Programación en Python

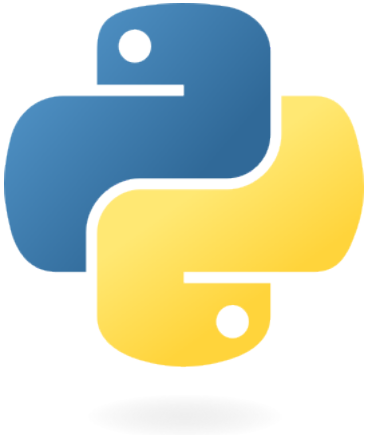
Clase 1: Conociendo Python

18/03/2026

Qué es Python

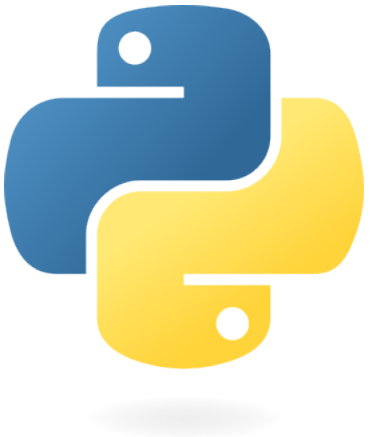
Python en una *slide*

- Lenguaje interpretado



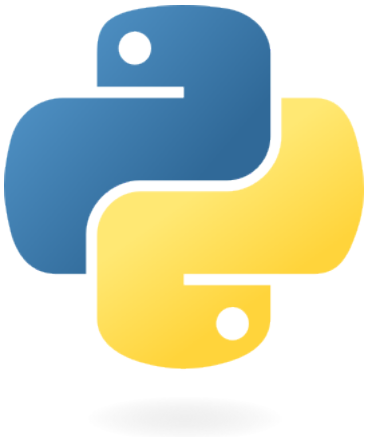
Python en una *slide*

- Lenguaje interpretado
- De propósito general



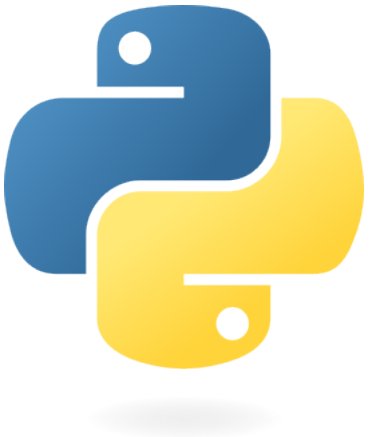
Python en una *slide*

- Lenguaje interpretado
- De propósito general
- De alto nivel



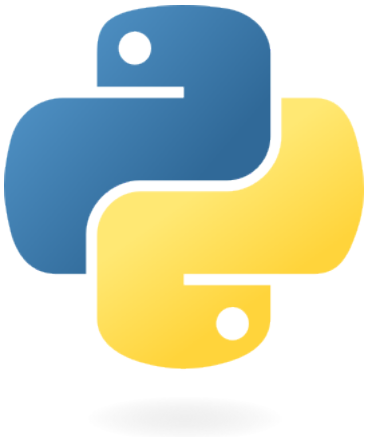
Python en una *slide*

- Lenguaje interpretado
- De propósito general
- De alto nivel
- Gratis



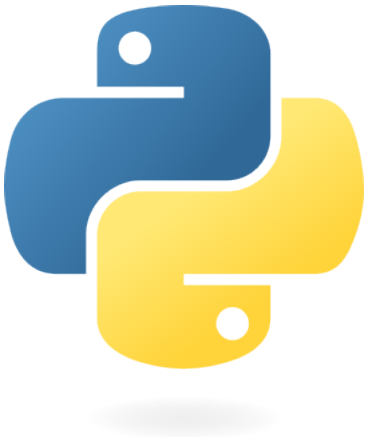
Python en una *slide*

- Lenguaje interpretado
- De propósito general
- De alto nivel
- Gratis
- Código abierto



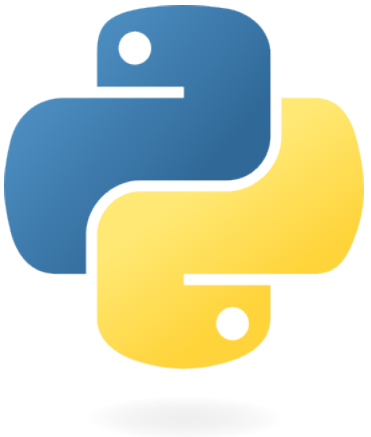
Python en una *slide*

- Lenguaje interpretado
- De propósito general
- De alto nivel
- Gratis
- Código abierto
- Ecosistema de librerías



Python en una *slide*

- Lenguaje interpretado
- De propósito general
- De alto nivel
- Gratis
- Código abierto
- Ecosistema de librerías
- Comunidad



Positron

Positron en una *slide*

- Editor de código

Positron en una *slide*

- Editor de código
- Creado por Posit (antes RStudio)

Positron en una *slide*

- Editor de código
- Creado por Posit (antes RStudio)
- Basado en Visual Studio Code

Positron en una *slide*

- Editor de código
- Creado por Posit (antes RStudio)
- Basado en Visual Studio Code
- Diseñado para ciencia de datos

Positron en una *slide*

- Editor de código
- Creado por Posit (antes RStudio)
- Basado en Visual Studio Code
- Diseñado para ciencia de datos
- Enfocado en R y Python

Positron en una *slide*

- Editor de código
- Creado por Posit (antes RStudio)
- Basado en Visual Studio Code
- Diseñado para ciencia de datos
- Enfocado en R y Python
- Gratis

Positron en una *slide*

- Editor de código
- Creado por Posit (antes RStudio)
- Basado en Visual Studio Code
- Diseñado para ciencia de datos
- Enfocado en R y Python
- Gratis
- Código abierto

Positron en una *slide*

- Editor de código
- Creado por Posit (antes RStudio)
- Basado en Visual Studio Code
- Diseñado para ciencia de datos
- Enfocado en R y Python
- Gratis
- Código abierto
- Rstudio ❤️ VS Code

A laburar

Tipos de datos elementales

Tipos de datos elementales

Python cuenta con 5 tipos de datos elementales:
`int`, `float`, `str`, `bool`, `NoneType`.

```
13              # <class 'int'>
2.71            # <class 'float'>
13.0            # <class 'float'>
13.             # <class 'float'>
.7              # <class 'float'>
"Hola hola"     # <class 'str'>
'Hola hola'     # <class 'str'>
True            # <class 'bool'>
False           # <class 'bool'>
None            # <class 'NoneType'>
```

i Escalares de verdad

Python no es un lenguaje vectorizado, a diferencia de R.
Los valores enteros, flotantes, etc. son **escalares** de verdad,
no vectores de longitud 1.

i Escalares de verdad

Python no es un lenguaje vectorizado, a diferencia de R.
Los valores enteros, flotantes, etc. son **escalares** de verdad,
no vectores de longitud 1.

✅ Conversión entre tipos

Es posible convertir entre tipos de datos, siempre que tenga sentido.

```
float(12)
bool(0)
float("1.25")
int(12.5)
str(False)
```

Variables y asignaciones

Variables y asignaciones

Las variables se asignan con el símbolo = y la sintaxis es:

```
<nombre> = <expresión>
```

Variables y asignaciones

Las variables se asignan con el símbolo = y la sintaxis es:

```
<nombre> = <expresión>
```

Por ejemplo:

```
mensaje = "¡Hola, curso!"
```

Variables y asignaciones

Las variables se asignan con el símbolo = y la sintaxis es:

```
<nombre> = <expresión>
```

Por ejemplo:

```
mensaje = "¡Hola, curso!"
```

o:

```
a = 1
b = -5
c = 6
x1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
x2 = (-b - (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
```

Asignacion múltiple

En vez de hacer esto:

```
a = 1
b = -5
c = 6
```

Asignacion múltiple

En vez de hacer esto:

```
a = 1
b = -5
c = 6
```

Python nos permite hacer esto:

```
a, b, c = 1, -5, 6
```

Asignacion múltiple

En vez de hacer esto:

```
a = 1
b = -5
c = 6
```

Python nos permite hacer esto:

```
a, b, c = 1, -5, 6
```

! La realidad siempre es más compleja

La asignación múltiple no es un simple atajo sintáctico, es consecuencia del **empaquetado** y **desempaquetado**, mecanismo que veremos más adelante.

Nombres permitidos

- Solo pueden contener **letras**, **números** y **guiones bajos**

Nombres permitidos

- Solo pueden contener **letras**, **números** y **guiones bajos**
- Deben comenzar con letra o guión bajo, pero no con un número.
  - `mensaje_1` ✅
  - `1_mensaje` ❌

Nombres permitidos

- Solo pueden contener **letras**, **números** y **guiones bajos**
- Deben comenzar con letra o guión bajo, pero no con un número.
  - `mensaje_1` ✅
  - `1_mensaje` ❌
- No pueden contener espacios
  - `el_mensaje` ✅
  - `el mensaje` ❌

Nombres permitidos

- Solo pueden contener **letras**, **números** y **guiones bajos**
- Deben comenzar con letra o guión bajo, pero no con un número.
  - `mensaje_1` ✅
  - `1_mensaje` ❌
- No pueden contener espacios
  - `el_mensaje` ✅
  - `el mensaje` ❌
- No pueden ser palabras reservadas de Python

Nombres permitidos

Consejos

- Usar nombres breves pero descriptivos
  - `mensaje` mejor que `mj`
  - `equipo_favorito` mejor que `eq_fav`
- No usar tildes ni caracteres específicos del castellano (`ñ`).

Expresiones

Expresiones

¿Qué es una expresión?

Una expresión es una combinación de **operadores** y **operandos** que se puede evaluar para producir un valor.

Expresiones

¿Qué es un operador?

Un símbolo o palabra que define una operación.

- Aritméticos: `+`, `-`, `*`, `/`, `**`, `//`, `%`.
- Lógicos: `not`, `and`, `or`.
- Comparación: `is`, `is not`, `==`, `!=`, `>`, `>=`, `<`, `<=`.

Expresiones

¿Qué es un operador?

Un símbolo o palabra que define una operación.

- Aritméticos: `+`, `-`, `*`, `/`, `**`, `//`, `%`.
- Lógicos: `not`, `and`, `or`.
- Comparación: `is`, `is not`, `==`, `!=`, `>`, `>=`, `<`, `<=`.

¿Y un operando?

Cada uno de los elementos sobre los que actúa el operador.

- Valores literales: `5`, `"posit"`, `True`, `1.8`, etc.
- Variables: `x`, `y`, `mi_objeto`, etc.
- Llamadas a funciones: `sum(x, 5)`, `fun(a, b, c, d)`, etc.

Ejemplos

```
1  2 ** 8
2
3
4
5
6
```

Ejemplos

```
2 ** 8

False or True



```

Ejemplos

```
2 ** 8
False or True
int(((a + b) ** 2) / 1.8)


```

Ejemplos

```
2 ** 8
False or True
int(((a + b) ** 2) / 1.8)
12 + sum(x, y, z)


```

Ejemplos

```
2 ** 8
False or True
int(((a + b) ** 2) / 1.8)
12 + sum(x, y, z)
1

```

Ejemplos

```
2 ** 8
False or True
int(((a + b) ** 2) / 1.8)
12 + sum(x, y, z)
1
"foo"
```

Ejemplos

```
2 ** 8
False or True
int(((a + b) ** 2) / 1.8)
12 + sum(x, y, z)
1
"foo"
```

Comparación de objetos

Python permite comparar objetos utilizando dos operadores:

- `==`
- `is`

Comparación de objetos

Python permite comparar objetos utilizando dos operadores:

- `==`
- `is`

La diferencia entre ellos es que `==` compara en **valor**, mientras que `is` compara en **identidad**.

Comparación de objetos

Python permite comparar objetos utilizando dos operadores:

- `==`
- `is`

La diferencia entre ellos es que `==` compara en **valor**, mientras que `is` compara en **identidad**.

La identidad de un objeto se obtiene llamando a `id()`:

```
1  >>> id("bar")
2  139233362987312
```

Veremos ejemplos más adelante.

Operaciones con `str`

Hay objetos que admiten ser utilizados con ciertos operadores.

Por ejemplo, la suma (`+`) y la multiplicación (`*`) tienen sentido con las cadenas de caracteres.

Operaciones con `str`

Hay objetos que admiten ser utilizados con ciertos operadores.

Por ejemplo, la suma (`+`) y la multiplicación (`*`) tienen sentido con las cadenas de caracteres.

Con cadenas de texto, sumar es concatenar.

```
>>> "f" + "a"
'fa'
```

Operaciones con `str`

Hay objetos que admiten ser utilizados con ciertos operadores.

Por ejemplo, la suma (`+`) y la multiplicación (`*`) tienen sentido con las cadenas de caracteres.

Con cadenas de texto, sumar es concatenar.

```
>>> "f" + "a"
'fa'
```

Y donde vale sumar, vale que multiplicar por `n` es sumar el valor `n` veces:

```
>>> "z" * 3
'zzz'
```

Métodos

Los objetos también pueden ofrecer acciones que no se implementan a través de operadores, sino de métodos.

Métodos

Los objetos también pueden ofrecer acciones que no se implementan a través de operadores, sino de métodos.

Por ahora, conformémonos con decir que un método es una función a la que se puede acceder a través de un objeto.

Métodos

Los objetos también pueden ofrecer acciones que no se implementan a través de operadores, sino de métodos.

Por ahora, conformémonos con decir que un método es una función a la que se puede acceder a través de un objeto.

Las cadenas de texto implementan una gran variedad de métodos, utiles para realizar diversas tareas.

Métodos de `str`

Cambiar la capitalización:

```
>>> texto = "Hola, qué tal?"
```

Métodos de `str`

Cambiar la capitalización:

```
>>> texto = "Hola, qué tal?"
>>> texto.upper()
'HOLA, QUÉ TAL?'
```

Métodos de `str`

Cambiar la capitalización:

```
>>> texto = "Hola, qué tal?"
>>> texto.upper()
'HOLA, QUÉ TAL?'
>>> texto.lower()
'hola, qué tal?'
```

Métodos de `str`

Cambiar la capitalización:

```
>>> texto = "Hola, qué tal?"
>>> texto.upper()
'HOLA, QUÉ TAL?'
>>> texto.lower()
'hola, qué tal?'
>>> texto.title()
'Hola, Qué Tal?'
```

Métodos de `str`

Cambiar la capitalización:

```
>>> texto = "Hola, qué tal?"
>>> texto.upper()
'HOLA, QUÉ TAL?'
>>> texto.lower()
'hola, qué tal?'
>>> texto.title()
'Hola, Qué Tal?'
>>> texto.capitalize()
'Hola, qué tal?'
```

Métodos de `str`

Eliminar espacios en blanco

```
>>> texto = "    algún texto   "
```

Métodos de `str`

Eliminar espacios en blanco

```
>>> texto = "    algún texto   "
>>> texto.rstrip()
'    algún texto'
```

Métodos de `str`

Eliminar espacios en blanco

```
>>> texto = "    algún texto   "
>>> texto.rstrip()
'    algún texto'
>>> texto.lstrip()
'algún texto   '
```

Métodos de `str`

Eliminar espacios en blanco

```
>>> texto = "    algún texto   "
>>> texto.rstrip()
'    algún texto'
>>> texto.lstrip()
'algún texto   '
>>> texto.strip()
'algún texto'
```

Métodos de `str`

Eliminar espacios en blanco

```
>>> texto = "    algún texto   "
>>> texto.rstrip()
'    algún texto'
>>> texto.lstrip()
'algún texto   '
>>> texto.strip()
'algún texto'
>>> texto.strip().capitalize() # Encadenamiento de métodos
'Algún texto'
```

f-*strings*

Las f-*strings* permiten **interpolar** cadenas de texto.

f-*strings*

Las f-*strings* permiten **interpolar** cadenas de texto.

Sin f-*strings*:

```
>>> nombre = "Tomás"
>>> edad = 30
>>> print("Soy " + nombre + " y tengo " + str(edad) + " años.")
Soy Tomás y tengo 30 años.
```

f-*strings*

Las f-*strings* permiten **interpolar** cadenas de texto.

Sin f-*strings*:

```
>>> nombre = "Tomás"
>>> edad = 30
>>> print("Soy " + nombre + " y tengo " + str(edad) + " años.")
Soy Tomás y tengo 30 años.
```

Con f-*strings*:

```
>>> nombre = "Tomás"
>>> edad = 30
>>> print(f"Soy {nombre} y tengo {edad} años.")
Soy Tomás y tengo 30 años.
```

Funciones

¿Por qué usamos funciones?

- Reutilización de código

¿Por qué usamos funciones?

- Reutilización de código
- Organización

¿Por qué usamos funciones?

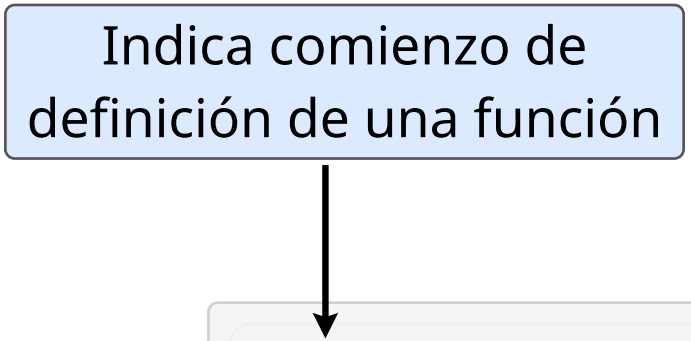
- Reutilización de código
- Organización
- Modularidad

¿Por qué usamos funciones?

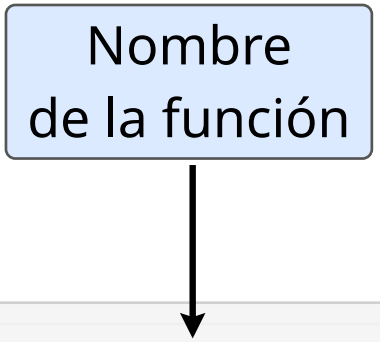
- Reutilización de código
- Organización
- Modularidad

Nada que ustedes no supieran ya 😉

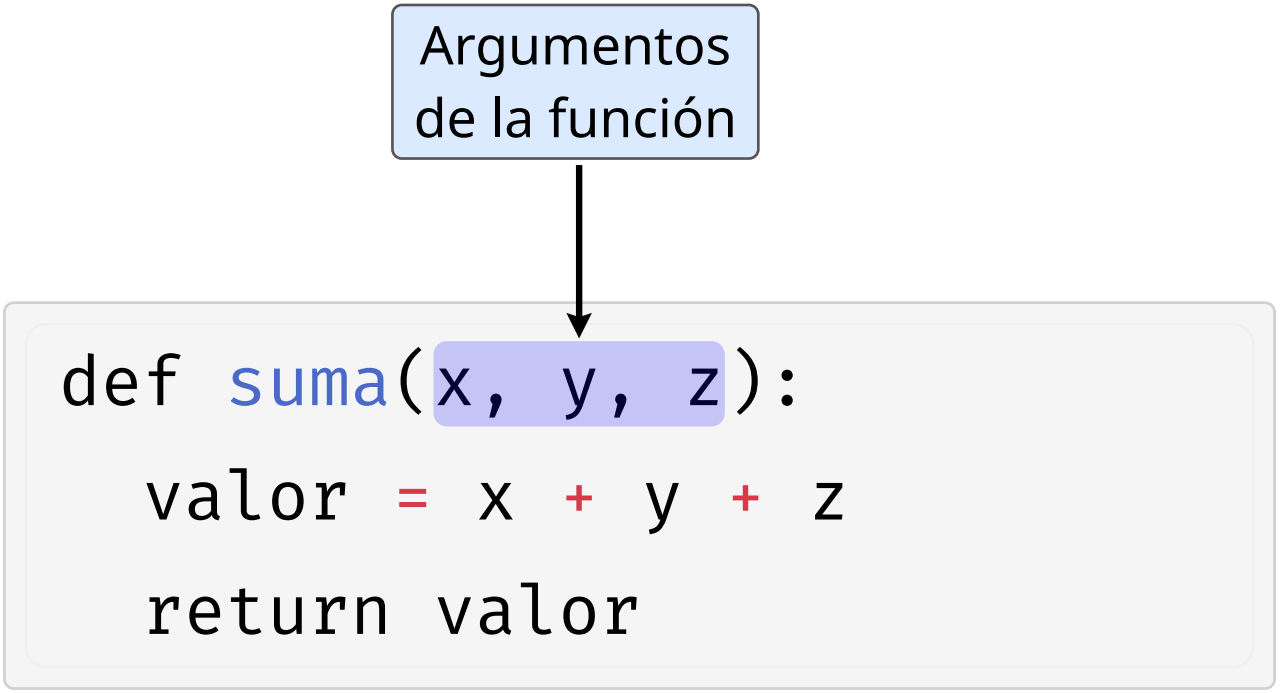
```
def suma(x, y, z):
    valor = x + y + z
    return valor
```

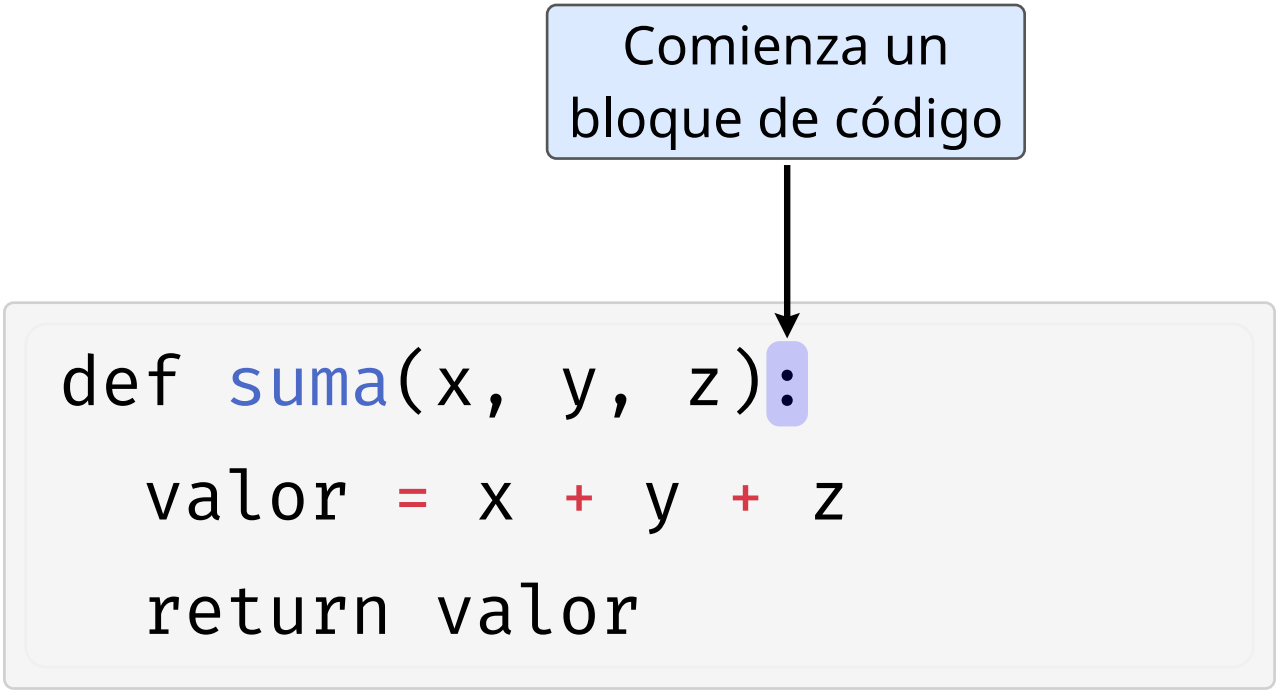



```
def suma(x, y, z):
    valor = x + y + z
    return valor
```




```
def suma(x, y, z):
    valor = x + y + z
    return valor
```


Argumentos
de la función
def suma(x, y, z):
valor = x + y + z
return valor


Comienza un
bloque de código
def suma(x, y, z):
valor = x + y + z
return valor

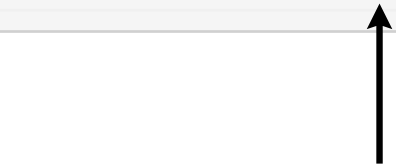
```
def suma(x, y, z):
    valor = x + y + z
    return valor
```

Código que se ejecuta cuando se llama a la función

```python
def suma(x, y, z):
    valor = x + y + z
    return valor
```


Indica que se devuelve un valor y el fin de la ejecución de la función

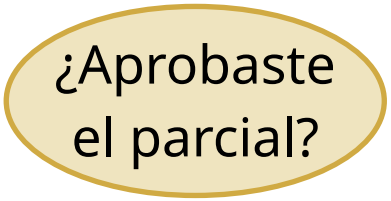
```python
def suma(x, y, z):
    valor = x + y + z
    return valor
```

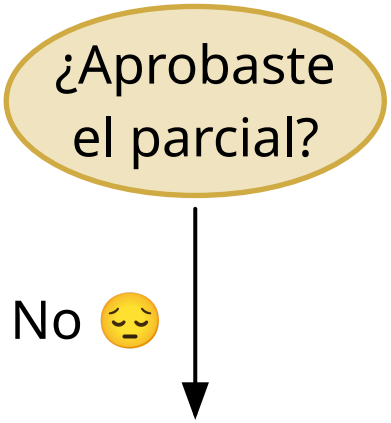


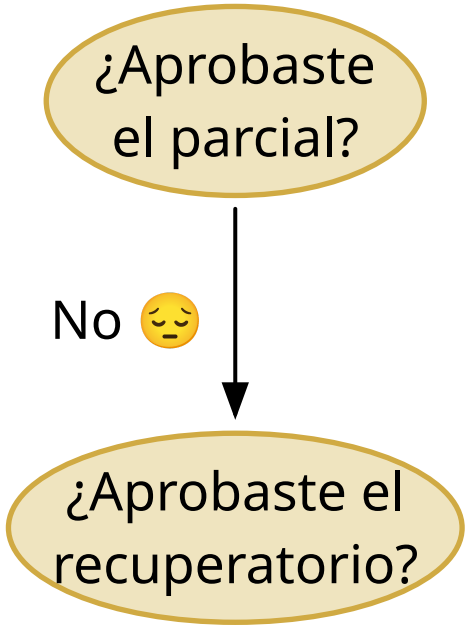
Valor que devuelve la función

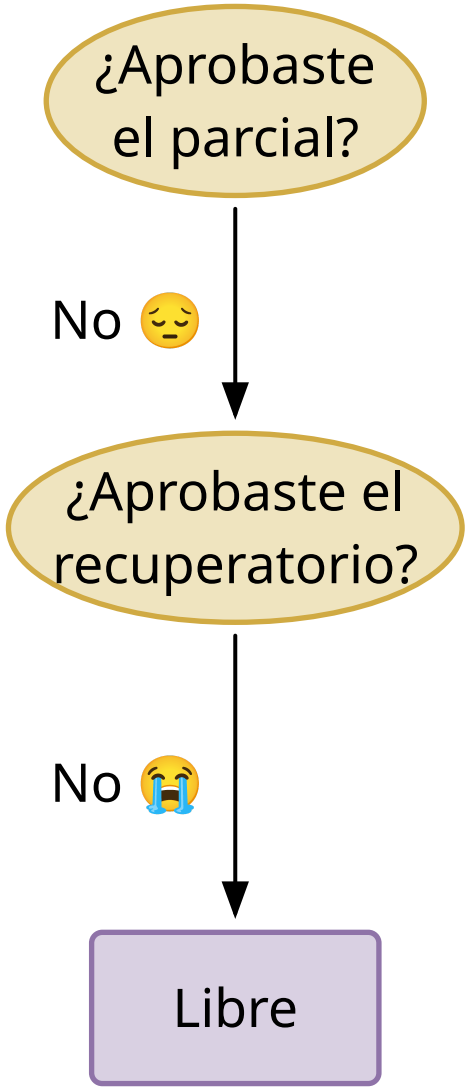
A laburar

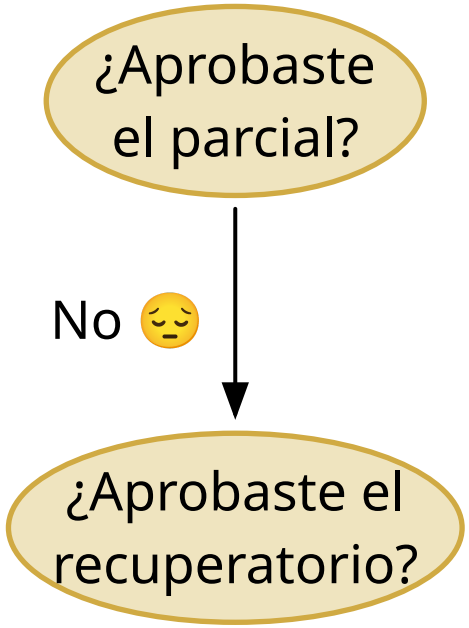
Ejecución condicional

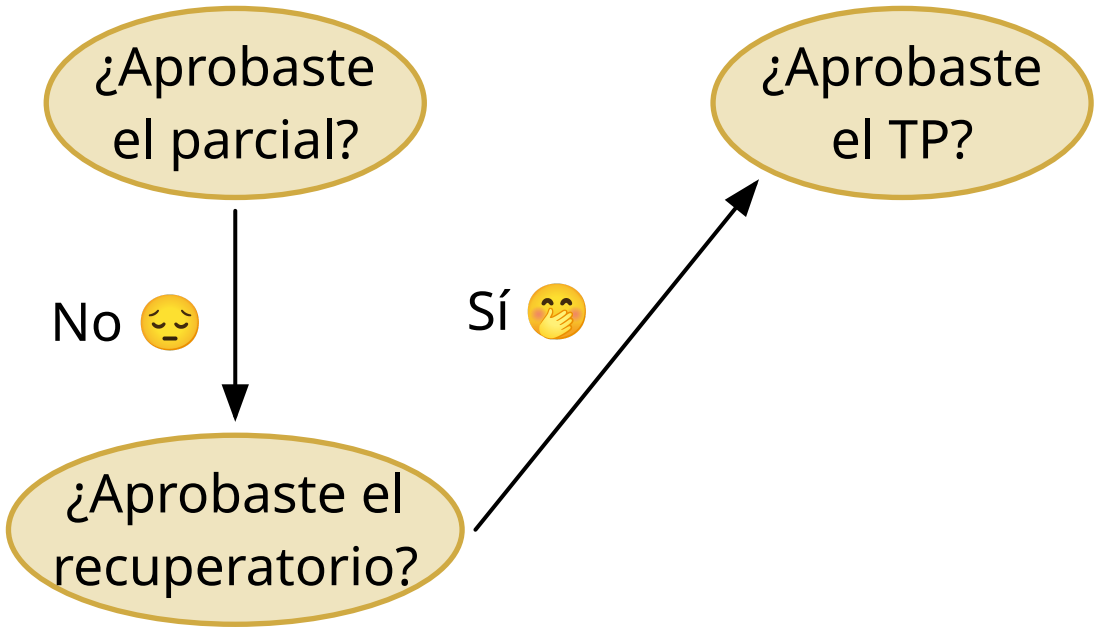

¿Aprobaste
el parcial?

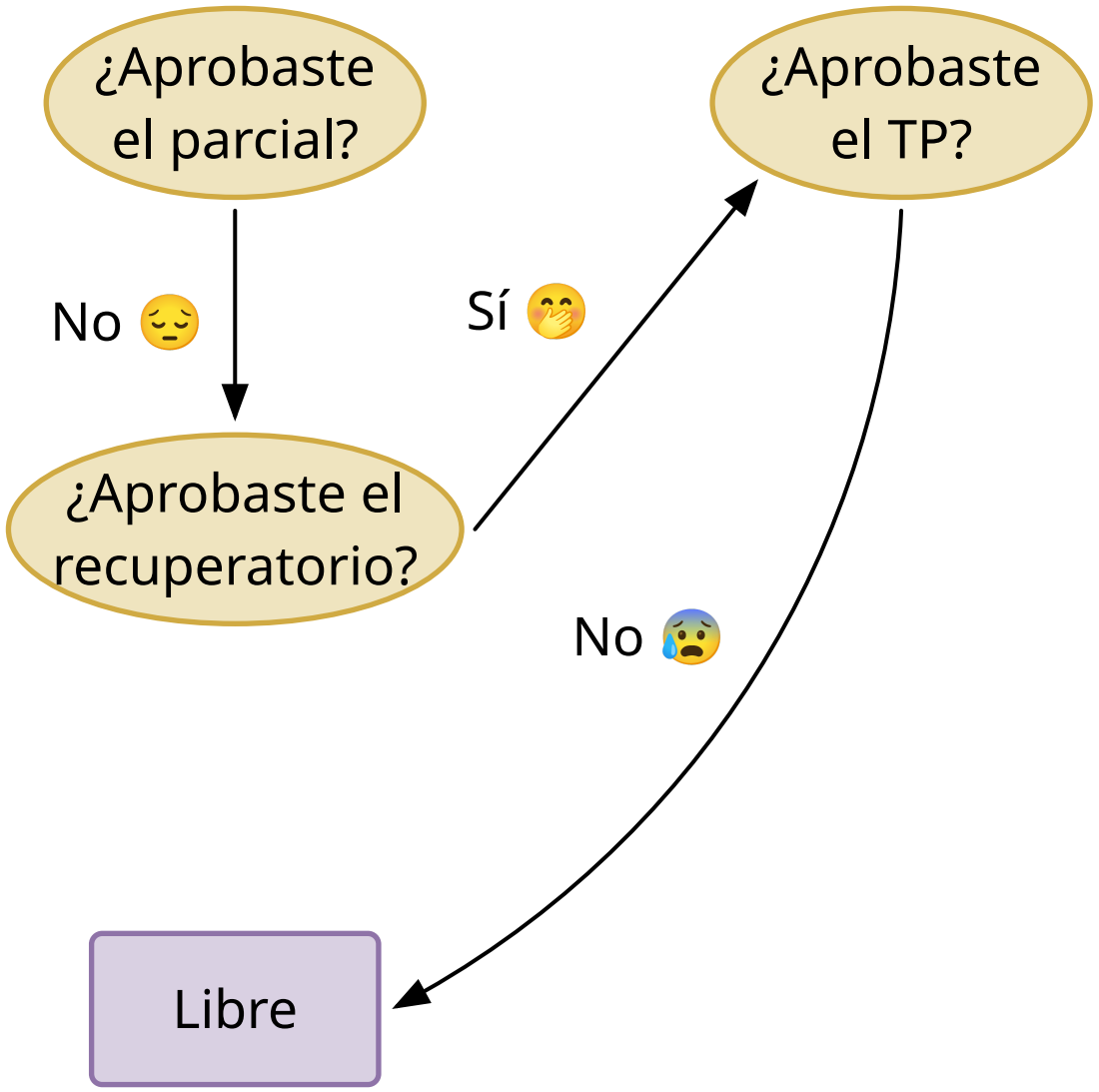

¿Aprobaste el parcial?
No 😔

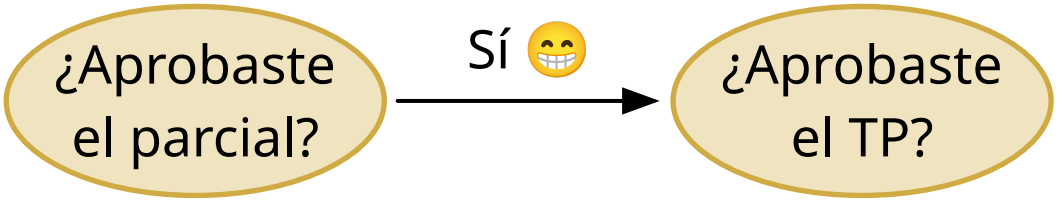

¿Aprobaste el parcial?
No 😔
¿Aprobaste el recuperatorio?

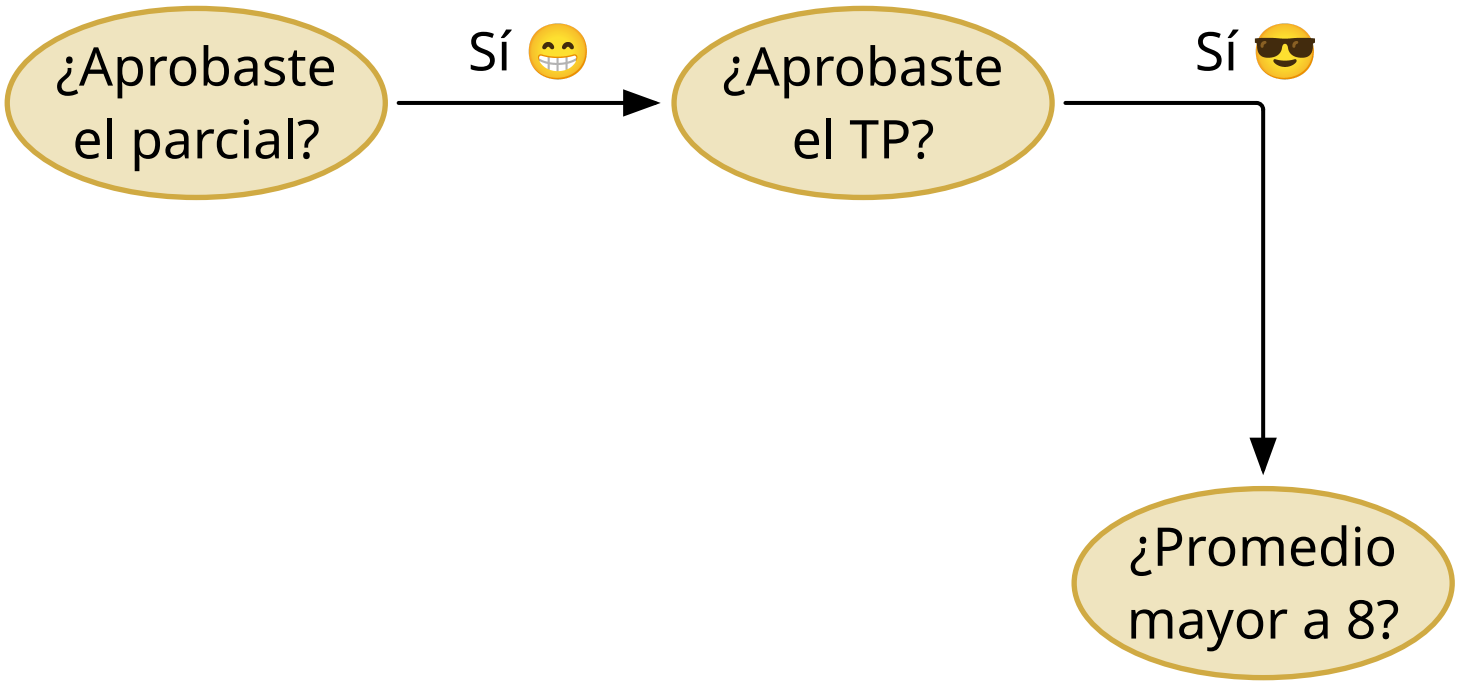

¿Aprobaste el parcial?
No 😔
¿Aprobaste el recuperatorio?
No 😭
Libre

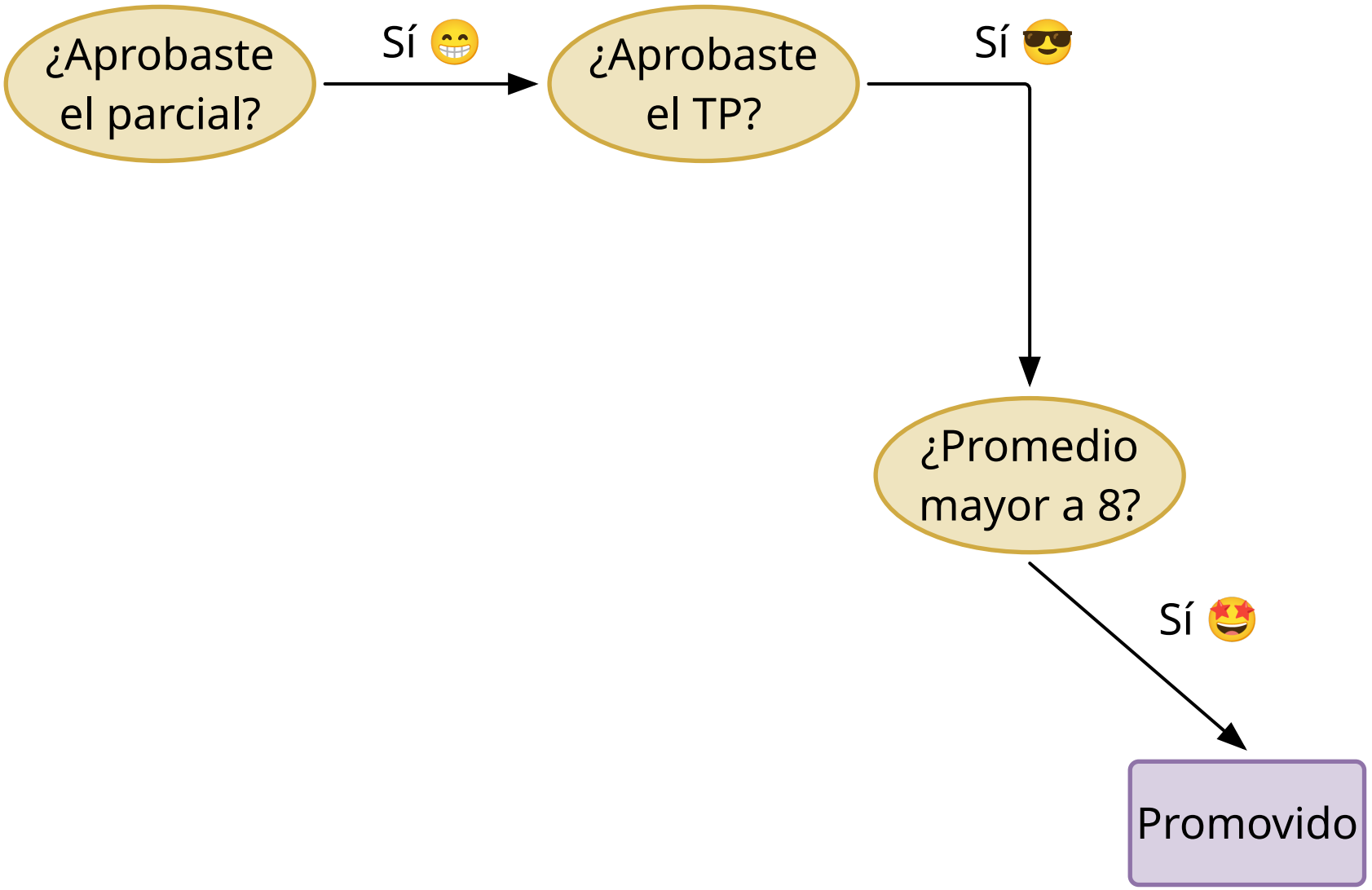

¿Aprobaste el parcial?
No 😔
¿Aprobaste el recuperatorio?

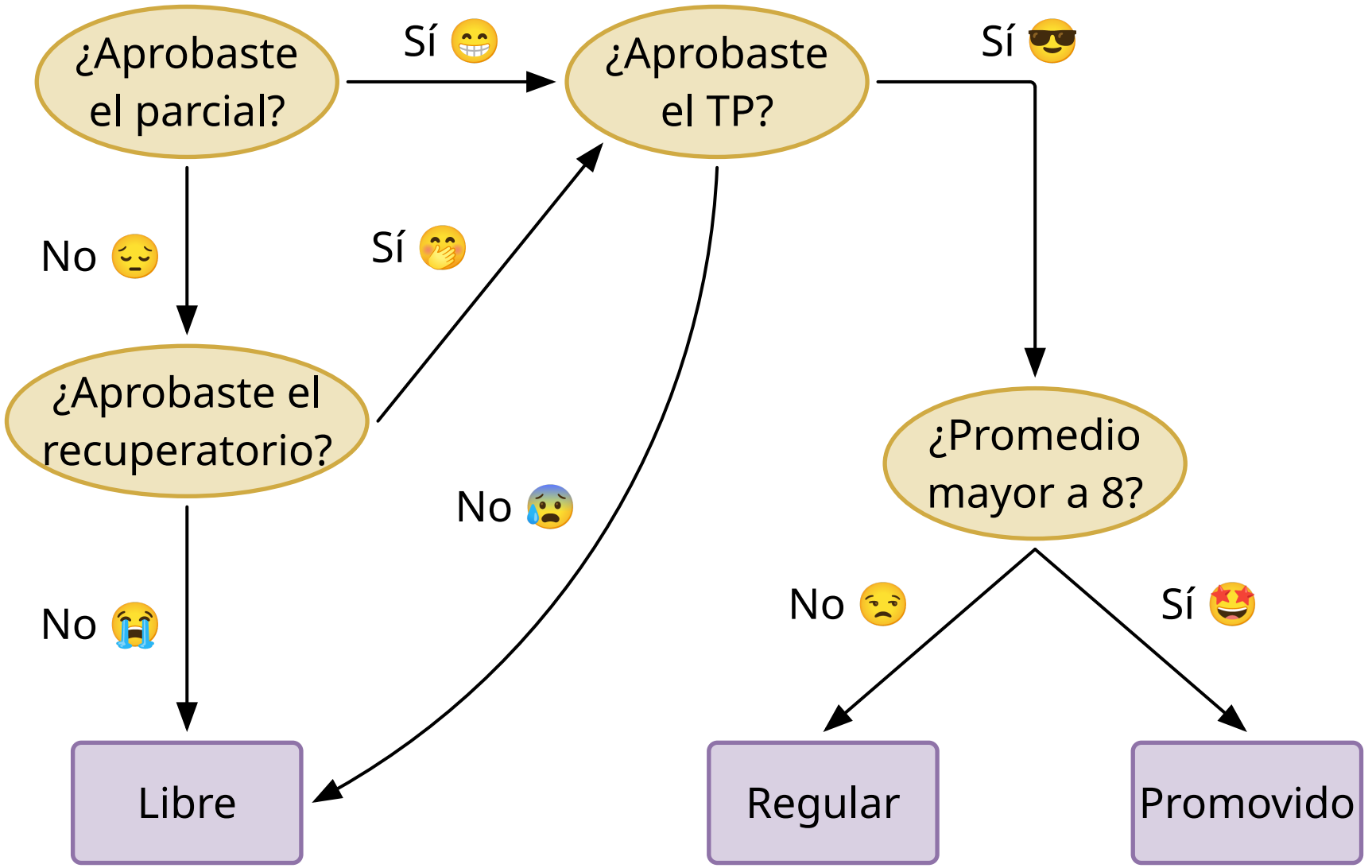

¿Aprobaste el parcial?
No 😔
¿Aprobaste el recuperatorio?
Sí 🤭
¿Aprobaste el TP?


¿Aprobaste el parcial?
No 😔
¿Aprobaste el recuperatorio?
Sí 🤭
¿Aprobaste el TP?
No 😰
Libre


¿Aprobaste el parcial?
Sí 😁
¿Aprobaste el TP?


¿Aprobaste el parcial?
Sí 😁
¿Aprobaste el TP?
Sí 😎
¿Promedio mayor a 8?


¿Aprobaste el parcial?
Sí 😁
¿Aprobaste el TP?
Sí 😎
¿Promedio mayor a 8?
Sí 🤩
Promovido


¿Aprobaste el parcial?
Sí 😁
¿Aprobaste el TP?
Sí 😎
No 😔
Sí 🤭
¿Aprobaste el recuperatorio?
¿Promedio mayor a 8?
No 😰
No 😭
No 😒
Sí 🤩
Libre
Regular
Promovido

Ejecución condicional

- Realizar diferentes acciones según qué se responde a cada pregunta

Ejecución condicional

- Realizar diferentes acciones según qué se responde a cada pregunta
- Preguntas ⟶ Condiciones

Ejecución condicional

- Realizar diferentes acciones según qué se responde a cada pregunta
- Preguntas ⟶ Condiciones
- Posible gracias a las siguientes **estructuras de control**:
  - `if`
  - `if-else`
  - `if-elif-else`

Partes de estructura `if`

```
if nota >= 6:
    print("Aprobado")
```

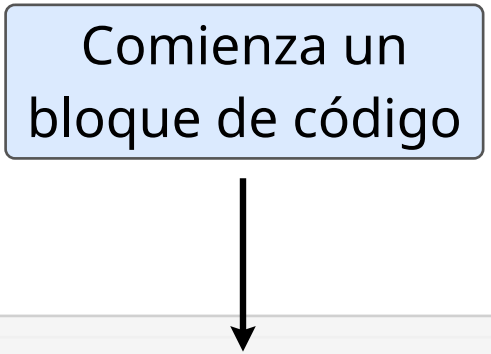
Partes de estructura `if`

Partes de estructura `if`

Expresión, llamada condición,
que al evaluarse resulta en `True` o `False`

```
if nota >= 6:
    print("Aprobado")
```

Partes de estructura `if`




```
if nota >= 6:
    print("Aprobado")
```

Partes de estructura `if`

```
if nota >= 6:
    print("Aprobado")
```

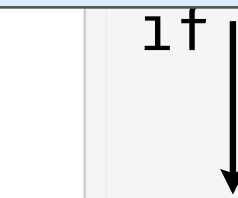

Bloque que se ejecuta si la condición es verdadera

Partes de estructura `if-else`

```
if nota >= 6:
    print("Aprobado")
else:
    print("Desaprobado")
```

Partes de estructura `if-else`

Declara bloque que se ejecuta cuando la condición del `if` no es verdadera.
Palabra clave es `else`.



```
if nota >= 6:
    print("Aprobado")
else:
    print("Desaprobado")
```

Partes de estructura `if-else`

```
if nota >= 6:
    print("Aprobado")
else:
    print("Desaprobado")
```


Bloque que se ejecuta
solo cuando
la condición del `if` es `False`

Partes de estructura `if-elif-else`

```
if nota >= 8:
    print("Promovido")
elif nota >= 6:
    print("Regular")
else:
    print("Desaprobado")
```

Partes de estructura `if-elif-else`

Primera condición a evaluar

```
if nota >= 8:
    print("Promovido")
elif nota >= 6:
    print("Regular")
else:
    print("Desaprobado")
```

Partes de estructura `if-elif-else`

```
if nota >= 8:
    print("Promovido")
elif nota >= 6:
    print("Bloque que se ejecuta si nota >= 8")
else:
    print("Desaprobado")
```

Partes de estructura `if-elif-else`

```
if nota >= 8:
    print("Promovido")
elif nota >= 6:
    print("Regular")
else:
    print("Desaprobado")
```

Si la nota no es mayor o igual al 8, se evalúa si es mayor o igual a 6. La palabra clave es `elif`.

Partes de estructura `if-elif-else`

```
if nota >= 8:
    print("Promovido")
elif nota >= 6:
    print("Regular")
else:
    print("Desaprobado")
```

Bloque que se ejecuta si `nota >= 6`

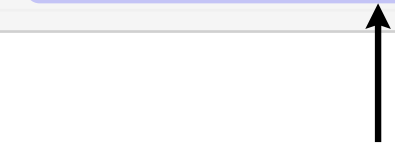
Partes de estructura `if-elif-else`

```
if nota >= 8:
    print("Promovido")
elif nota >= 6:
    print("Regular")
else:
    print("Desaprobado")
```


Declara bloque alternativo.
Se ejecuta cuando nota no es mayor o igual a 8, ni mayor o igual a 6.

Partes de estructura `if-elif-else`

```
if nota >= 8:
    print("Promovido")
elif nota >= 6:
    print("Regular")
else:
    print("Desaprobado")
```



Código del bloque alternativo

A laburar

Bucles

Ejecución repetitiva

- Ejecutar la misma acción para cada valor en una secuencia.

Ejecución repetitiva

- Ejecutar la misma acción para cada valor en una secuencia.
- Ejecutar una acción hasta que se cumpla (o se deje de cumplir) una condición.

Ejecución repetitiva

- Ejecutar la misma acción para cada valor en una secuencia.
- Ejecutar una acción hasta que se cumpla (o se deje de cumplir) una condición.
- No hace falta copiar y pegar el mismo código todo el tiempo...

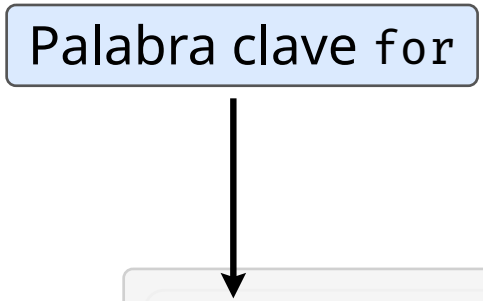
Ejecución repetitiva

- Ejecutar la misma acción para cada valor en una secuencia.
- Ejecutar una acción hasta que se cumpla (o se deje de cumplir) una condición.
- No hace falta copiar y pegar el mismo código todo el tiempo...
- gracias a las siguientes **estructuras de control**:
  - `for`
  - `while`

Partes de estructura `for`

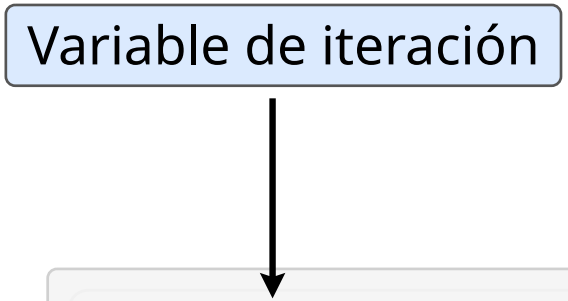
```
for i in range(3):
    print(f"El numero es {i}")
```

Partes de estructura `for`



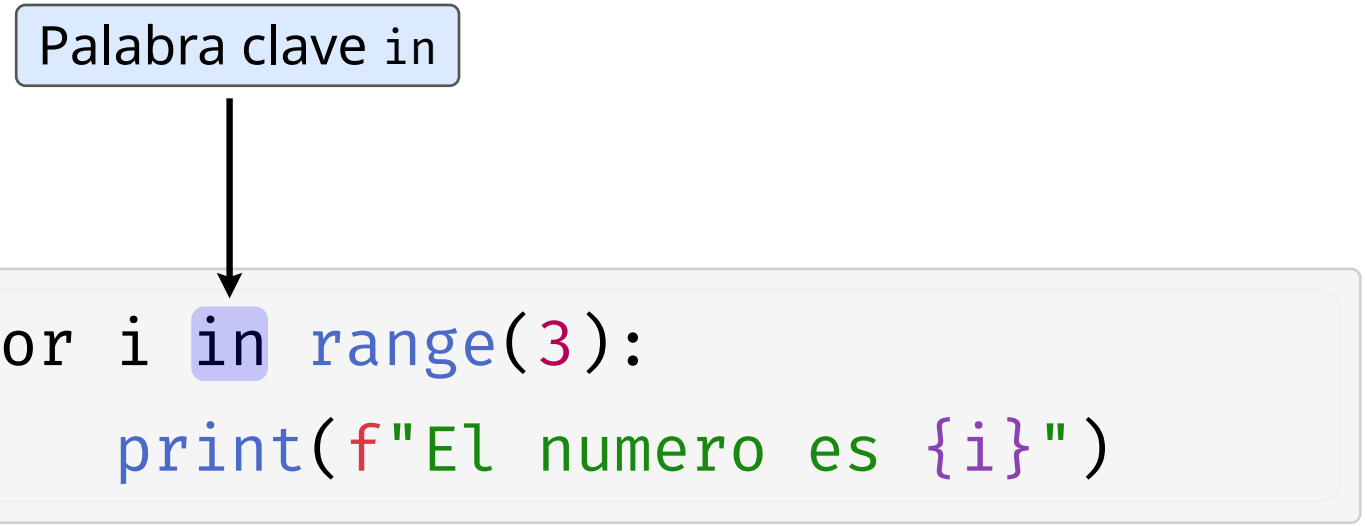

```python
for i in range(3):
    print(f"El numero es {i}")
```

Partes de estructura `for`

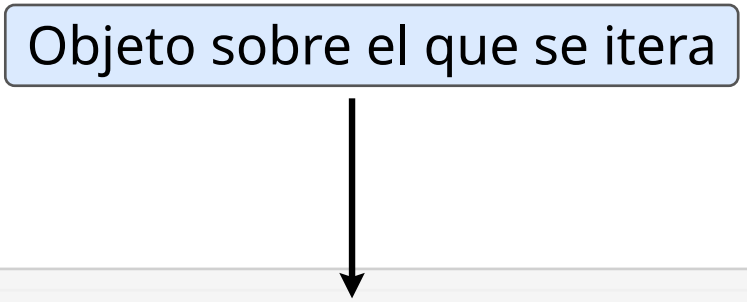



```
for i in range(3):
    print(f"El numero es {i}")
```

Partes de estructura `for`

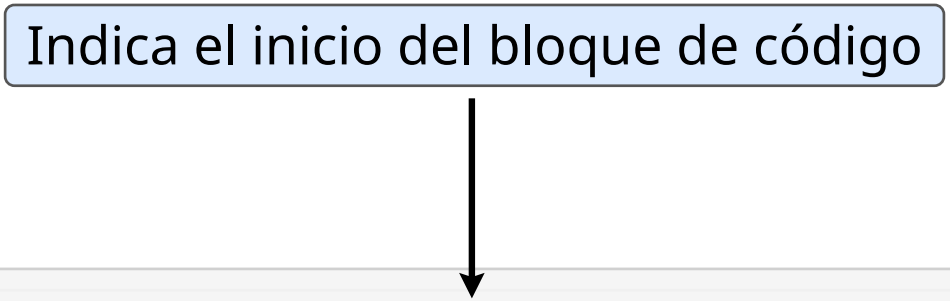


Partes de estructura `for`




```
for i in range(3):
    print(f"El numero es {i}")
```

Partes de estructura `for`




```
for i in range(3):
    print(f"El numero es {i}")
```

Partes de estructura `for`

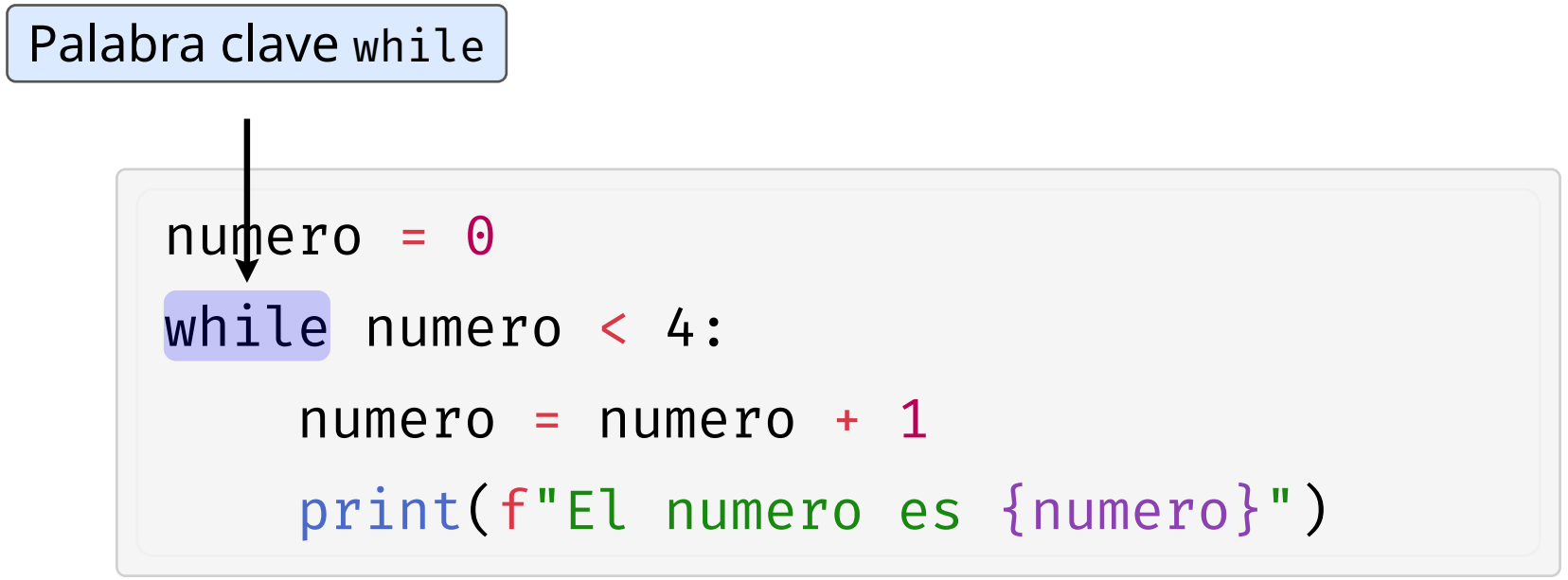
```python
for i in range(3):
    print(f"El numero es {i}")
```


Bloque de código que se ejecuta en cada iteración

Partes de estructura `while`

```
numero = 0
while numero < 4:
    numero = numero + 1
    print(f"El numero es {numero}")
```

Partes de estructura `while`

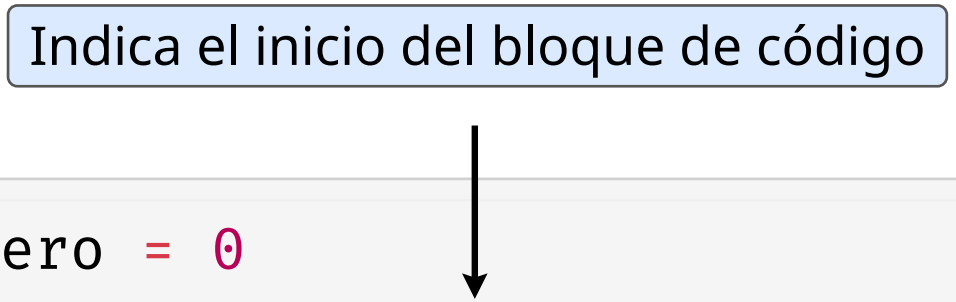


Partes de estructura `while`

Condición que se evalúa a `True` o `False`


```
numero = 0
while numero < 4:
    numero = numero + 1
    print(f"El numero es {numero}")
```

Partes de estructura `while`




```
numero = 0
while numero < 4:
    numero = numero + 1
    print(f"El numero es {numero}")
```

Partes de estructura `while`

```
numero = 0
while numero < 4:
    numero = numero + 1
    print(f"El numero es {numero}")
```

Bloque de código que se ejecuta en cada iteración mientras la condición sea verdadera

A laburar

Fin