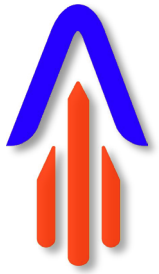
3. Conceptos básicos (2/2)

Introducción a Python para Deep Learning



0. Introducción (recordatorio)

- En las siguientes dos sesiones (ya estamos en la segunda) veremos los conceptos básicos de Python, desde variables hasta clases, pasando por tipos de datos
- Sin una base sólida sería imposible afrontar lo que queda por delante de este curso, y mucho menos los cursos más avanzados del programa

0. Introducción (recordatorio)

En la unidad anterior vimos:

1. Documentación
2. Configuración de VSC
3. Entrada y salida básica
4. Variables y tipos de datos
5. Operadores y expresiones
6. Estructuras de control
7. Cadenas de caracteres

Contenidos

8. Estructuras de datos (listas, tuplas, diccionarios y conjuntos)
9. Funciones
10. Programación orientada a objetos
11. Archivos
12. Gestión de errores

8. Estructuras de datos

1. Listas
2. Tuplas
3. Conjuntos
4. Diccionarios
5. Enumeraciones
6. Manipulación de secuencias
7. Mutabilidad e inmutabilidad
8. Iteración sobre estructuras de datos
9. Comprensión de listas y diccionarios

8.1. Listas

- Colección ordenada de elementos que pueden ser de cualquier tipo (números, cadenas, booleanos, etcétera)
- Acceso mediante índices (como vimos con las cadenas)
- También funciona el *slicing*
- Modificación de elementos mediante índices y asignación

```
# Creación
lista = [1, 2, 3, "cuatro", True]

# Acceso
print(lista[0])   # 1
print(lista[-1])  # True

# Modificación
lista[1] = "dos"
```

8.1. Listas

Operaciones básicas:

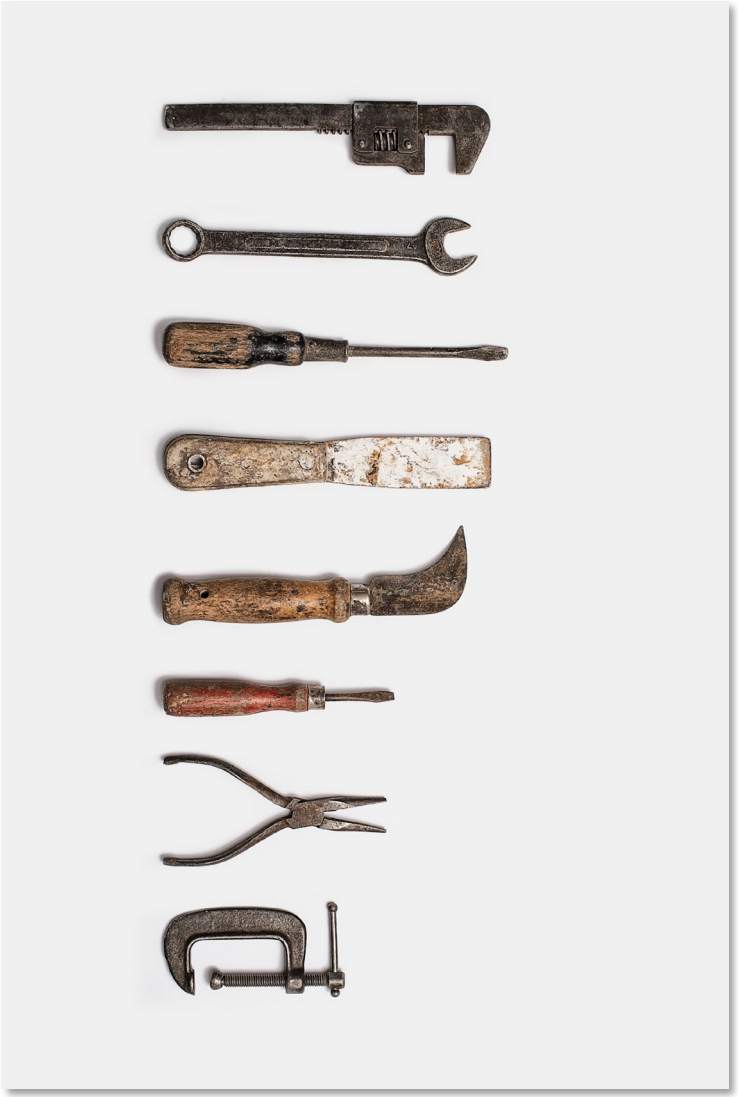
- `append()`: añadir al final
- `insert()`: añadir en una posición
- `remove()`: eliminar valor
- `pop()`: sacar el primero
- `del`: eliminar rango

```python
# Creación de la lista
lista = [1, 2, 3, "cuatro", True]

# Añadir al final
lista.append(5)

# Insertar en una posición
lista.insert(2, "nuevo")

# Eliminar por valor
lista.remove("cuatro")

# Sacar el primero
lista.pop(0)

# Eliminar un rango
del lista[1:3]
```

8.1. Listas

Utilidades:

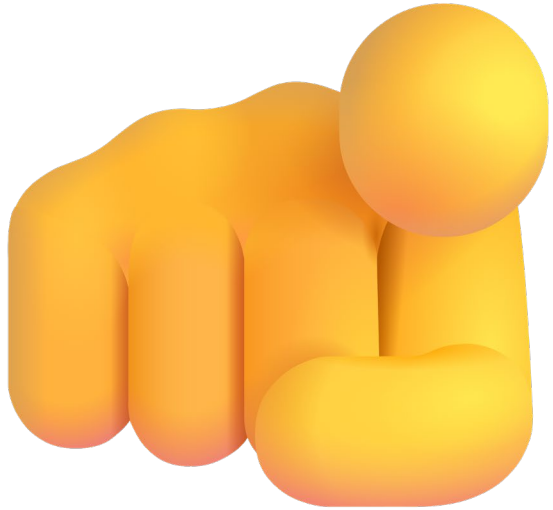
- `len()`: longitud
- `sort()`, `sorted()`: ordenar
- `reverse()`: invertir
- `count()`: contar
- `index()`: encontrar



8.1. Listas

Tienes una lista de nombres de estudiantes que han asistido a una clase, sin orden en particular y pueden repetirse. Gestiónala utilizando las funciones de listas:

1. Ana, Carlos, Beatriz, Ana, David, Beatriz, Eduardo
2. ¿Número total de estudiantes?
3. Ordena alfabéticamente
4. Ordena forma inversa alfabéticamente
5. ¿Cuántas veces aparece Ana?
6. Encuentra la posición de la primera aparición de Beatriz
7. Crea una nueva lista ordenada sin modificar la original



8.1. Listas

Recorrer listas:

- las listas son iteradores, es decir, podemos utilizar bucles `for` para pasar por todos sus elementos
- obviamente, también podemos hacerlo con bucles `while`, aunque necesitamos esforzarnos un poco más

```
lista = [1, 2, 3, "cuatro", True]

for elemento in lista:
    print(elemento)

index = 0
while index < len(lista):
    print(lista[index])
    index += 1
```

8.1. Listas

Listas anidadas:

- una lista puede contener otras listas
- serían listas multidimensionales
- lo veremos al hablar de NumPy

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

# 1 2
# 3 4
# 5 6

print(lista_anidada[1][0])
# 3
```

8.1. Listas

Comprensión de listas:

- creación de listas de forma compacta
- para quienes le gusten los *one-liners*
- priorizar claridad (en mi opinión)

```python
cuantos = 5
# [0, 1, 4, 9, 16]

cuadrados = [x**2 for x in range(cuantos)]
print(cuadrados)

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

8.1. Listas

Copiar listas:

- mirad este código y decidme qué hace

```
lista = [1, 2, 3, "cuatro", True]

lista_2 = lista
```

8.1. Listas

Copiar listas:

- mirad este código y decidme qué hace
- ¿qué ocurrirá si modifico un elemento de alguna de las dos listas?

```
lista = [1, 2, 3, "cuatro", True]

lista_2 = lista
```

8.1. Listas

Copiar listas:

- mirad este código y decidme qué hace
- ¿qué ocurrirá si modifico un elemento de alguna de las dos listas?
- En Python, la asignación de listas (y otros objetos no elementales) ***es una referencia***

```
lista = [1, 2, 3, "cuatro", True]

lista_2 = lista

lista[0] = "Anda..."
print(lista_2)

# ['Anda...', 2, 3, 'cuatro', True]
```

8.1. Listas

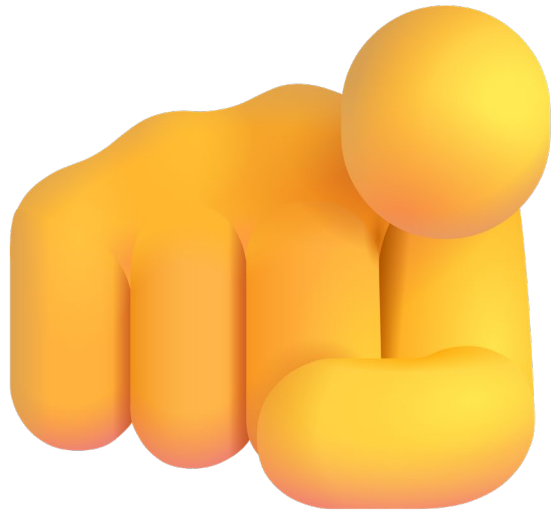
Copiar listas:

- mirad este código y decidme qué hace
- ¿qué ocurrirá si modifico un elemento de alguna de las dos listas?
- En Python, la asignación de listas (y otros objetos no elementales) ***es una referencia***
- Solución: `copy()` o `[:]`

```
lista = [1, 2, 3, "cuatro", True]

lista_2 = lista.copy()
# lista_2 = lista[:]

lista[0] = "Anda..."
print(lista_2)

# [1, 2, 3, 'cuatro', True]
```

8.1. Listas

¡Hagamos un ejercicio!



8.2. Tuplas

- Similares a las listas, pero **inmutables**, es decir, una vez creadas, no se pueden modificar
- Ventaja: eficiencia
- Acceso como en listas
- Iteración como en listas
- Tuplas anidadas como en listas

```
tupla = (1, 2, 3,
"cuatro", True)

# Esto falla
tupla[0] = "uno"

print(tupla[0])   # 1
print(tupla[-1])  # True
```

8.2. Tuplas

- Desempaquetado: asignación de cada elemento a una variable independiente

```
a, b, c = (1, 2, 3)

print(a)  # 1
print(b)  # 2
```

8.2. Tuplas

- Tuplas con un elemento

```
# Esto falla
tupla_un_elemento = (42)
print(tupla_un_elemento)

# Esto no
tupla_un_elemento = (42,)
print(tupla_un_elemento)
```

8.2. Tuplas

Métodos comunes:

- `len()`: número de elementos
- `count()`: cuántas veces un valor
- `index()`: posición valor

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

print(len(tupla))
# 5

print(tupla.count(1))
# 3

print(tupla.index(3))
# 2
```

8.2. Tuplas

Conversión:

- tupla inmutable
- de tupla a listas: mutable
- modificación
- de lista a tupla: inmutable de nuevo

```
tupla = (1, 2, 3)
lista = list(tupla)
lista.append(4)
tupla = tuple(lista)
print(tupla)
# (1, 2, 3, 4)
```

8.2. Tuplas

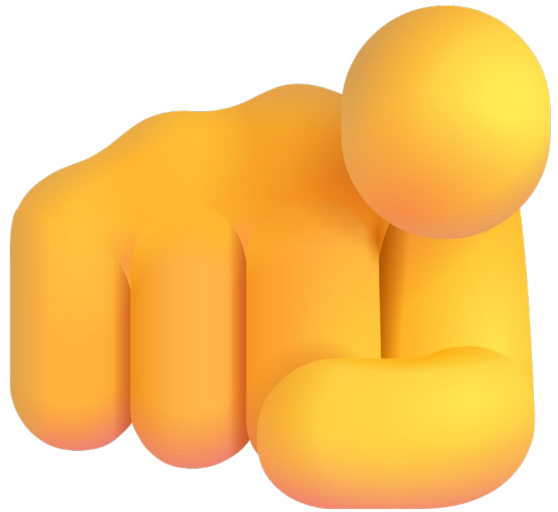
Tuplas con nombre:

- cada elemento tiene un nombre
- el código es más legible, más limpio, más autodocumentado, más *pythonico*

```python
from collections import namedtuple

Punto = namedtuple("Punto", "x y")

punto = Punto(10, 20)

print(punto.x, punto.y)
# 10 20
```

8.2. Tuplas

¡Hagamos un ejercicio!



8.3. Conjuntos

- *Set* en inglés (y en Python)
- Colección no ordenada de valores
- Sin elementos duplicados

```
# Declaración explícita
un_conjunto = {1, 2, 3, 4}

# Conversión de lista
otro_conjunto = set([2, 3, 5, 7])

# Eliminación de duplicados
un_conjunto_distinto = {1, 2, 3, 3, 4}
otro_conjunto_mas = set([2, 3, 3, 5, 7])

print(un_conjunto)
print(otro_conjunto)
print(un_conjunto_distinto)
print(otro_conjunto_mas)
```

8.3. Conjuntos

Operaciones básicas:

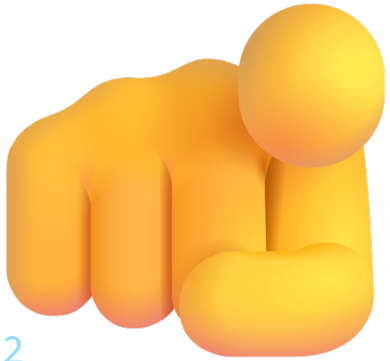
- `add()`: añadir elemento
- `remove()`: eliminar, error si no existe
- `discard()`: eliminar sin error
- `pop()`: sacar uno al azar
- `clear()`: vaciar

```
conjunto = {1, 2, 2, 3, 3, 4}

print(conjunto)
# {1, 2, 3, 4}

# conjunto.remove(7)
# Generaría error

conjunto.remove(3)

conjunto.discard(6)
# No genera error
elemento_eliminado = conjunto.pop()
conjunto.clear()
```

8.3. Conjuntos

Operaciones de conjuntos:

- `union()` o `|`: combina los elementos de dos conjuntos
- `intersection()` o `&`: elementos comunes de dos conjuntos
- `difference()` o `-`: elementos sólo en uno de los conjuntos
- `symmetric_difference()` o `^`: elementos que están en uno o en otro, pero no en ambos



```python
conjunto_1 = {1, 2, 3}
conjunto_2 = {3, 4, 5}

union = conjunto_1 | conjunto_2
print(union)

interseccion = conjunto_1 & conjunto_2
print(interseccion)

diferencia = conjunto_1 - conjunto_2
print(diferencia)

diferencia_simetrica = conjunto_1 ^ conjunto_2
print(diferencia_simetrica)
```

8.3. Conjuntos

Subconjuntos y superconjuntos:

- `issubset()`: para subconjuntos
- `issuperset()`: para superconjuntos

```
conjunto_1 = {1, 2, 3}
conjunto_2 = {3, 4, 5}

union = conjunto_1 | conjunto_2

print(conjunto_1.issubset(union))
# True
print(union.issuperset(conjunto_1))
# True
```

8.3. Conjuntos

Pertenencia:

- verificar si un elemento pertenece a un conjunto
- operador `in`

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

print(3 in conjunto)  # True
print(6 in conjunto)  # False
```

8.3. Conjuntos

Conjuntos inmutables:

- `frozenset()`
- no permiten modificación tras su creación

```
# conjunto = set([1, 2, 3])
conjunto = frozenset([1, 2, 3])

print(conjunto)

conjunto.add(4)

print(conjunto)
```

8.3. Conjuntos

Iteración:

- usando bucles `for`
- usando bucles `while` (?)

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

for elemento in conjunto:
    print(elemento)

# Spoiler: no sale bien
indice = 0
while indice < len(conjunto):
    print(conjunto[indice])
    indice += 1
```

8.3. Conjuntos

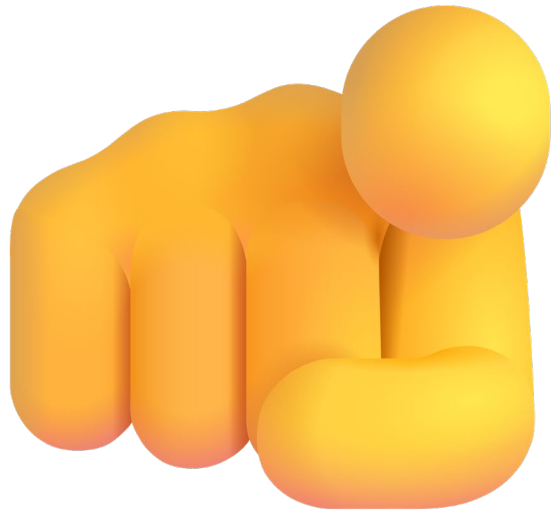
Iteración:

- usando bucles `for`
- usando bucles `while` (?)
  - previa conversión a lista

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

for elemento in conjunto:
    print(elemento)

# Conversión a lista
lista = list(conjunto)
indice = 0
while indice < len(lista):
    print(lista[indice])
    indice += 1
```

8.3. Conjuntos

¡Hagamos un ejercicio!



8.4. Diccionarios

- Colección de pares **clave-valor**
- Cada clave es única
- Se utiliza para acceder al valor asociado

```
diccionario = {"nombre": "Ana", "edad": 25, "ciudad": "Madrid"}
print(diccionario)
```

8.4. Diccionarios

- Colección de pares **clave-valor**
- Cada clave es única
- Se utiliza para acceder al valor asociado
- Quizá quede más claro así

```
diccionario = {
    "nombre": "Ana",
    "edad": 25,
    "ciudad": "Madrid"
}
print(diccionario)
```

8.4. Diccionarios

- Acceso a valores a través de la clave, entre corchetes, tras el nombre de la variable
- Si la clave no existe, se producirán un error
- Para evitarlo, se puede usar `get()`, proporcionando un valor predeterminado para el caso de que la clave no exista

```
# ...

print(diccionario["nombre"])
# "Ana"

print(diccionario.get("edad"))
# 25

# print(diccionario["pais"])
# ¡Error!

print(diccionario.get("pais", "No disponible"))
# "No disponible"
```

8.4. Diccionarios

- Modificación de valores accediendo a la clave y asignando el nuevo valor
- Atención: si la clave no existe, se agregará un nuevo elemento

```
# ...

print(diccionario["edad"])
# 25

diccionario["edad"] = 26
print(diccionario["edad"])
# 26

diccionario["edad"] = diccionario["edad"] + 1
print(diccionario["edad"])
# 27

print(diccionario.get("pais", "No disponible"))
# "No disponible"
diccionario["pais"] = "España"
print(diccionario.get("pais", "No disponible"))
# España
```

8.4. Diccionarios

Métodos:

- `pop()`: eliminar por clave, obtener
- `del`: eliminar sin obtener
- `clear()`: vaciar

```
# ...

diccionario["pais"] = "España"
print(diccionario)

valor = diccionario.pop("ciudad")
print(diccionario)
print(valor)

del diccionario["edad"]
print(diccionario)

diccionario.clear()
print(diccionario)
```

8.4. Diccionarios

Mas métodos:

- `len()`: número de pares
- `keys()`: lista de claves
- `values()`: lista de valores
- `items()`: lista de tuplas clave-valor

```
# ...

print(len(diccionario))
# Número de pares clave-valor
print(diccionario.keys())
# Claves
print(diccionario.values())
# Valores
print(diccionario.items())
# Pares clave-valor
```

8.4. Diccionarios

Comprobación de claves:

- operador `in` usando la clave correspondiente

```
# ...

if "pais" in diccionario:
    print(f"País: {diccionario["pais"]}")

if "pais" not in diccionario:
    print("País no disponible")
```

8.4. Diccionarios

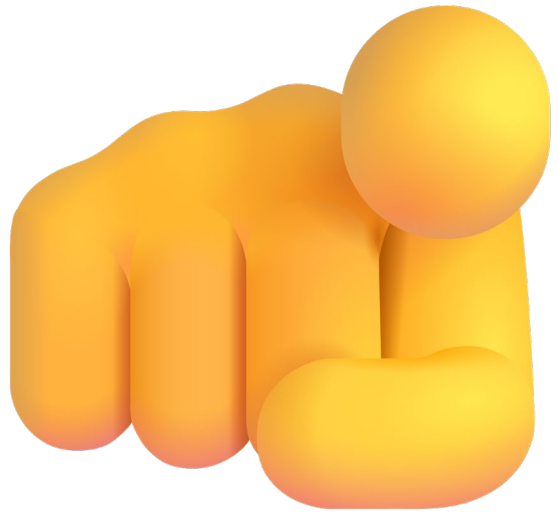
Recorrer diccionarios:

- usando bucles `for`
- esta vez no vamos a intentarlo con bucles `while`, aunque se podría, porque... ¿para qué?

```python
diccionario = {
    "nombre": "Ana",
    "edad": 25,
    "ciudad": "Madrid"
}

for clave in diccionario:
    print(f"{clave}: {diccionario[clave]}")

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

8.4. Diccionarios

¡Hagamos un ejercicio!



8.5. Enumeraciones

Supongamos que tenemos que referirnos a los días de la semana desde el código

- Opción 1: literales de texto
- Opción 2: variables con nombre (porque no tenemos constantes en Python)

Problemas opción 1:

- "lunes", "Lunes", "LUNES", "lUNES", "Lumes"...

Problemas opción 2:

- Declaración: Lunes = "Lunes"
- Posibles asignaciones posteriores

8.5. Enumeraciones

Solución: enumeraciones

- evitan la introducción de errores
- los IDE suelen autocompletar
- agrupan valores simbólicos bajo un mismo nombre
- permiten crear un conjunto de constantes con nombres legibles
- cada miembro de la enumeración tiene un nombre y un valor único

```
from enum import Enum

class DiaSemana(Enum):
    Lunes = 1
    Martes = 2
    Miercoles = 3
    Jueves = 4
    Viernes = 5

dia = DiaSemana.Lunes

print(dia)
print(dia.name)
print(dia.value)
```

8.5. Enumeraciones

Se puede iterar por todos los posibles valores que tiene una enumeración

```
# ...

for dia in DiaSemana:
    print(dia)
    print(dia.name)
    print(dia.value)
```

8.5. Enumeraciones

Control de flujo usando enumeraciones

```
# ...

dia_actual = DiaSemana.Viernes

if dia_actual == DiaSemana.Lunes:
    print("Comienza la semana")
elif dia_actual == DiaSemana.Viernes:
    print("Casi es fin de semana")
```

8.5. Enumeraciones

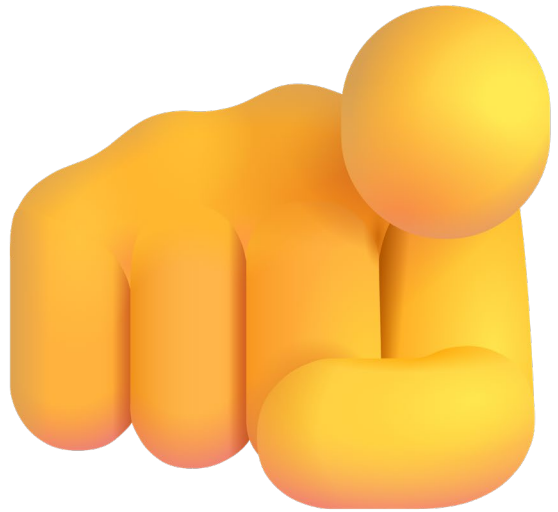
Enumeraciones con atributos personalizados

Usando `namedtuple` (que vimos anteriormente)

```python
from collections import namedtuple
from enum import Enum

MesItem = namedtuple(
    typename='MesItem',
    field_names=[
        'nombre',
        'posicion',
        'dias'])
class Mes(MesItem, Enum):
    Enero = MesItem(
        nombre='Enero',
        posicion=0,
        dias=31)
    # ...
    Diciembre = MesItem(
        nombre='Diciembre',
        posicion=11,
        dias=31)

mes = Mes.Enero
print(mes.nombre)
print(mes.posicion)
print(mes.dias)
```

8.5. Enumeraciones

¡Hagamos un ejercicio!



9. Funciones

1. Definición de funciones
2. Argumentos y parámetros
3. Ámbito de las variables
4. Retorno de valores
5. Anotaciones de tipos
6. Documentación
7. Funciones como objetos de primera categoría
8. Funciones anónimas

9.1. Definición de funciones

- Función: bloque de código reutilizable
- Debería realizar una tarea específica
- Permiten organizar el código, evitar la repetición y hacer que los programas sean más modulares y fáciles de mantener
- En Python, se definen con la palabra clave `def`, seguidas de un nombre, un conjunto opcional de parámetros entre paréntesis, y un bloque de código indentado que se ejecuta cuando la función es llamada
- Pueden recibir entradas (argumentos), realizar cálculos o acciones, y opcionalmente devolver un valor utilizando la palabra clave `return`

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

saludar()
```

9.1. Definición de funciones

- Consejo: usar `pass` para prototipar
- `pass` no hace nada, pero una función necesita un cuerpo
- Así, no incumpliremos las normas de sintaxis de Python cuando aún no tengamos muy claro qué hacer

```
def saludar():
    pass

saludar()
```

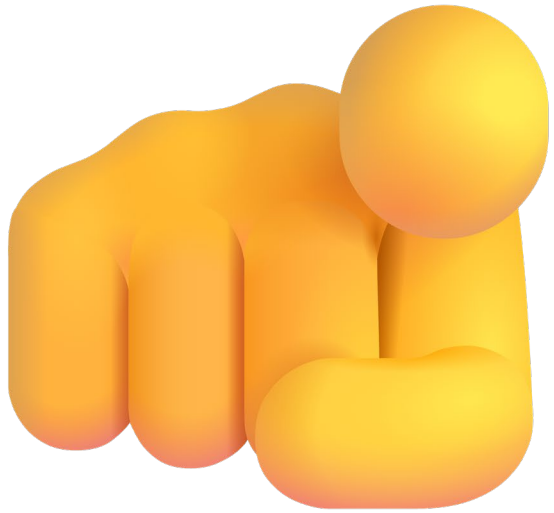
9.2. Parámetros

- Los parámetros se indican entre los paréntesis, separados por comas
- Dentro de la función, las podremos utilizar como variables

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

saludar("Hola", "Andy")
```

9.2. Parámetros

- ¿Se pasan por valor o por referencia?



9.2. Parámetros

Ni una cosa ni la otra: se pasan por referencia a objeto

```
def por_valor(x):
    x = x * 2
    print(f"número dentro: {x}")
    return

def por_referencia(l):
    l.append("D")
    print(f"lista dentro: {l}")
    return
```

```
lista = ["E"]
numero = 6
print(f"número antes: {numero}")
por_valor(numero)
print(f"número después: {numero}")
print(f"lista antes: {lista}")
por_referencia(lista)
print(f"lista después: {lista}")
```

9.2. Parámetros

- La posición importa...
- ...pero podemos usar los nombres

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

saludar(nombre="Andy", saludo="Hola")
```

9.2. Parámetros

- También podemos tener valores predeterminados para los parámetros
- Los que tengan valores predeterminados deben ir después de los que no los tengan

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

saludar(nombre="Andy")
saludar(saludo="Buenas", nombre="Andy")
```

9.2. Parámetros

Por si fuera poco, podemos tener funciones con un número indeterminado de parámetros posicionales

```
def sumar_todo(*numeros):
    return sum(numeros)

print(sumar_todo(1, 2, 3, 4, 5))
# Resultado: 15
```

9.2. Parámetros

- Por si fuera poco, podemos tener funciones con un número indeterminado de parámetros posicionales
- Esta manera de pasar los parámetros se conoce como `*args`

```
def sumar_todo(*numeros):
    return sum(numeros)

print(sumar_todo(1, 2, 3, 4, 5))
# Resultado: 15
```

9.2. Parámetros

- E incluso hacer lo mismo con parámetros con nombre, como si de un diccionario se tratase
- Esta manera de pasar los parámetros se conoce como **kwargs

```python
def mostrar_info(**parametros):
    for clave, valor in parametros.items():
        print(f"{clave}: {valor}")

mostrar_info(nombre="Carlos")

mostrar_info(
    nombre="Carlos",
    edad=30,
    ciudad="Madrid")
```

9.2. Parámetros

Es posible utilizar combinaciones de tipos de parámetros

Orden:

1. primero los posicionales
2. luego los con valor por defecto
3. después `*args`
4. finalmente `**kwargs`

9.3. Ámbito de las variables

- En principio no es posible acceder a las variables declaradas dentro de una función

```
def funcion():
    x = 10  # Variable local
    print(x)
funcion()
# print(x)
# ¡Error!
```

9.3. Ámbito de las variables

- Desde el interior de la función se puede acceder a variables definidas en el exterior, previamente a la declaración de la función
- ¡Pero no puede modificarla!

```
x = 5  # Variable global
def funcion():
    print(x)
funcion()
print(x)
```

9.3. Ámbito de las variables

A menos que la declaremos como global

```
x = 5  # Variable global
def funcion():
    global x
    x = 10
funcion()
print(x)
# Resultado: 10
```

9.3. Ámbito de las variables

También podemos re-declarar variables globales dentro de una función

```
x = 5
def funcion():
    x = 10
    print(x)
funcion()
print(x)
```

9.3. Ámbito de las variables

- ¡Evita todos estos líos!
- Trata en la medida de lo posible las variables globales como constantes
- Si necesitas una variable global, úsala, pero mantenlas al mínimo
- No la modifiques dentro de las funciones
- Si hay que cambiarla, pásala como parámetro, devuelve el valor, y que el responsable de esa variable la guarde

9.4. Retorno de valores

- Las funciones no tienen por qué devolver valores
- Si no lo hacen devuelven None de forma implícita

```
def saludar():
    print("Hola, Andy")

resultado = saludar()
if resultado is None:
    print("No devuelve nada")
```

9.4. Retorno de valores

- Pero si quieren devolver algo deben usar la palabra reservada `return`
- Además de devolver un valor, `return` termina la ejecución de la función: lo que haya a continuación no se ejecuta

```
def saludar():
    return "Hola, Andy"
    print("Esto no sirve")

resultado = saludar()
print(resultado)
```

9.4. Retorno de valores

- Una función puede devolver varios valores
- El resultado será una tupla
- Se puede asignar a una o varias variables

```python
def saludar():
    return "Hola", "Andy"

resultado = saludar()
print(f"{resultado[0]}, {resultado[1]}")
saludo, nombre = saludar()
print(f"{saludo}, {nombre}")
```

9.5. Anotaciones de tipos

- Python es un lenguaje interpretado de tipado dinámico
- No es necesario indicar el tipo de las variables
- Pero no que no sea necesario no quiere decir que no sea aconsejable

9.5. Anotaciones de tipos

- Python permite anotar el tipo de las variables, indicando su tipo deseado
- Para ello, dos puntos tras el nombre de la variable, y a continuación el indicador del tipo

```
nombre: str = "Andy"
edad: int = 50
datos: dict = {
    "nombre": nombre,
    "edad": edad
}
print(datos)
```

9.5. Anotaciones de tipos

- Sin embargo, nada nos obliga a seguir estas indicaciones
- Este programa se ejecutará correctamente, pero está claro que es incorrecto
- Pero... ¿por qué no me ayuda VSC?
- Porque no se lo hemos pedido

```
nombre: str = 50
edad: int = "Andy"
datos: dict = {
    "nombre": nombre,
    "edad": edad
}
print(datos)
```

9.5. Anotaciones de tipos

- Instalar extensión **Pylance**
- **Python > Analysis: Type Checking Mode basic** (estará en off)
- `python.analysis.typeCheckingMode": "basic"`
- Con eso activado, la cosa cambia
- Y si ponéis el puntero encima, se explica
- Sin embargo, el código sigue siendo ejecutable

```python
nombre: str = 50
edad: int = "Andy"
datos: dict = {
    "nombre": nombre,
    "edad": edad
}
print(datos)
```

9.5. Anotaciones de tipos

- Pero qué hacemos viendo esto en la sección dedicada a funciones, ¿por qué no lo hemos visto al hablar de las variables?
- Porque, aunque son útiles en la declaración de las propias variables, cuando de verdad lucen es al usarlas en las funciones

9.5. Anotaciones de tipos

- También se puede indicar el tipo de los datos que devuelve una función
- Permiten crear una especie de contrato

```
def saludar(saludo: str, nombre: str) ->
str:
    resultado = f"{saludo}, {nombre}"
    return resultado

saludo = saludar("Hola", "Andy")
# saludo += 1
# Aviso del IDE
```

9.5. Anotaciones de tipos

- Es posible utilizar tipos como listas, diccionarios, etcétera
- Para ello tenemos que importarlos de la biblioteca `typing`

```python
from typing import List, Dict

def procesar_lista(numeros: List[int]) -> List[int]:
    return [n * 2 for n in numeros]

def procesar_diccionario(d: Dict[str, int]) -> None:
    for clave, valor in d.items():
        print(f"{clave}: {valor}")

print(procesar_lista([1, 2, 3]))
diccionario = {
    "nombre": "Ana",
    "edad": 25,
    "ciudad": "Madrid"
}
print(procesar_diccionario(diccionario))
```

9.5. Anotaciones de tipos

- A veces una función puede volver un valor de un cierto tipo, pero si no se cumplen algunas condiciones devolver None
- El tipo del valor devuelvo, en este caso, se denomina opcional

```
from typing import Optional

def obtener_nombre(id: int) -> Optional[str]:
    if id == 1:
        return "Carlos"
    else:
        return None

resultado_1 = obtener_nombre(1)
resultado_2 = obtener_nombre(2)

print(resultado_1)
print(resultado_2)
```

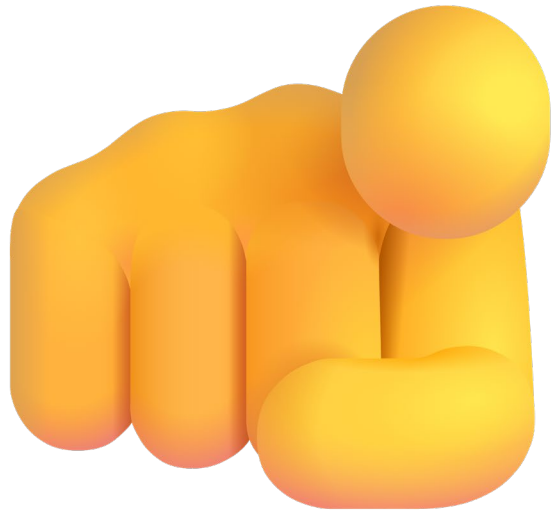
9.5. Anotaciones de tipos

- En otras ocasiones, el tipo puede ser múltiple
- En esos casos, se denomina union

```python
from typing import Union

def sumar_mixto(
    a: Union[int, float],
    b: Union[int, float]
) -> Union[int, float]:

    return a + b

resultado_1 = sumar_mixto(1, 2)
resultado_2 = sumar_mixto(1.0, 2.0)

print(resultado_1)
print(resultado_2)
```

9.5. Enumeraciones

¡Hagamos un ejercicio!



9.6. Documentación

- Ya hablamos sobre ella al principio de la unidad anterior
- Los comentarios permiten añadir líneas no ejecutables de código
- Imprescindibles para explicar el funcionamiento de nuestros programas...
  - ...a otras personas
  - ...¡y a nuestro futuro yo!
- Pero al documentar funciones tenemos que ser aún más cuidadosos

Fuente: Reddit

9.6. Documentación

- Las funciones permiten modularizar el código
- La mayoría no mirará en su interior para ver qué hace
- Acudirá a la documentación para aprender a utilizarlas
- ¿Cómo funciona *realmente* `print()`?
  - ¿Lo sabemos?
  - ¿Necesitamos saberlo?
  - No, lo que necesitamos es **saber utilizarla**
- Comentarios encima para contar qué hace la función
- Incluso comentarios en el cuerpo de la función para aclarar los puntos más complejos

```python
def factorial(n: int) -> int:
    resultado = 1
    for i in range(2, n + 1):
        resultado *= i

    return resultado

print(factorial(7))
# Resultado: 5040
```

9.6. Documentación

- Las funciones permiten modularizar el código
- La mayoría no mirará en su interior para ver qué hace
- Acudirá a la documentación para aprender a utilizarlas
- ¿Cómo función *realmente* `print()`?
  - ¿Lo sabemos?
  - ¿Necesitamos saberlo?
  - No, lo que necesitamos es **saber utilizarla**
- Comentarios encima para contar qué hace la función
- Incluso comentarios en el cuerpo de la función para aclarar los puntos más complejos

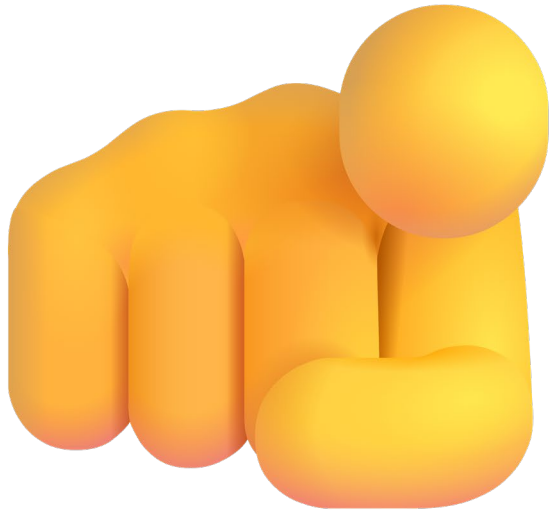
```python
# Calcula el factorial de un número
# Es el resultado de multiplicarlo
# por todos los anteriores, hasta el 1
def factorial(n: int) -> int:
    # Empezamos por el 1
    resultado = 1
    # Vamos desde el 2 hasta el número
    # Recuerda: range() para uno antes
    for i in range(2, n + 1):
        resultado *= i

    return resultado

print(factorial(7))
# Resultado: 5040
```

9.6. Documentación

- Podemos añadir comentarios justo antes para contar qué hace la función
- Podemos incluir comentarios en el cuerpo de la función para aclarar los puntos más complejos
- Pero habrá que ir a ver el código fuente
- O ir a la documentación en línea
- Debe existir un sistema más práctico...

9.6. Documentación

- ¡Y ya lo hemos usado!
- ¿Qué pasa cuando ponemos el puntero del ratón sobre una función que no sabemos cómo funciona?
- ¿Cómo nos ofrece los nombres de los parámetros?
- Gracias a una norma llamada *docstring* (presente también en otros lenguajes de programación)



9.6. Documentación

- docstring: cadena de texto justo después de la definición de la función
- Debe ir envuelta entre triples comillas (""" o ''')
- Ejemplo sencillo

```python
def factorial(n: int) -> int:
    """
    Calcula el factorial de un número
    Es el resultado de multiplicarlo
    por todos los anteriores, hasta el 1
    """

    resultado = 1
    # Vamos desde el 2 hasta el número
    # Recuerda: range() para uno antes
    for i in range(2, n + 1):
        resultado *= i

    return resultado
```

9.6. Documentación

Esta información se puede consultar posteriormente:

- dejando el puntero encima
- `print(función.__doc__)`
- `help(función)`

```
    Calcula el factorial de un número.
    Es el resultado de multiplicarlo
    por todos los anteriores, hasta el 1

Help on function factorial in module __main__:

factorial(n: int) -> int
    Calcula el factorial de un número.
    Es el resultado de multiplicarlo
    por todos los anteriores, hasta el 1
```

9.6. Documentación

Ejemplo más completo:

- `:param`: descripción de parámetros
- `:return`: descripción de la salida
- el IDE es el responsable de interpretarlos
- PEP 257: convenciones para docstring

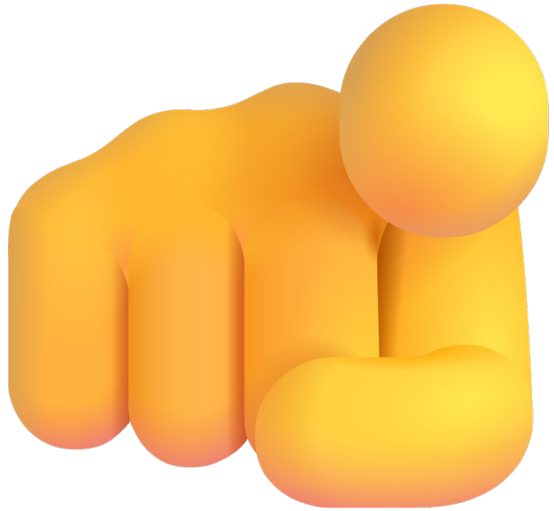
```python
def factorial(n: int) -> int:
    """
    Calcula el factorial de un número.
    Es el resultado de multiplicarlo
    por todos los anteriores, hasta el 1

    :param n: número con el que calcular
    el factorial

    :return: resultado del cálculo
    """

    resultado = 1
    # ...
```

9.6. Documentación

- Que tu documentación sea clara y concisa
- Describe el propósito de la función en la primera línea
- Usa un estilo de documentación que sea consistente en todo el código
- Opcionalmente, describe los parámetros y el valor devuelto en secciones separadas
- ¡Actualiza la documentación si la función cambia!

9.6. Documentación

- ¡Hagamos un ejercicio!



9.7. Como objetos de primera categoría

Las funciones son objetos de primera categoría. Eso quiere decir que se pueden:

- Asignar a variables
- Pasar como parámetro
- Devolver como resultado desde una función
- Almacenar en estructuras de datos como listas o diccionarios

9.7. Como objetos de primera categoría

- Flexibilidad para diseñar funciones reutilizables y combinarlas de manera creativa

- Facilita la programación funcional en Python, permitiendo trabajar con funciones como datos

9.7. Como objetos de primera categoría

Asignar a variables

```
def saludar():
    print("Hola")

# Asignar la función a
una variable
saludo = saludar
saludo()
# Resultado: "Hola"
```

9.7. Como objetos de primera categoría

Pasar como parámetro

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

def cuadrado(x):
    return x**2

print(aplicar_funcion(cuadrado, 5))
# Resultado: 25
```

9.7. Como objetos de primera categoría

Devolver desde una función

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

multiplicador = generar_multiplicador(4)
print(multiplicador(10))
# Resultado: 40
```

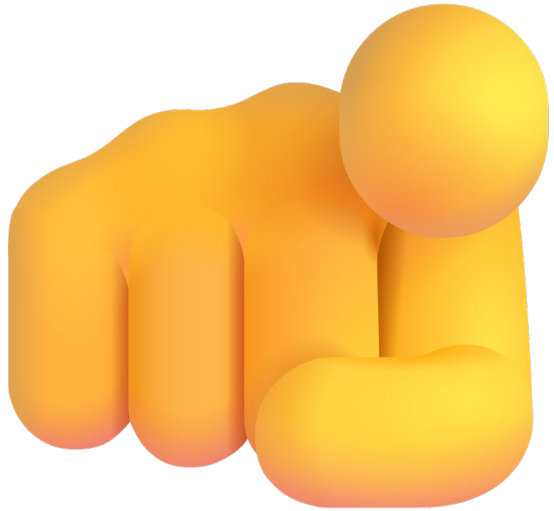
9.7. Como objetos de primera categoría

Guardar en estructuras de datos

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

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

operaciones = {
    "suma": sumar,
    "resta": restar
}

print(operaciones["suma"](5, 3))
# Resultado: 8
print(operaciones["resta"](5, 3))
# Resultado: 2
```

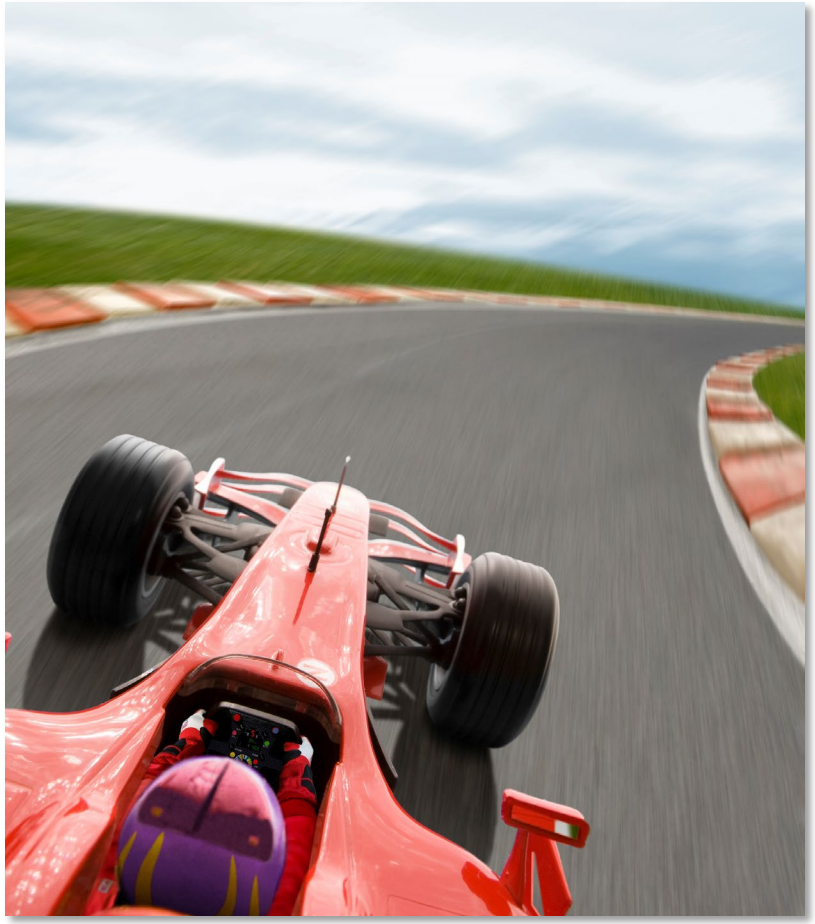
9.7. Como objetos de primera categoría

- ¡Hagamos un ejercicio!



9.8. Funciones anónimas

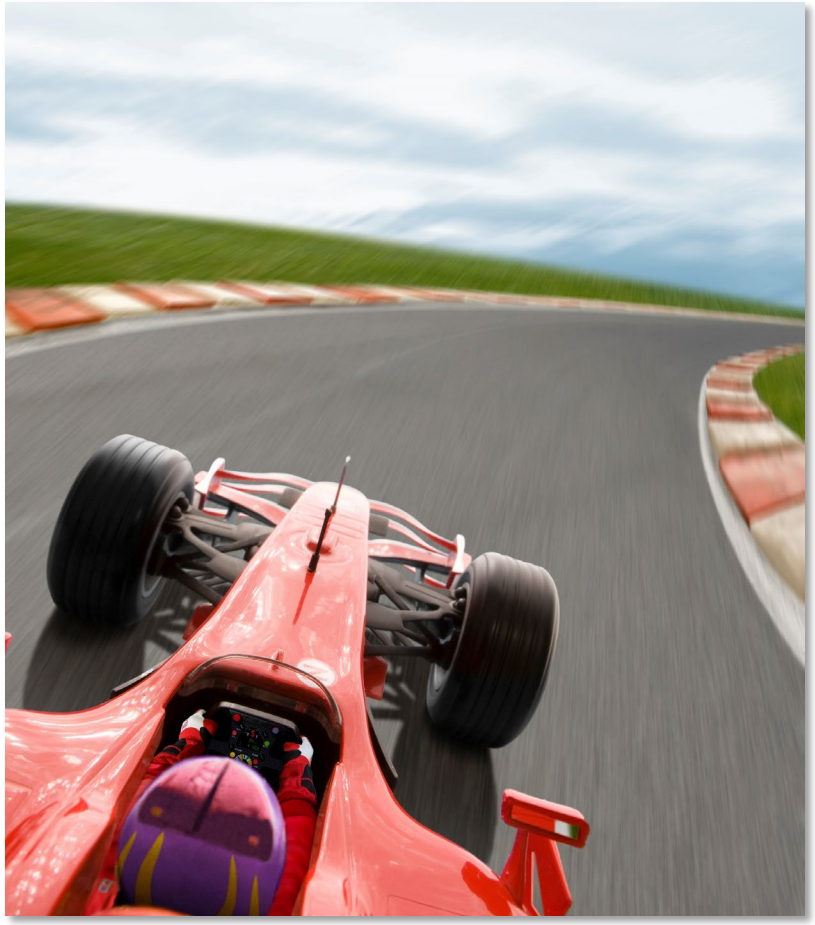
- No se declaran formalmente con `def`
- Se declaran en el momento, usando la palabra reservada `lambda`
- Pensadas para tareas simples y rápidas, generalmente en una sola línea



9.8. Funciones anónimas

Sintaxis:

```
lambda parámetros: expresión
```



9.8. Funciones anónimas

Ejemplo sin lambda

```
lista = [(2, 'b'), (3, 'a'), (1, 'c')]

def ordenar_por_primero(tupla):
    return tupla[0]


lista_ordenada = sorted(
    lista,
    key=ordenar_por_primero)

print(lista_ordenada)
# [(1, 'c'), (2, 'b'), (3, 'a')]
```

9.8. Funciones anónimas

Ejemplo con lambda

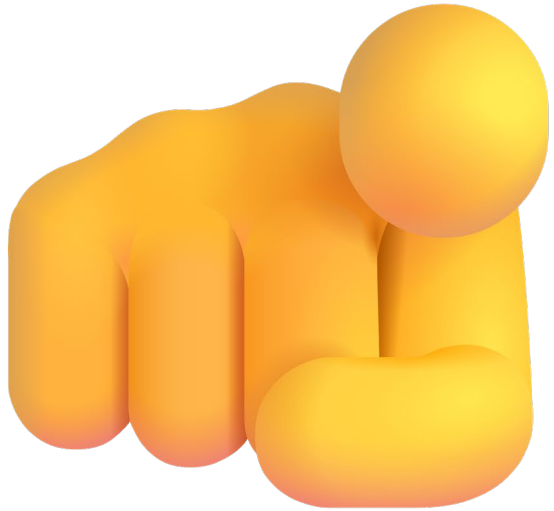
```
lista = [(2, 'b'), (3, 'a'), (1, 'c')]

lista_ordenada_lambda = sorted(
    lista,
    key=lambda tupla: tupla[0])

print(lista_ordenada_lambda)
# [(1, 'c'), (2, 'b'), (3, 'a')]
```

9.8. Funciones anónimas

- Las funciones anónimas están limitadas a una sola expresión, no pueden contener múltiples líneas ni estructuras complejas
- Si el cuerpo de la función es complejo o largo, es mejor usar una función normal con `def`
- Aunque sean útiles, no debe usarse para todo: debe primar la legibilidad y el mantenimiento del código frente a "ser guay"

9.8. Funciones anónimas

- ¡Hagamos un ejercicio!



10. Programación orientada a objetos

1. Introducción
2. Clases y objetos
3. Métodos estáticos
4. Herencia
5. Clases de datos
6. Métodos especiales

10.1. Introducción

- Ya sabéis qué es, ¿verdad?
- Modelado del mundo real
- Reusabilidad de código
- Encapsulación
- Abstracción
- Herencia
- Polimorfismo

10.2. Clases y objetos

- Llevamos usando clases ya un buen rato, en realidad
- Por ejemplo, existe una clase cadena (`str`)
- `saludo = "Hola"`
- `saludo` es un ejemplar de la clase cadena
- `cadena.split()` es un método de ese objeto

10.2. Clases y objetos

- Obviamente, podemos crear nuestras propias clases
- Ya lo hicimos con las enumeraciones
- Palabra reservada `class`
- Un inicializador
- Un método
- `self` hace referencia al ejemplar de la clase

```python
class Saludo:
    def __init__(self, saludo, nombre):
        self.saludo = saludo
        self.nombre = nombre

    def saludar(self):
        print(f"{self.saludo}, {self.nombre}!")

# Ejemplo de uso
saludo = Saludo("Hola", "Juan")
saludo.saludar()
# Hola, Juan!
```

10.3. Métodos estáticos

- Podemos crear métodos que no accedan a los atributos de la clase
- No es necesario crear un ejemplar de la clase para usarlo
- Decoradores: modifican el comportamiento
- `@staticmethod` es un decorador

```
    ...
    @staticmethod
    def saludo_clasico():
        print("Hola, Mundo!")

# Ejemplo de uso
saludo = Saludo("Hola", "Juan")
saludo.saludar()
Saludo.saludo_clasico()
# Hola, Juan!
```

10.4. Herencia

- La hemos utilizado al combinar las enumeraciones, ¿os acordáis?
- La enumeración hereda de Enum
- Podemos crear nuestras propias clases y jerarquías de herencia

```
from enum import Enum

class DiaSemana(Enum):
    Lunes = 1
    Martes = 2
    Miercoles = 3
    Jueves = 4
    Viernes = 5

dia = DiaSemana.Lunes

print(dia)
print(dia.name)
print(dia.value)
```

10.4. Herencia

```python
# Clase base Vehiculo
class Vehiculo:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    def acelerar(self):
        print(f"El {self.marca} {self.modelo} "
              f"está acelerando.")

    def frenar(self):
        print(f"El {self.marca} {self.modelo} "
              f"está frenando.")
```

```python
# Clase derivada Coche
class Coche(Vehiculo):
    def __init__(self, marca, modelo, numero_puertas):
        super().__init__(marca, modelo)
        self.numero_puertas = numero_puertas

    def abrir_puertas(self):
        print(f"El {self.marca} {self.modelo} "
              f"con {self.numero_puertas} puertas "
              f"está abriendo las puertas.")

mi_coche = Coche("Simca", "1000", 4)
mi_coche.acelerar()
mi_coche.frenar()
mi_coche.abrir_puertas()
```

10.5. Clases de datos

- A veces nos bastan clases que sólo tengan atributos
- Se conocen como clases de datos
- Son más simples
- Inicializador implícito

```
from dataclasses import
dataclass

@dataclass
class Punto:
    x: int
    y: int

punto = Punto(10, 20)
print(punto.x, punto.y)
# 10 20
```

10.6. Métodos especiales

- Permiten modificar el comportamiento de las clases
- `__str__`: ofrece una representación legible de la clase. Útil para usar con `print()`
- `__repr__`: similar, pero para usar con `repr()`. Útil para desarrolladores y depuración

```python
from dataclasses import dataclass

@dataclass
class Punto:
    x: int
    y: int

    def __str__(self):
        return f"{self.x}, {self.y}"

    def __repr__(self):
        return f"Punto(x={self.x}, y={self.y})"

punto = Punto(10, 20)
print(punto.x, punto.y)
print(punto)
print(repr(punto))
```

10.6. Métodos especiales

- Iteradores para usarlos con el bucle `for`
- Métodos especiales `__iter__()` y `__next__()`
- Python los buscará para saber si la clase funciona o no cómo iterador

```
class Contador:
    def __init__(self, limite):
        self.limite = limite
        self.actual = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.actual < self.limite:
            self.actual += 1
            return self.actual
        else:
            raise StopIteration

contador = Contador(5)
for num in contador:
    print(num)
```

10.6. Métodos especiales

- Sobrecarga de operadores
- Para poder sumar dos ejemplares de una clase, por ejemplo
- Existen otros como `__sub__()`, `__mul__()`, etcétera

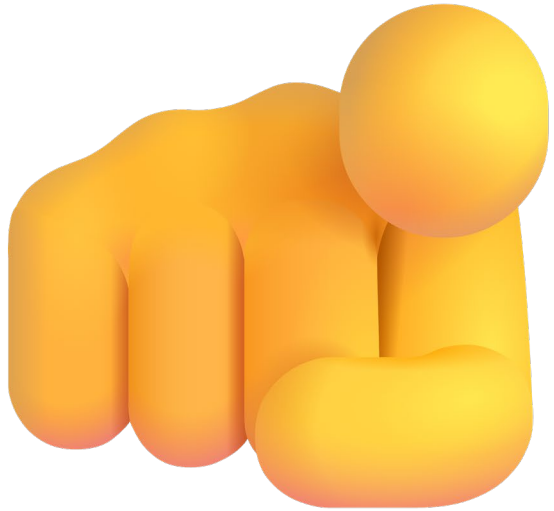
```python
class Punto:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, otro):
        return Punto(
            self.x + otro.x,
            self.y + otro.y)

    def __str__(self):
        return f"Punto({self.x}, {self.y})"

punto_1 = Punto(2, 3)
punto_2 = Punto(4, 1)

punto = punto_1 + punto_2

print(punto)
```

10. Programación orientada a objetos

Podríamos contar mucho más sobre la programación orientada a objetos, pero con estos conceptos básicos tenemos lo necesario para avanzar

10. Programación orientada a objetos

- ¡Hagamos un ejercicio!



11. Archivos

1. Apertura
2. Escritura
3. Lectura
4. Cierre
5. Uso seguro con `with`

11.1. Apertura

Utiliza la función open ( ) para abrir un archivo. Puedes especificar el modo de apertura:

- "r" para lectura
- "w" para escritura (sobrescribe el archivo si ya existe)
- "a" para añadir contenido sin sobrescribir
- "r+" para leer y escribir

```
archivo = open(
  "archivo.txt", "w")
# Crea o abre el archivo
para escribir
```

11.2. Escritura

- Usa el método `write()` para añadir contenido

```
archivo.write(
  "Hola, mundo!")
```

11.3. Lectura

- Lee el contenido de un archivo usando `read()`, `readline()` o `readlines()`

```
contenido =
  archivo.read()
print(contenido)
```

11.4. Cierre

- Tras de trabajar con un archivo, es importante cerrarlo con `close()` para liberar recursos

```
archivo.close()
```

11.5. Uso seguro con `with`

- La palabra reservada `with` facilita el trabajo con los archivos
- Al abandonar el bloque, el archivo se cierra automáticamente

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

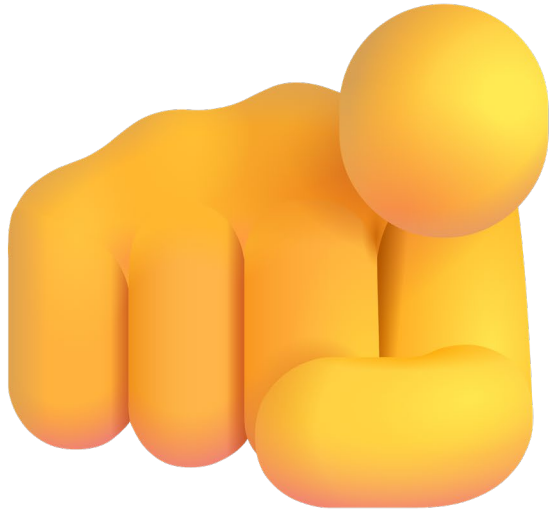
11. Ejemplo completo

- Creamos el archivo para escritura
- Añadimos dos líneas
- Lo guardamos y cerramos de forma implícita
- Lo abrimos para leerlo
- Se cierra de forma implícita
- Mostramos el contenido

```
# Crear y escribir en un archivo
with open("archivo.txt", "w") as archivo:
    archivo.write("Este es un archivo "
                  "de ejemplo.\n")
    archivo.write("Está escrito con la "
                  "función write en Python.\n")

# Leer el archivo recién creado
with open("archivo.txt", "r") as archivo:
    contenido = archivo.read()

# Mostrar el contenido leído
print(contenido)
```

11. Archivos

- ¡Hagamos un ejercicio!

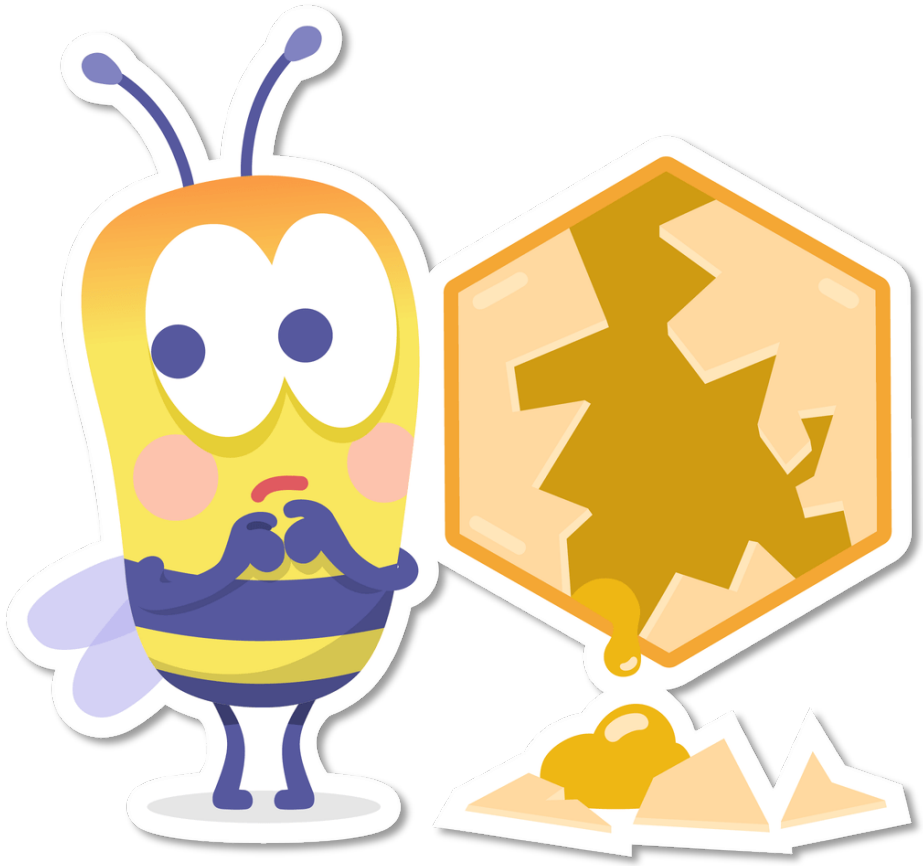


11. Archivos

- Existen múltiples interfaces de acceso a archivos en Python, tanto nativas como de otros desarrolladores: os, shutil, csv, json, yaml, pickle, etcétera
- Por ejemplo, pandas, que veremos en otra sesión, tiene su propia interfaz de acceso a archivos, tanto para lectura como para escritura
- Lo importante es encontrar la que más nos convenga

12. Gestión de errores

1. try-except
2. raise



12.1. `try-except`

- Python ofrece un mecanismo de manejo de errores a través de bloques `try-except`
- Permite capturar y manejar excepciones
- Evita que el programa se detenga abruptamente

```
try:
    file = open("archivo.txt", "r")
except FileNotFoundError:
    print("El archivo no existe.")
```

12.1. `try-except`

- Pueden contener varias instrucciones
- Se pueden capturar diferentes tipos de excepciones con bloques `except` separados
- También se puede usar `else` para ejecutar código si no se produce ninguna excepción
- Y `finally` para ejecutar código siempre, independientemente de si hubo o no una excepción

```
try:
    # Intentar abrir un archivo
    file = open("archivo_inexistente.txt", "r")
except FileNotFoundError:
    print("Error: el archivo no existe.")
except PermissionError:
    print("Error: no tienes permisos")
else:
    print("Archivo abierto correctamente.")
    file.close()
finally:
    print("Este bloque siempre se ejecuta.")
```

12.1. `try-except`

- Se pueden capturar excepciones genéricas si no sabemos su tipo
- Y podemos mostrar detalles sobre la misma

```
try:
    file = open("archivo_inexistente.txt", "r")
except Exception as e:
    print(f"Se produjo un error: {e}")
```

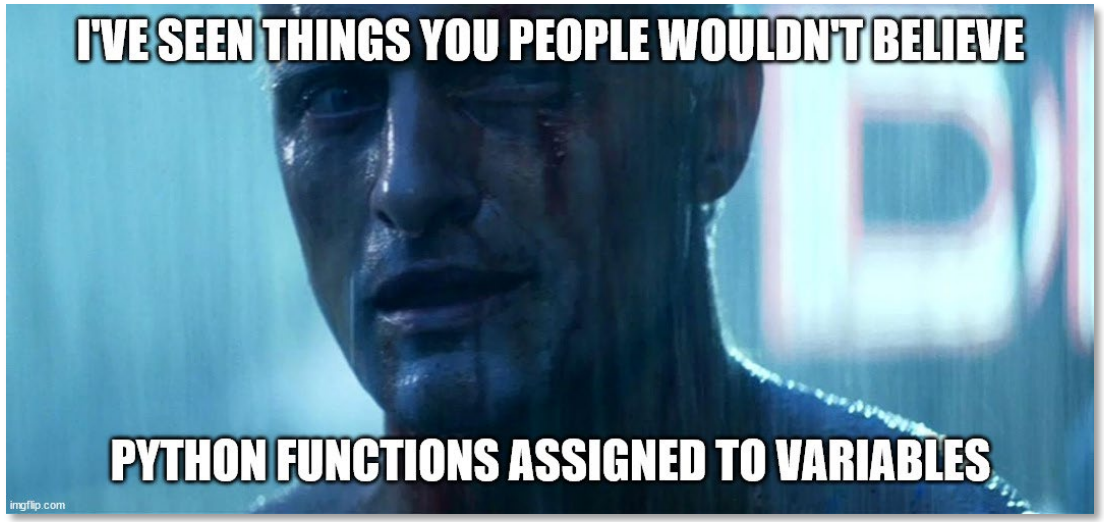
12.2. `raise`

- Puedes lanzar tus propias excepciones utilizando `raise`

```
raise NotImplementedError
raise ValueError("Error personalizado")
```

Conclusiones

- Pues sí que hemos visto cosas...
  - Estructuras de datos
  - Funciones
  - Más funciones
  - Clases y objetos
  - Archivos
  - Gestión de errores



Fuente: Know Your Meme
Fuente: Meme Generator

Conclusiones

- Con estos conocimientos, estamos en condiciones para afrontar lo que nos queda por delante
- Ya tenéis lo necesario para adentraros en el maravillo mundo de Python, para entenderlo casi todo
- Ahora empieza lo mejor

Recursos y referencias

- Python Tutorial
- PEP 257: Docstring Conventions
- Contra la programación orientada a objetos
- Operator and Function Overloading in Custom Python Classes

3. Conceptos básicos (2/2)

Introducción a Python para Deep Learning