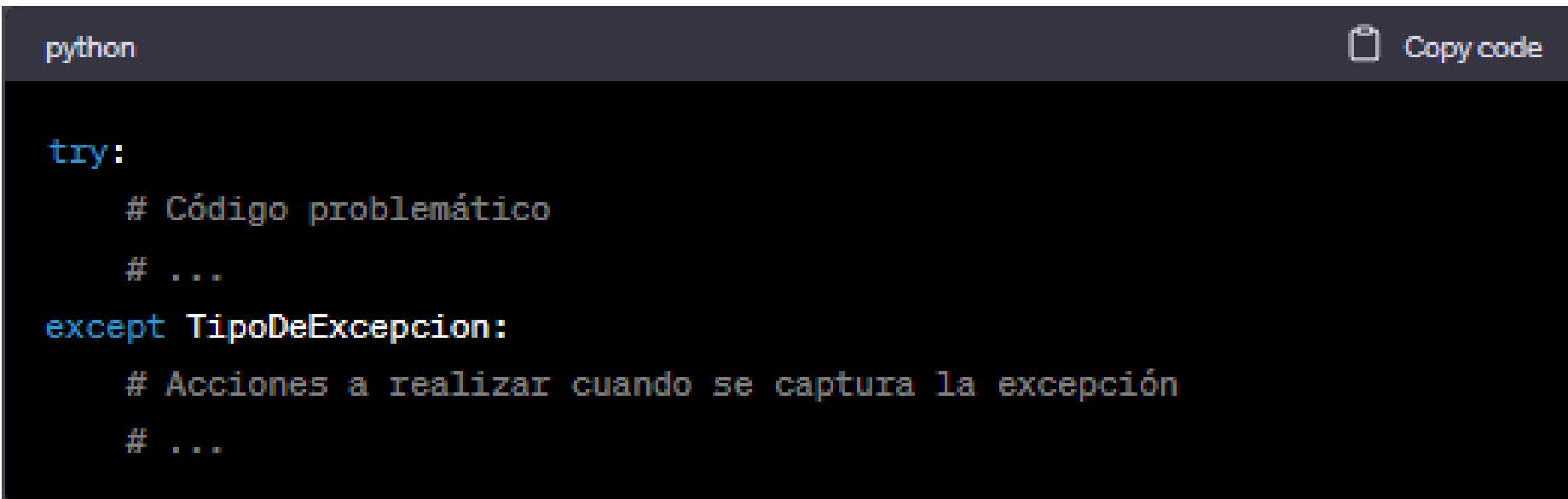
EXEPCIONES

GESTIÓN DE ERRORES Y CONTROL DE FLUJO:

La gestión de errores y el control de flujo son aspectos importantes en la programación para garantizar un comportamiento adecuado de nuestros programas y evitar que se interrumpan por errores inesperados. En Python, podemos utilizar excepciones para gestionar errores y controlar el flujo del programa de manera más controlada.

Uso de excepciones para gestionar errores y controlar el flujo del programa:

En lugar de que un programa se interrumpa abruptamente por un error, podemos utilizar bloques try-except para capturar excepciones y realizar acciones específicas cuando se produzcan errores.



A screenshot of a Python code editor window titled "python". The code shown is:

```
try:  
    # Código problemático  
    # ...  
except TipoDeExcepcion:  
    # Acciones a realizar cuando se captura la excepción  
    # ...
```

The "Copy code" button is visible in the top right corner of the editor window.

EXEPCIONES

GESTIÓN DE ERRORES Y CONTROL DE FLUJO:

Dentro del bloque try, colocamos el código que podría generar una excepción. Si se produce una excepción del tipo especificado en el bloque except, el control se transfiere al bloque except correspondiente, donde podemos manejar el error de manera apropiada. Esto nos permite tomar acciones específicas, como mostrar mensajes de error, intentar una solución alternativa o continuar con el flujo del programa.

Evitar que el programa se interrumpa por un error inesperado:

El uso de excepciones nos permite evitar que un programa se interrumpa completamente por un error inesperado. Al capturar excepciones, podemos manejar errores de manera controlada y tomar acciones para mitigar sus efectos.

Por ejemplo, si nuestro programa requiere la lectura de un archivo, podríamos manejar excepciones de archivo no encontrado o errores de lectura para mostrar un mensaje de error adecuado y continuar ejecutando el programa sin interrupción.

```
python

try:
    archivo = open("datos.txt", "r")
    # Código para trabajar con el archivo
    # ...
except FileNotFoundError:
    print("El archivo no se encontró.")
except IOError:
    print("Ocurrió un error al leer el archivo.")
```

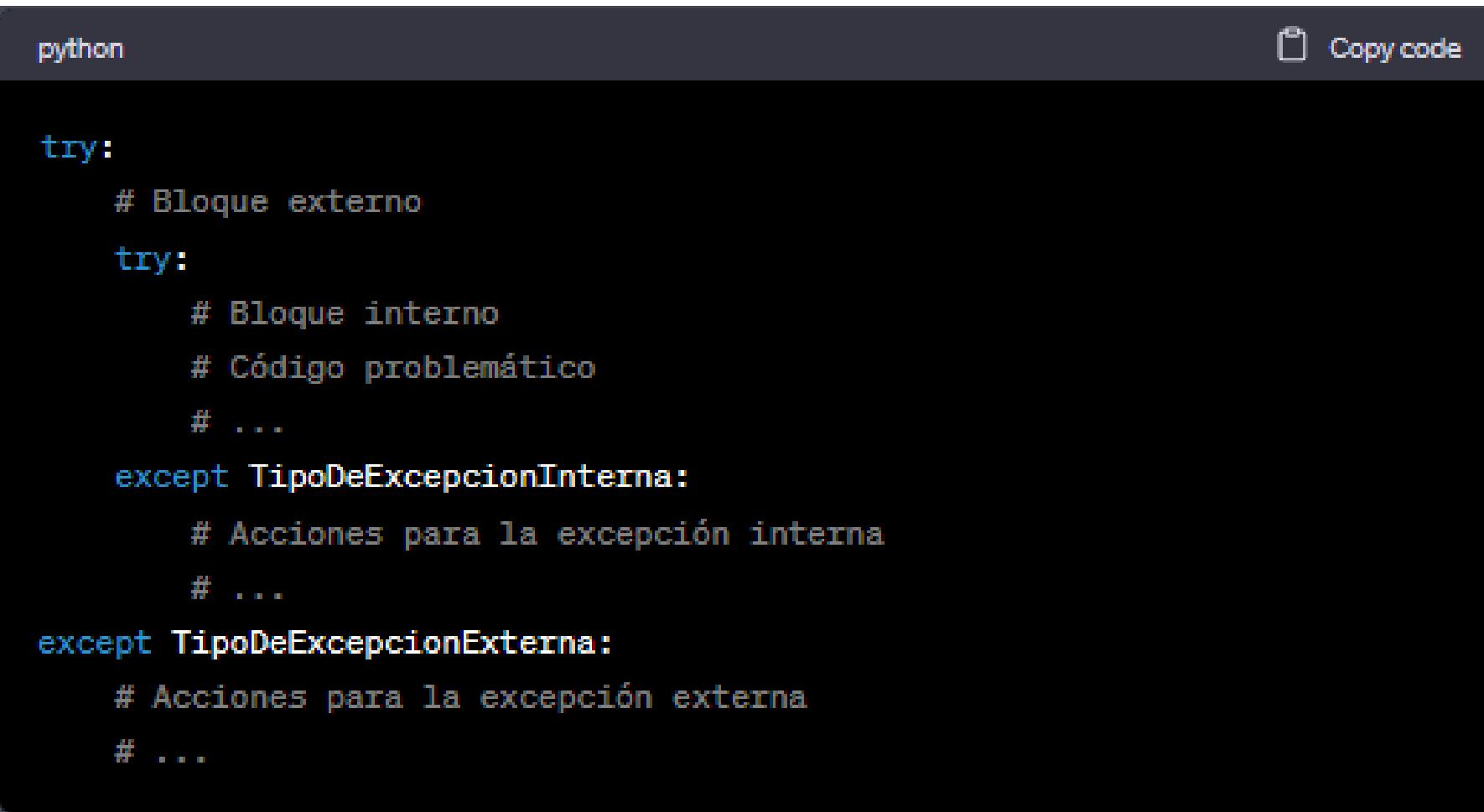
EXEPCIONES

MANEJO DE EXCEPCIONES ANIDADAS:

En Python, podemos manejar excepciones anidadas cuando ocurren dentro de otro bloque try-except. Esto nos permite capturar y manejar diferentes excepciones de manera más granular y específica.

Captura y manejo de excepciones que ocurren dentro de otro bloque try-except:

Podemos anidar bloques try-except para capturar excepciones que se producen dentro de un bloque try interno. Esto nos permite manejar errores de manera específica y realizar acciones adecuadas para cada situación.



The screenshot shows a code editor window with a dark theme. The title bar says "python". The code itself is as follows:

```
python

try:
    # Bloque externo
    try:
        # Bloque interno
        # Código problemático
        #
    except TipoDeExcepcionInterna:
        # Acciones para la excepción interna
        #
    except TipoDeExcepcionExterna:
        # Acciones para la excepción externa
        #

```

On the right side of the code editor, there is a "Copy code" button.

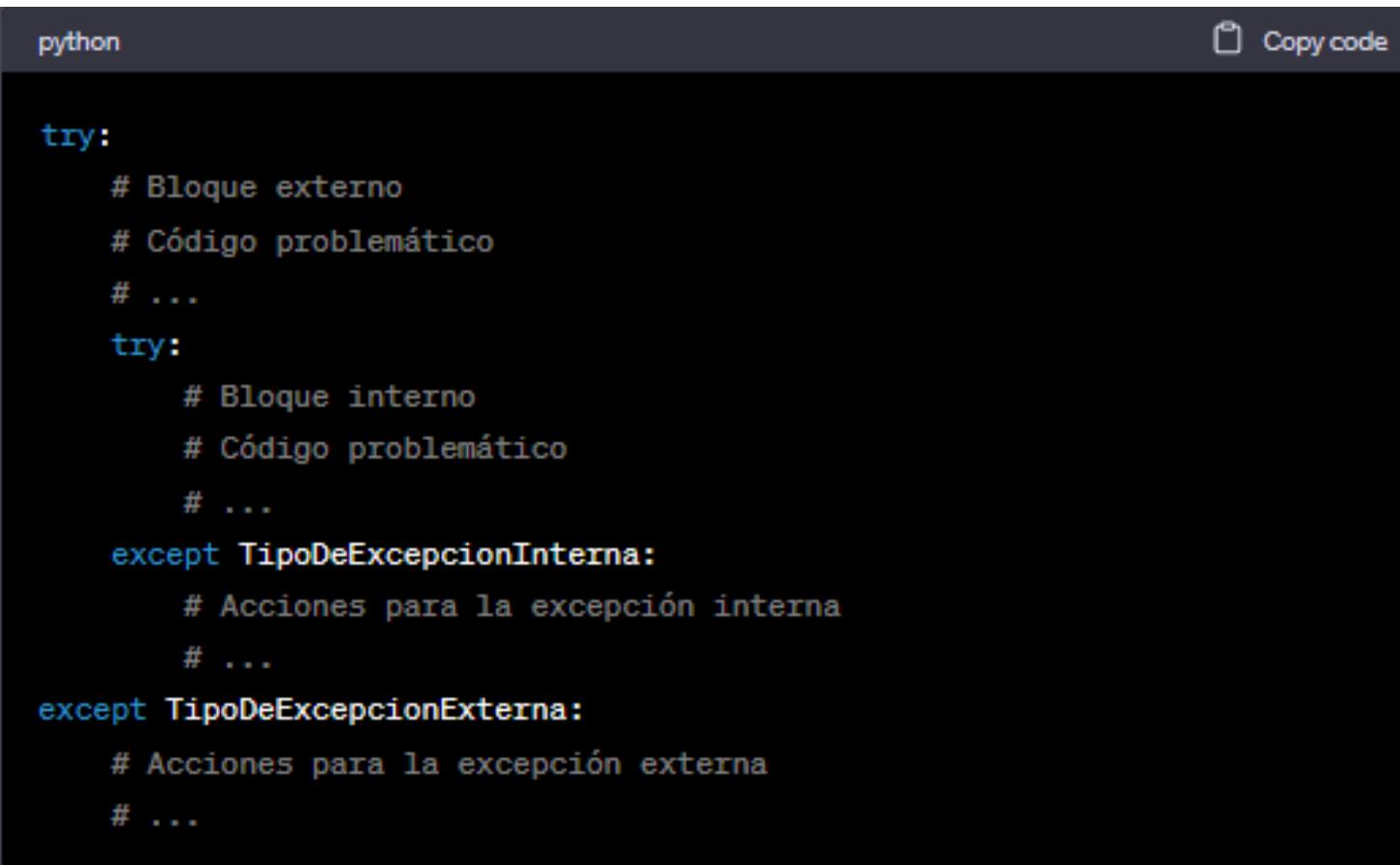
EXEPCIONES

MANEJO DE EXCEPCIONES ANIDADAS:

En este ejemplo, hemos anidado un bloque try-except dentro de otro. El bloque interno captura una excepción de tipo TipoDeExcepcionInterna y realiza acciones específicas para manejarla. Si se produce una excepción de tipo TipoDeExcepcionExterna en el bloque externo, se ejecutará el bloque except correspondiente y se realizarán las acciones definidas para esa excepción externa.

Anidamiento de bloques try-except para un manejo más granular de las excepciones:

El anidamiento de bloques try-except nos permite manejar diferentes excepciones de manera más granular, capturando y tratando errores específicos en diferentes niveles de anidamiento.



A screenshot of a Python code editor window titled "python". The code is as follows:

```
try:
    # Bloque externo
    # Código problemático
    # ...
    try:
        # Bloque interno
        # Código problemático
        # ...
    except TipoDeExcepcionInterna:
        # Acciones para la excepción interna
        # ...
except TipoDeExcepcionExterna:
    # Acciones para la excepción externa
    # ...
```

The code illustrates nested try-except blocks. The outer block handles an external exception (TipoDeExcepcionExterna) and the inner block handles an internal exception (TipoDeExcepcionInterna). Both blocks include placeholder code for handling the respective exceptions.

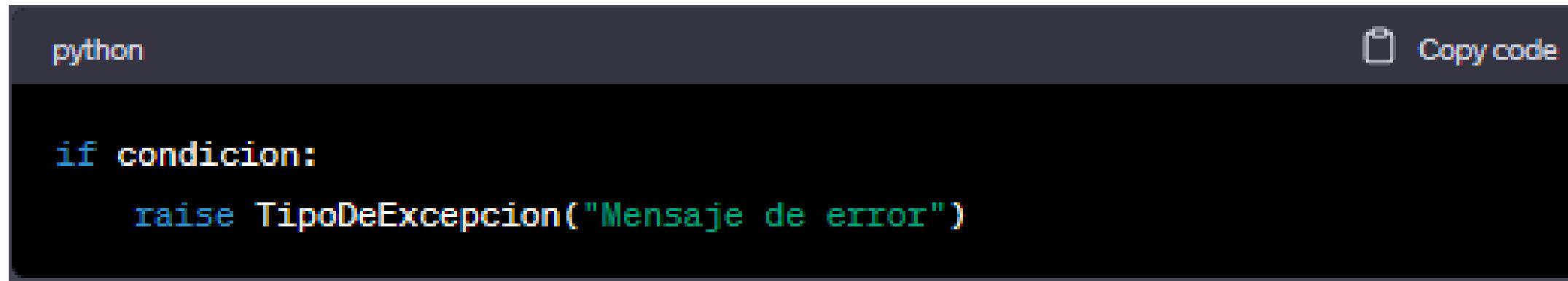
EXEPCIONES

USO DE LA CLÁUSULA RAISE:

necesito que me enseñes y me expliques de manera mas resumida y efectiva

Lanzamiento manual de excepciones en el código:

Podemos utilizar la cláusula raise seguida de una instancia de excepción para lanzar una excepción manualmente en cualquier parte de nuestro código.



A screenshot of a Python code editor. The title bar says "python". The code area contains the following Python code:

```
python

if condicion:
    raise TipoDeExcepcion("Mensaje de error")
```

On the right side of the code area, there is a "Copy code" button with a clipboard icon.

En este ejemplo, si la condición especificada se cumple, se lanzará una excepción de tipo TipoDeExcepcion con un mensaje de error específico. Esto nos permite generar errores controlados y personalizados en situaciones particulares.

Propagación de excepciones a través de bloques try-except superiores:

Cuando se lanza una excepción en un bloque try-except, la excepción puede propagarse a bloques try-except superiores hasta que se encuentre un bloque que pueda manejarla o hasta que el programa se interrumpa si no se captura la excepción.

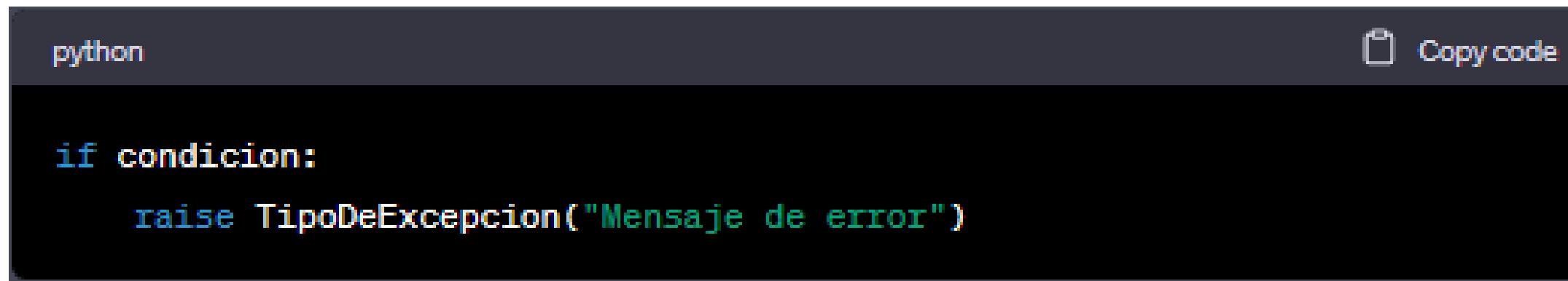
EXEPCIONES

USO DE LA CLÁUSULA RAISE:

necesito que me enseñes y me expliques de manera mas resumida y efectiva

Lanzamiento manual de excepciones en el código:

Podemos utilizar la cláusula raise seguida de una instancia de excepción para lanzar una excepción manualmente en cualquier parte de nuestro código.



A screenshot of a Python code editor. The title bar says "python". The code area contains the following Python code:

```
python

if condicion:
    raise TipoDeExcepcion("Mensaje de error")
```

On the right side of the code area, there is a "Copy code" button with a clipboard icon.

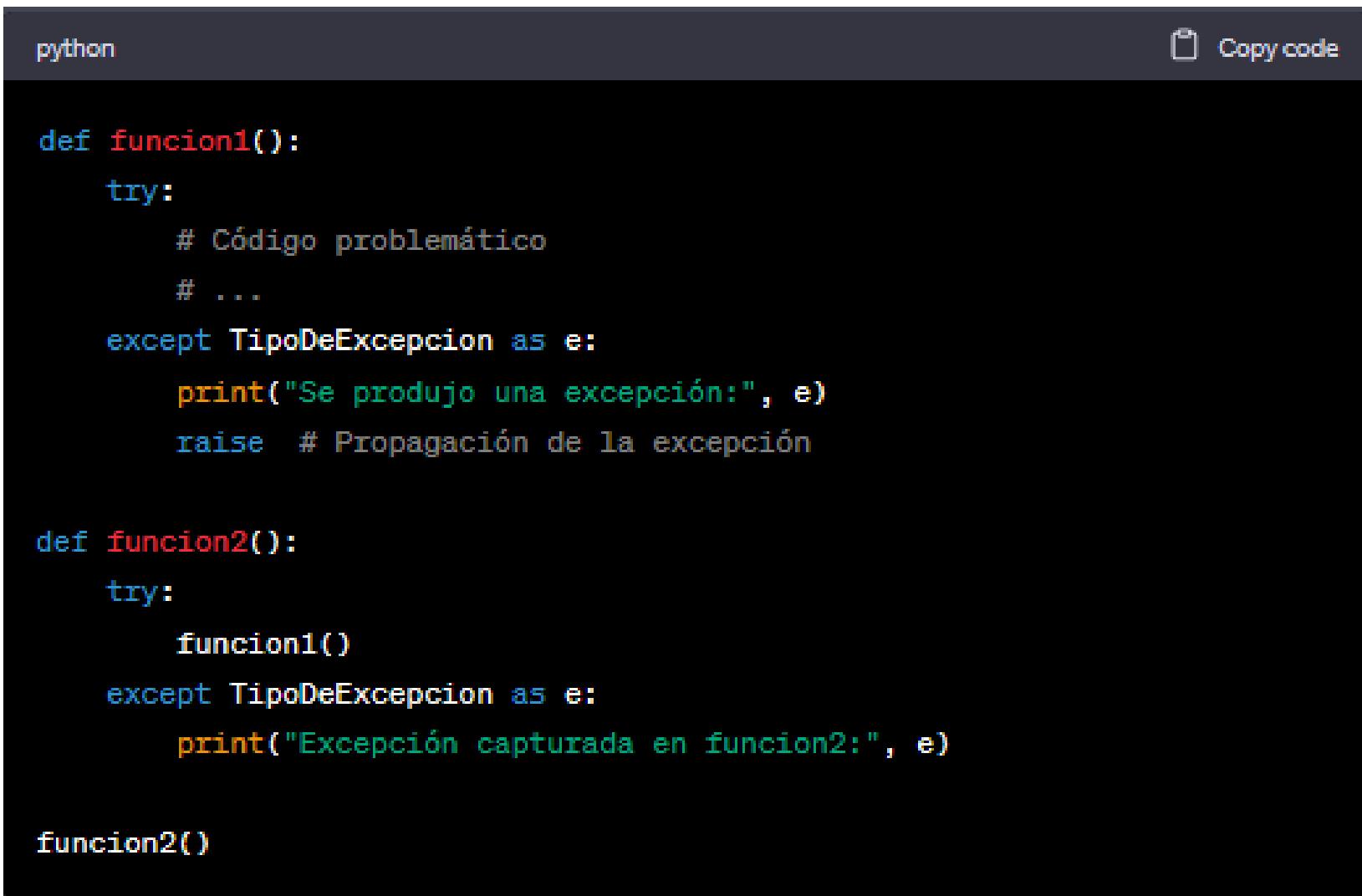
En este ejemplo, si la condición especificada se cumple, se lanzará una excepción de tipo TipoDeExcepcion con un mensaje de error específico. Esto nos permite generar errores controlados y personalizados en situaciones particulares.

Propagación de excepciones a través de bloques try-except superiores:

Cuando se lanza una excepción en un bloque try-except, la excepción puede propagarse a bloques try-except superiores hasta que se encuentre un bloque que pueda manejarla o hasta que el programa se interrumpa si no se captura la excepción.

EXEPCIONES

USO DE LA CLÁUSULA RAISE:



```
python
Copy code

def funcion1():
    try:
        # Código problemático
        # ...
    except TipoDeExcepcion as e:
        print("Se produjo una excepción:", e)
        raise # Propagación de la excepción

def funcion2():
    try:
        funcion1()
    except TipoDeExcepcion as e:
        print("Excepción capturada en funcion2:", e)

funcion2()
```

En este ejemplo, la función `funcion1()` contiene un bloque `try-except` que captura una excepción de tipo `TipoDeExcepcion`. Dentro del bloque `except`, se imprime un mensaje y luego se utiliza la cláusula `raise` sin argumentos para propagar la excepción. La función `funcion2()` también contiene un bloque `try-except` que captura la excepción `TipoDeExcepcion` y la maneja.

Cuando se ejecuta el código, si se produce una excepción en `funcion1()`, esta se captura y se imprime un mensaje. Luego, la excepción se propaga hacia arriba y es capturada por el bloque `try-except` en `funcion2()`, donde se maneja nuevamente.

El uso de la cláusula `raise` nos permite lanzar manualmente excepciones en el código y controlar situaciones de error específicas. Además, la propagación de excepciones nos permite gestionar errores en diferentes niveles de bloques `try-except`, permitiendo que una excepción se propague hacia arriba en la cadena de llamadas hasta que se encuentre un bloque que pueda manejarla.

EXEPCIONES

USO DE LA CLÁUSULA ASSERT:

En Python, la cláusula assert se utiliza para verificar condiciones en el código y lanzar una excepción si la condición es falsa. Nos permite realizar verificaciones durante el desarrollo y asegurar que se cumplan ciertas condiciones en nuestro programa.

Uso de la afirmación assert para verificar condiciones en el código:

La sintaxis básica de una afirmación assert es la siguiente:

```
python
assert condicion, "Mensaje de error"
```

Si la condición especificada es verdadera, el programa continúa su ejecución sin problemas. Sin embargo, si la condición es falsa, se lanza una excepción AssertionError con el mensaje de error proporcionado.

```
python
edad = 17
assert edad >= 18, "Debes ser mayor de edad para ingresar"
```

EXEPCIONES

USO DE LA CLÁUSULA ASSERT:

En este ejemplo, estamos verificando si la variable "edad" es mayor o igual a 18. Si la condición es falsa, se lanzará una excepción `AssertionError` con el mensaje de error "Debes ser mayor de edad para ingresar".

Lanzamiento de excepciones basadas en afirmaciones falsas:

La cláusula `assert` nos permite lanzar excepciones basadas en afirmaciones falsas, lo que nos ayuda a detectar y solucionar errores en nuestro código.

The screenshot shows a dark-themed Python code editor. At the top, there's a toolbar with a 'python' icon and a 'Copy code' button. The code itself is as follows:

```
def dividir(a, b):
    assert b != 0, "El divisor no puede ser cero"
    return a / b

resultado = dividir(10, 0)
```

En este ejemplo, estamos definiendo una función "dividir" que realiza una división. Antes de realizar la división, utilizamos una afirmación `assert` para verificar si el divisor es distinto de cero. Si el divisor es cero, se lanzará una excepción `AssertionError` con el mensaje de error "El divisor no puede ser cero".

El uso de la cláusula `assert` nos permite realizar verificaciones en nuestro código y detectar condiciones incorrectas o inesperadas. Nos ayuda a asegurarnos de que se cumplan ciertas condiciones y a detectar errores durante el desarrollo. Si una afirmación es falsa, se lanzará una excepción que nos indica que algo no está como debería estar.

EXEPCIONES

EJERCICIOS:

Problema de los gatitos:

Supongamos que estás desarrollando un programa para un refugio de gatos. El refugio tiene diferentes gatitos y necesita un sistema para controlar su información. Cada gatito tiene un nombre, una edad en meses y una condición de salud (representada por una cadena de texto).

El problema es el siguiente: debes implementar una función que tome como entrada una lista de gatitos y los clasifique en dos grupos: los gatitos saludables y los gatitos que necesitan atención médica. Un gatito se considera saludable si su condición de salud es "Saludable" y su edad es menor a 12 meses. Los gatitos que no cumplen estas condiciones se consideran que necesitan atención médica.

El objetivo es que tu función devuelva dos listas: una lista con los gatitos saludables y otra lista con los gatitos que necesitan atención médica.

Ejercicio de excepciones en Python:

Escribe un programa en Python que solicite al usuario ingresar dos números enteros. Luego, realiza la división del primer número entre el segundo número y muestra el resultado.

Sin embargo, debes manejar posibles errores que puedan ocurrir durante la ejecución del programa. En particular, debes manejar las siguientes situaciones:

1. Si el usuario ingresa un valor que no sea un número entero, el programa debe mostrar un mensaje de error indicando que el valor ingresado no es válido.
2. Si el usuario ingresa 0 como segundo número, el programa debe mostrar un mensaje de error indicando que la división entre cero no está permitida.

El programa debe garantizar que no se produzcan errores inesperados y que muestre mensajes de error claros en caso de que se ingresen valores no válidos o se intente realizar una división entre cero.

EXEPCIONES

EJERCICIOS: SOLUCION

```
python Copy code

def clasificar_gatitos(gatitos):
    gatitos_saludables = []
    gatitos_atencion_medica = []

    for gatito in gatitos:
        if gatito["condicion_salud"] == "Saludable" and gatito["edad"] < 12:
            gatitos_saludables.append(gatito)
        else:
            gatitos_atencion_medica.append(gatito)

    return gatitos_saludables, gatitos_atencion_medica

# Ejemplo de uso
lista_gatitos = [
    {"nombre": "Pelusa", "edad": 6, "condicion_salud": "Saludable"},
    {"nombre": "Garfield", "edad": 18, "condicion_salud": "Enfermo"},
    {"nombre": "Mittens", "edad": 8, "condicion_salud": "Saludable"},
    {"nombre": "Whiskers", "edad": 3, "condicion_salud": "Enfermo"}
]

saludables, atencion_medica = clasificar_gatitos(lista_gatitos)

print("Gatitos saludables:")
for gatito in saludables:
    print(gatito["nombre"])

print("Gatitos que necesitan atención médica:")
for gatito in atencion_medica:
    print(gatito["nombre"])
```

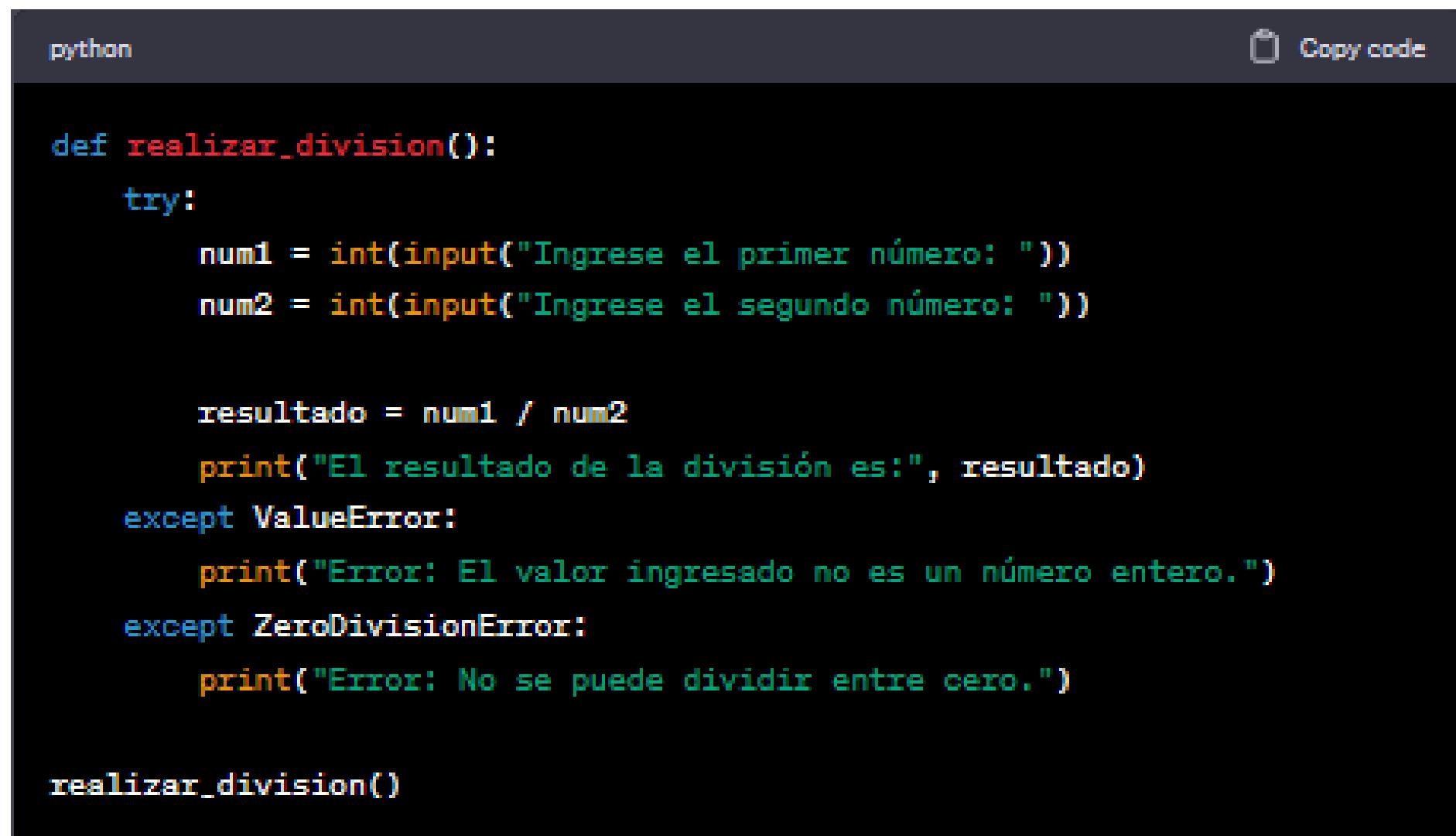
EXEPCIONES

EJERCICIOS: SOLUCION

En este ejemplo, definimos la función clasificar_gatitos que recibe como argumento una lista de diccionarios que representan a los gatitos. La función itera sobre cada gatito y los clasifica según las condiciones establecidas. Luego, retorna dos listas: gatitos_saludables y gatitos_atencion_medica.

Luego, probamos la función con una lista de gatitos de ejemplo y mostramos los resultados separando los gatitos saludables de los que necesitan atención médica.

Ejercicio de excepciones en Python (solución en código):



```
python
Copy code

def realizar_division():
    try:
        num1 = int(input("Ingrese el primer número: "))
        num2 = int(input("Ingrese el segundo número: "))

        resultado = num1 / num2
        print("El resultado de la división es:", resultado)
    except ValueError:
        print("Error: El valor ingresado no es un número entero.")
    except ZeroDivisionError:
        print("Error: No se puede dividir entre cero.")

realizar_division()
```

EXEPCIONES

EJERCICIOS: SOLUCION

En este ejemplo, definimos la función realizar_division que solicita al usuario ingresar dos números enteros utilizando la función input y los guarda en las variables num1 y num2. Luego, intenta realizar la división y muestra el resultado.

Sin embargo, utilizamos bloques try-except para capturar posibles excepciones que pueden ocurrir durante la ejecución. Si se produce un ValueError, se muestra un mensaje de error indicando que el valor ingresado no es un número entero. Si se produce un ZeroDivisionError, se muestra un mensaje de error indicando que no se puede dividir entre cero.

De esta manera, el programa maneja adecuadamente los posibles errores y muestra mensajes de error claros en caso de ingresar valores no válidos o intentar realizar una división entre cero.