PYTHON BASICO

Como vamos a trabajar

Vamos a utilizar un entorno de trabajo colaborativo para que tengáis acceso a todo, mis códigos y que también tengáis vuestros códigos.

Lo que haremos será trabajar con **Github**.

1. Instalar Python

Necesitamos un IDE para trabajar. El IDE es **Visual Studio Code** que sirve para programar con Python

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.

Debemos separa los conceptos:

1. Jupyter es un cuaderno de trabajo Python. Sirve para análisis de datos
2. Python básico se mueve en cualquier script.

Todo esto está reunido dentro de cualquier entorno.

Para trabajar con Jupyter, necesitamos un entorno.

Para trabajar con Script, necesitamos un entorno.

Para trabajar con Fast Api o Django, necesitamos un entorno.

La diferencia es que mediante Visual Studio Code, podemos utilizar TODO.

Mediante extensiones, podemos hacer que nuestro entorno funcione sin problemas dentro de VS Code.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Aplicación, Word

El contenido generado por IA puede ser incorrecto.

Una vez que tenemos esto, ya se ha ejecutado un elemento para poder crear aplicaciones de Python

Todas las extensiones de los ficheros Python (mentira) terminan en **.py**

Simplemente, cualquier fichero que tengamos se puede ejecutar mediante VS Code sin necesidad de nada más.

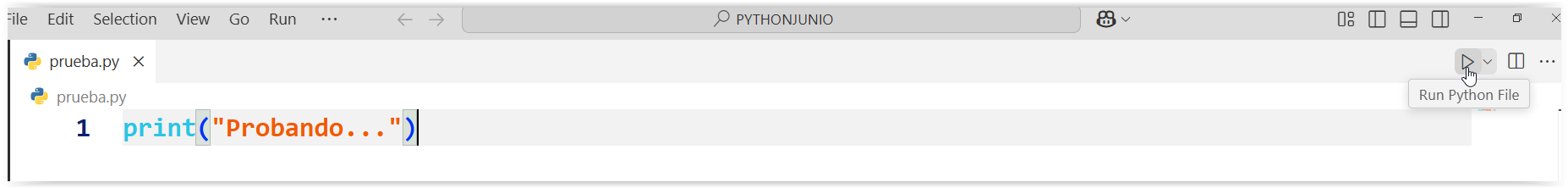
Para trabajar, simplemente tendremos una carpeta con nuestros ficheros Python.

Dependiendo del proyecto/framework se generan más carpetas. Pero en este tipo básico, simplemente necesitamos una carpeta en nuestro ordenador y dentro iremos creando todas las clases y características

PYTHONJUNIO

Interfaz de usuario gráfica, Aplicación, Tabla

El contenido generado por IA puede ser incorrecto.



Python es un lenguaje de programación. Contiene programación orientada a objetos y se comunica con múltiples elementos, por ejemplo, Web, IoT, Api, Análisis de datos, Big Data…

**Importante:**

* **Diferencia mayúsculas de minúsculas**
* **Las tabulaciones son básicas, delimitan el código del programa**
* **Las variables (mentira) no tienen tipado, simplemente le asignamos un valor.**

Las variables se declaran en el código

numero = 14

texto = “Soy un texto”

**print** en nuestro entorno será el que nos servirá para visualizar que sucede. Dependiendo del Framework que utilicemos, no se utiliza print

Dependiendo del tipo de variable, podremos sumar o restar o dividir o podremos buscar una letra o convertir a mayúsculas

Para trabajar y ser ordenados, iremos nombrando a nuestros ficheros de una forma determinada. En principio no importa el nombre del fichero, pero normalmente, en Python se declaran los programas en minúsculas.

Creamos un nuevo programa llamado **python01variables.py**

**print("Ejemplo de variables")**

**numero = 19**

**texto = "Hola mundo"**

**#Comentarios en Python**

**# Para comentar y descomentar código de una sola vez en VS Code.ç**

**# Comentar:  CONTROL + K + C**

**# Descomentar: CONTROL + K + U**

**print(numero)**

**print(texto)**

**# Tenemos la posibilidad de concatenar +: Verifica el tipado**

**# print("numero " + numero) # error**

**# Con coma no verifica tipado**

**print("numero ", numero)**

**print("texto " + texto)**

**#print f nos permite concatenar múltiples variables en un String sin**

**#importar el tipado: Cada variable irá entre llaves {variable}**

**print(*f*"El texto es {texto} y el numero es {numero}")**

**#Dentro de las variables tenemos funciones de conversión lo que permiten convertir**

**#entre un tipo y otro, por ejemplo:**

**#str(variable): Convierte un valor a string**

**#float(variable): Convierte variable a tipo decimal**

**#int(variable): Convierte variable a entero**

**print("numero " + *str*(numero))**

CONDICIONALES

Un condicional dentro de cualquier lenguaje implica que un programa no será lineal.

Para poder utilizar condicionales/preguntas en nuestro programa se utilizan las TABULACIONES para delimitar si la pregunta es correcta o no es correcta.

Al hablar de condiciones, estamos hablando de IF

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Para poder comparar, necesitamos la sintaxis de los operadores de comparación de Python.

> Mayor

< Menor

>= Mayor o igual

<= Menor o igual

== Igual

!= Distinto

Vamos a realizar un simple programa en el que indicaremos si un número es positivo o negativo o cero.

Pediremos el número al usuario mediante una función de Python llamada **input()**

Creamos un nuevo programa llamado **python02condicionales.py**

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

ELSE

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

ELSE IF ANIDADO

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Si tiene que ver con la misma pregunta, lo que podemos utilizar es la sentencia **elif**

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

**DEPURACION DE CODIGO**

La depuración de código nos permite visualizar las variables dinámicamente en nuestro programa o nos permite averiguar que ha sucedido.

En cualquier programa tenemos tres tipos de errores:

1. **Errores de compilación**: Algo he escrito mal y el programa no funciona.
2. **Errores en ejecución**:El programa compila y ejecuta, pero se me ha escapado algún detalle de comprobación. Por ejemplo, en este ejemplo anterior, comprobar que ha introducido números.
3. **Errores lógicos**: Simplemente mi programa funciona, no da ningún error, pero no hace lo que quiero. Debemos averiguar qué sucede

Para depurar en Python con VS Code, simplemente debemos agregar un Punto de interrupción dentro de nuestro programa en la parte de la izquierda de los números.

Texto

El contenido generado por IA puede ser incorrecto.

Para ejecutar el programa en modo depuración, tenemos un icono de un bicho a la izquierda de nuestro código llamado Run and Debug

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Cuando hablamos de condicionales, tenemos también los operadores relacionales:

Son dos palabras clave:

* **and**: Todas las condiciones deben cumplirse
* **or**: Cualquier condición entrará en el IF

Para visualizar esto y tener un ejemplo de sintaxis:

Creamos una nueva App llamada **python03mayormenor.py**

Vamos a pedir tres números y mostramos el mayor, menor e intermedio de ellos.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

**print("Operadores relacionales")**

**num1 = *int*(input("Número 1"))**

**num2 = *int*(input("Número 2"))**

**num3 = *int*(input("Número 3"))**

**mayor = 0**

**menor = 0**

**intermedio = 0**

**if (num1 >= num2 and num1 >= num3):**

**mayor = num1**

**elif (num2 >= num1 and num2 >= num3):**

**mayor = num2**

**else:**

**mayor = num3**

**if (num1 <= num2 and num1 <= num3):**

**menor = num1**

**elif (num2 <= num1 and num2 <= num3):**

**menor = num2**

**else:**

**menor = num3**

**suma = num1 + num2 + num3**

**intermedio = suma - mayor - menor**

**print(*f*"Mayor: {mayor}, Menor: {menor}, Intermedio: {intermedio}")**

**print("Fin de programa")**

BUCLES

Un bucle es una secuencia iterativa que se repite N veces en el código.

Tenemos dos tipos de bucles:

1. **while:** Bucle condicional que debemos entrar y salir con una condición. Dentro del bucle debemos cambiar “algo” para que se cumpla la condición y el bucle finalice.

Sintaxis:

while (condición == true):

#instrucciones

condición = false

print(“Fin de programa”)

1. **for:** Bucle que nos permite repetir una secuencia N veces. La secuencia se realiza mediante un contador declarado dentro de la sintaxis del propio bucle.

Entra y sale solo, no hace bucles infinitos.

Un bucle contador for siempre comienza en CERO

Sintaxis:

for **contador** in range(final):

#instrucciones

Tenemos otro tipo de sintaxis en el que podemos indicar el inicio y el final de contador

for **contador** in range(**inicio, final**):

#instrucciones

Creamos un nuevo programa llamado **python04bucles.py**

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Vamos a realizar un ejemplo con la Conjetura de Collatz.

La conjetura indica lo siguiente:

Todo número entero siempre llegará a ser 1 si se cumplen las siguientes condiciones:

* Si el número es par, se divide entre 2.
* Si el número es impar, se multiplica por 3 y se suma 1

6, 3, 10, 5, 16, 8, 4, 2, 1

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

**print("Ejemplos y sintaxis de bucles")**

**#REALIZAMOS EL TIPICO DE MOSTRAR LOS NUMEROS PARES ENTRE UN RANGO**

**for i in range(2, 20):**

**#PREGUNTAMOS SI UN NUMERO ES PAR**

**#UTILIZAMOS EL OPERADOR DEL RESTO: %**

**if (i % 2 == 0):**

**print(*f*"Par: {i}")**

**print("----------Collatz----------------")**

**numero = *int*(input("Introduzca numero: "))**

**while (numero != 1):**

**if (numero % 2 == 0):**

**numero = *int*(numero / 2)**

**else:**

**numero = numero \* 3 + 1**

**print(numero)**

**print("Fin de programa")**

**CLASE STRING PYTHON**

Los objetos, al ser declarados con algún tipado (todavía explícitamente no), contienen una serie de características o métodos.

En Python existen clases y, dependiendo de dichas clases, tendremos una serie de métodos que nos pueden ayudar en programación.

Por ejemplo, tenemos clases que ya vienen con nuestro compilador y no necesitamos nada más: String

Por otro lado, tendremos clases que habrá que importar y utilizarlas en el programa, por ejemplo, la clase Math.

Por último, tenemos clases que no existen en el compilador y tendremos que agregarlas a nuestro programa utilizando herramientas externas.

Tenemos una función para saber la longitud de cualquier objeto conjunto: **len()**

Un String diferencia mayúsculas de minúsculas.

texto = “Lenguaje Python”

len(texto) 🡪 15

Dentro de este conjunto, también tenemos la posibilidad de acceder a cada elemento individual (letra) mediante un índice: Objeto[índice]

texto[0] 🡪 L

texto[1] 🡪 e

Tenemos una serie de métodos para trabajar con estos objetos string:

* lower(): Convierte a minúscula un texto
* upper(): Convierte a mayúscula un texto
* find(“Contenido a buscar”): Busca el contenido dentro del texto y nos devuelve su posición. Si no lo encuentra, nos devuelve -1
* rfind(“Contenido a buscar”): Busca el contenido dentro del texto y nos devuelve su posición. Si no encuentra, nos devuelve -1. Comienza a buscar desde el final

Dentro de Python tenemos POO y eso nos permite tener sobrecarga en algunos métodos/funciones

* find(“Contenido a buscar”, índice): Busca el contenido dentro del texto y nos devuelve su posición comenzando a buscar desde el índice. Si no lo encuentra, nos devuelve -1
* startswith(“contenido”): Indica si el texto comienza con el contenido
* endswith(“contenido”): Indica si el texto finaliza con el contenido
* replace(“antiguo”, “nuevo”): Sustituye el contenido del texto antiguo por el nuevo
* count(“texto”): Nos devuelve el número de veces que aparece el texto

Tenemos métodos para preguntar por el tipo de contenido del string:

* isdigit(): Indica si el contenido del texto son números
* isalpha(): Indica si el contenido del texto son letras
* isalnum(): Indica si el contenido está formado por letras y números
* isupper(): Indica si un texto está en mayúsculas
* islower(): Indica si el texto está en minúscula

Dentro de Python, tenemos una funcionalidad que se llama **SLICING** que nos permite poder extraer subcadenas de un conjunto

texto = “primer python”

Tenemos dos posibilidades para obtener una subcadena de un texto a partir de sus índices:

1. Queremos la subcadena desde una posición concreta en adelante: Objeto[índice:]

texto[8:] 🡪 ython

1. Devuelve una subcadena desde una posición inicial hasta una posición final

texto[0:2] 🡪 pri

Vamos a realizar una aplicación para validar un mail.

* Que el email contenga @
* Que el email contenga .
* @ ni al inicio ni al final
* Solamente una @ en el email
* Que exista un punto después de la @
* Dominio de 2 a 3 caracteres

Creamos un nuevo programa llamado **python05validaremail.py**

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.

Paco Garcia Serrano

**print("Validacion email string")**

**email = input("Introduzca email: ")**

**# • Que el email contenga @**

**# • Que el email contenga .**

**# • @ ni al inicio ni al final**

**# • Solamente una @ en el email**

**# • Que exista un punto después de la @**

**# • Dominio de 2 a 3 caracteres**

**if (email.find("@") == -1):**

**print("No existe @")**

**elif (email.count(".") == 0):**

**print("Email sin punto")**

**elif (email.startswith("@") == True or email.endswith("@") == True):**

**print("@ al inicio o al final")**

**elif (email.count("@") > 1):**

**print("Tenemos más de una @")**

**elif (email.find("@") > email.rfind(".")):**

**print("Debe existir un punto después de @")**

**else:**

**#posicion del ultimo punto**

**ultimoPunto = email.rfind(".")**

**#Recuperamos el dominio a partir de dicho punto en adelante**

**#no queremos el punto incluido**

**dominio = email[ultimoPunto + 1:]**

**longitud = len(dominio)**

**if (longitud >=2 and longitud <= 3):**

**print(*f*"Email correcto: {email}")**

**else:**

**print("Dominio de dos a tres caracteres")**

**print("Fin de programa")**

Vamos a realizar un ejemplo en el que recorreremos todos los caracteres de un texto (numérico) y mostraremos la suma de cada carácter.

Imagen que contiene Interfaz de usuario gráfica

El contenido generado por IA puede ser incorrecto.

Creamos un programa nuevo llamado **python06numerosstring.py**

**print("Sumar números de textos")**

**textoNumeros = input("Introduzca un texto numerico: ")**

**suma = 0**

**# Realizamos un bucle para recorrer cada letra del texto**

**for i in range(len(textoNumeros)):**

**#ALMACENAMOS CADA LETRA DE CADA POSICION**

**letra = textoNumeros[i]**

**# EXTRAEMOS CADA LETRA Y LA CONVERTIMOS A int**

**numero = *int*(letra)**

**suma = suma + numero**

**print(*f*"La suma de {textoNumeros} es: {suma}")**

**print("Fin de programa")**

## METODOS/FUNCIONES DENTRO DEL LENGUAJE PYTHON

Una función es un método, en realidad son lo mismo.

Son herramientas que contiene el lenguaje de Python y que pueden venir dentro de una clase o no.

* Un método en una clase: La clase String tiene un método que permite buscar si una cadena comienza con un determinado Carácter/Texto: startswith()
* Método sin clase: Dichos métodos están “sueltos” y no dependen de una clase determinada. Por ejemplo: **len(texto)**

Un programa de Python, como hicimos el otro día, UTILIZA métodos y funciones de una clase o de un Programa

Estos métodos/funciones nos permiten realizar códigos NO lineales, es decir, separar ciertas características/funcionalidades del programa en partes o zonas.

Un método nos permite también poder reutilizar nuestro código, lo escribimos una vez y lo utilizamos múltiples veces.

Los métodos pueden estar en el mismo programa principal o separado en diferentes ficheros de Python.

Todo funciona mediante llamadas y declaraciones de métodos, por ejemplo, imaginemos que tenemos por ahí un método que nos valida el Email:

PROGRAM MAIN.PY

print(“Programa principal”)

print(“Introduzca un email”)

email = input()

#AQUI TODO EL CODIGO DE VALIDACION

**validarEmail()**

**metodos.validarEmail()**

#METODO DE VALIDAR UN MAIL

**validarEmail()**

{

#NUESTRO CODIGO DE VALIDACION

}

Podemos plantear nuestro programa de dos formas mediante métodos:

1. En el mismo Main donde estamos, crear un método para validar un Email
2. Podríamos tener otro fichero con métodos

**METODOS.PY**

#METODO DE VALIDAR UN MAIL

**validarEmail()**

{

#NUESTRO CODIGO DE VALIDACION

}

Para crear métodos en Python se utiliza una palabra clave llamada **def** y, a continuación, el nombre del método

La sintaxis de los nombres de método es como en Java/JavaScript, es decir, la primera palabra en minúscula y cada palabra siguiente con la inicial en mayúscula: **nombreMetodo()**

En Python, de forma obligada, los métodos deben estar al inicio de la clase.

Por ahora, los métodos los veremos en la misma clase **main.py**.

Creamos un nuevo programa llamado **python07metodos.py**

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.

**# EN UN PROGRAMA LOS METODOS DEBEN**

**# ESTAR ANTES DE LA LLAMADA, POR LO QUE**

**# SE SUELEN DECLARAR AL INICIO**

***def* primerMetodo():**

**#ESTE CODIGO JAMAS SE EJECUTARA**

**# SIN SER LLAMADO DE FORMA EXPLICITA**

**print("Ejecutando metodo 1")**

***def* segundoMetodo():**

**print("Ejecutando metodo 2")**

**#-------------------------------------------------------------**

**#CODIGO PRINCIPAL DE NUESTRO PROGRAMA MAIN**

**print("Ejemplo métodos")**

**#LLAMADA LOS METODOS**

**primerMetodo()**

**segundoMetodo()**

**primerMetodo()**

**print("Fin de programa")**

## PARAMETROS EN LOS METODOS

Un método puede recibir parámetros, dichos parámetros NO son opcionales (por ahora), deben ser enviados al método para poder ejecutarlo en la llamada.

Los tipos de datos en los métodos no se definen (por ahora), simplemente se escribe el nombre de la variable que voy a utilizar dentro del método.

Dicha variable tendrá ámbito de método, es decir, solamente la podemos utilizar dentro del **def**

Un método puede contener múltiples parámetros, simplemente se separan mediante coma dichos parámetros.

El orden de la llamada de dichos métodos con parámetros será el orden de los parámetros

Ejemplo:

def metodoParametros(**nombre**):

print(“Su nombre es: “ + nombre)

Petición al método

metodoParametros(12)

metodoParametros(“Paco”)

def metodoParametrosMultiple(**nombre, apellidos, edad**):

print(“Su nombre es: “ + nombre + apellidos)

metodoParametrosMultiple(“Paco”, “Garcia”, 44)

Vamos a probar esto en el mismo programa que tenemos de métodos.

**#Comenta: control + k + c**

**#Descomentar: control + k + u**

**# EN UN PROGRAMA LOS METODOS DEBEN**

**# ESTAR ANTES DE LA LLAMADA, POR LO QUE**

**# SE SUELEN DECLARAR AL INICIO**

***def* primerMetodo():**

**#ESTE CODIGO JAMAS SE EJECUTARA**

**# SIN SER LLAMADO DE FORMA EXPLICITA**

**print("Ejecutando metodo 1")**

***def* segundoMetodo():**

**print("Ejecutando metodo 2")**

***def* saludar(*nombre*):**

**print("Bienvenido/a a Python ", nombre)**

***def* despedirse(*nombre*, *dia*):**

**print(*f*"Hasta luego {nombre} en el dia {dia}")**

**#-------------------------------------------------------------**

**#CODIGO PRINCIPAL DE NUESTRO PROGRAMA MAIN**

**print("Ejemplo métodos")**

**saludar("Paco")**

**despedirse("Paco", "Juernes")**

**#LLAMADA LOS METODOS**

**primerMetodo()**

**segundoMetodo()**

**primerMetodo()**

**print("Fin de programa")**

Dentro de los métodos se pueden dividir en dos categorías:

* **Métodos de acción:** Ejecutan una serie de instrucciones
* **Métodos return:** Ejecutan una serie de acciones y devuelven un valor después de ser ejecutados. Siempre deben devolver un valor.

Acción: **print()**

Return: len()

Vamos a visualizar esta nueva teoría dentro del mismo programa en el que estamos.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

**#METODO DE ACCION**

***def* mostrarMenu():**

**print("Seleccione una opción")**

**print("1.- Convertir a mayúsculas")**

**print("2.- Convertir a minúsculas")**

**#METODOS CON RETURN. LA SINTAXIS ES LA MISMA**

***def* convertirMayusculas(*texto*):**

**return texto.upper()**

***def* convertirMinusculas(*texto*):**

**return texto.lower()**

**#-------------------------------------------------------------**

**#CODIGO PRINCIPAL DE NUESTRO PROGRAMA MAIN**

**print("Ejemplo métodos")**

**mostrarMenu()**

**#LOS METODOS QUE DEVUELVEN VALOR DEBEN ALMACENAR EL**

**#VALOR DEVUELTO EN ALGUN SITIO/VARIABLE**

**mayus = convertirMayusculas("curso de python")**

**print(mayus)**

**minus = convertirMinusculas("CURSo dE PyThon")**

**print(minus)**

Tendremos un programa que realizará lo siguiente:

* Pedirá dos números al usuario
* Mostraremos un menú con las acciones que el usuario desee realizar
  + Sumar
  + Multiplicar

Tendremos tres métodos, uno para mostrar el menú, otro para sumar y otro para multiplicar.

Realizaremos todo en un Bucle para que el usuario seleccione siempre opciones hasta salir.

Si el usuario no introduce números, le mostramos un mensaje hasta que nos de números correctos.

Vamos a realizar un método que comprobará nuestro número

Recibirá un STRING y con el método **isdigit()** indicará si es un número o no y se quedará en el método hasta que nos de un dígito.

Creamos un nuevo método llamado **getComprobarNumero()**

Llamamos a nuestro programa **python08metodoscalculadora.py**

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

**#DECLARACION DE LOS METODOS**

***def* sumarNumeros(*num1*, *num2*):**

**return num1 + num2**

***def* multiplicarNumeros(*num1*, *num2*):**

**return num1 \* num2**

***def* mostrarMenu():**

**print("0.- Salir")**

**print("1.- Sumar")**

**print("2.- Multiplicar")**

**print("3.- Introducir nuevos numeros")**

**print("Seleccione una opcion")**

***def* getComprobarNumero():**

**print("Introduzca un numero")**

**#NECESITAMOS UN STRING PARA COMPROBAR**

**aux = input()**

**while (aux.isdigit() == False):**

**print("Esto no es un numero")**

**print("Introduzca un numero")**

**aux = input()**

**num = *int*(aux)**

**return num**

**#---------------------------------**

**#PROGRAMA PRINCIPAL**

**print("Calculadora de métodos")**

**numero1 = getComprobarNumero()**

**numero2 = getComprobarNumero()**

**#CREAMOS UNA VARIABLE OPCION PARA ENTRAR AL BUCLE**

**opcion = 1**

**while (opcion != 0):**

**mostrarMenu()**

**opcion = *int*(input())**

**if (opcion == 1):**

**suma = sumarNumeros(numero1, numero2)**

**print(*f*"La suma de {numero1} y {numero2} es {suma}")**

**elif (opcion == 2):**

**multi = multiplicarNumeros(numero1, numero2)**

**print(*f*"{numero1} \* {numero2} = {multi}")**

**elif (opcion == 3):**

**numero1 = getComprobarNumero()**

**numero2 = getComprobarNumero()**

**else:**

**print("Opcion incorrecta")**

**print("Fin de programa")**

## LIBRERIAS EN PYTHON

Una librería es una funcionalidad con una serie de características de algún tipo.

Son métodos especializados en algo en particular y que no están por defecto en nuestro programa.

Tenemos varias librerías dentro del lenguaje Python, algunas están integradas mediante el compilador de Python y otras tendremos que agregarlas mediante un repositorio de librerías de Python llamado **PyPi**

Para utilizar dichas librerías del repositorio es necesario tener un instalador de librerías llamado **pip**

Por ejemplo, ahora mismo, en nuestro programa no podemos leer ficheros.

Python tiene librerías integradas para leer ficheros lo que pasa es que no las estamos utilizando.

Para utilizar librerías en cualquier programa debemos indicar la librería a utilizar:

* Traer la librería y acceder a sus métodos por su nombre

**PROGRAMA.PY**

import miLibreria

miLibreria.metodo1()

miLibreria.metodo2()

* Traer la librería y acceder a sus métodos por su alias

**PROGRAMA.PY**

import miLibreria as alias

alias.metodo1()

alias.metodo2()

* Podemos traer directamente los métodos o lo necesario de la librería para nuestro programa

**PROGRAMA.PY**

from miLibreria import medodo1(), metodo2()

metodo1()

metodo2()

Existe una librería nativa llamada **math** que nos permite realizar operaciones matemáticas complejas como, por ejemplo, tangente, coseno.

Si queremos utilizar la librería math en nuestro programa:

Texto

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Otra opción

Interfaz de usuario gráfica

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.

Nosotros podemos crear nuestras librerías personalizadas con métodos.

Una librería se utiliza, no se ejecuta.

Se ejecuta un main.

En nuestra arquitectura, vamos a dividir las clases entre Main/Librerias

**LIBRERIA.PY**

def metodo1():

#codigo

def metodo2():

#codigo

**MAIN.PY**

from libreria.py import metodo1, metodo2

Para esta nueva arquitectura cambiamos los nombres a utilizar.

* libraryXXdescripcion.py: Clase de métodos
* mainXXdescripcion.py: Clase de ejecución principal

Vamos a reutilizar código que tenemos.

Lo que haremos será tener una clase de Validaciones de Email o de Nif (Principal).

Tendremos una librería que se encargará de indicar si está bien el Email u otro método que indicará si está bien el NIF.

Para validar un NIF simplemente debemos dividir el número del NIF entre 23 y el resto nos dará un número que corresponde a una letra entre 0 y 23.

Texto, Tabla

El contenido generado por IA puede ser incorrecto.

Creamos una clase llamada **library09validaciones.py**

**LIBRERIA09VALIDACIONES.PY**

***def* validarEmail(*email*):**

**if (email.find("@") == -1):**

**return False**

**elif (email.count(".") == 0):**

**return False**

**elif (email.startswith("@") == True or email.endswith("@") == True):**

**return False**

**elif (email.count("@") > 1):**

**return False**

**elif (email.find("@") > email.rfind(".")):**

**return False**

**else:**

**#posicion del ultimo punto**

**ultimoPunto = email.rfind(".")**

**#Recuperamos el dominio a partir de dicho punto en adelante**

**#no queremos el punto incluido**

**dominio = email[ultimoPunto + 1:]**

**longitud = len(dominio)**

**if (longitud >=2 and longitud <= 3):**

**return True**

**else:**

**return False**

***def* getLetraNif(*numeroDni*):**

**resultado = numeroDni % 23**

**letrasDni="TRWAGMYFPDXBNJZSQVHLCKET"**

**letra = letrasDni[resultado]**

**return letra**

A continuación, creamos la clase principal llamada **main09validaciones.py**

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

**MAIN09VALIDACIONES.PY**

**import library09validaciones as li**

**print("Ejemplo librerias propias")**

**print("Introduzca un email")**

**email = input()**

**respuesta = li.validarEmail(email)**

**print(*f*"El email {email} es {respuesta}")**

**print("Introduzca numero NIF")**

**dni = *int*(input())**

**letra = li.getLetraNif(dni)**

**print(*f*"La letra del NIF {dni} es {letra}")**

**print("Fin de programa")**

Otra sintaxis

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.

## CONJUNTOS DE OBJETOS

Hasta ahora, hemos trabajado con tipos de dato simples, es decir, solamente almacenan un único valor, ya sea String, int o lo que sea.

Dentro de Python existen una serie de objetos para almacenar múltiples elementos.

Todos los objetos conjuntos empiezan en CERO.

Dichos objetos son tres:

1. **Listas**: Son elementos dinámicos que nos permiten almacenar múltiples valores en su interior

Las listas podemos hacer que crezcan o decrezcan en sus elementos.

Se accede a cada objeto de su interior mediante un índice

En una lista podemos almacenar todo lo que deseemos, desde números, String u objetos complejos en su interior

Para averiguar el tamaño de una lista, lo haremos mediante la función **len()**

Tenemos una serie de métodos para trabajar con listas:

* **sort():** Ordena una lista si está compuesta por elementos simples: int, String
* **append(objeto):** Agrega un nuevo elemento a la lista y la lista crece.
* **insert(índice, objeto):** Inserta un elemento en una posición de una lista. La posición del índice debe existir
* **remove(objeto):** Elimina el primer objeto que encuentra dentro de la lista
* **pop(índice):** Elimina un objeto de la lista mediante su posición
* **clear():** Elimina todos los objetos de la lista
* **del objetoList[0:**2]: Elimina desde la posición 0 hasta la posición 2
* **Recuperar un elemento de la lista por su índice:** **objetoLista[índice]**

La diferencia entre los objetos conjuntos que vamos a visualizar está en su declaración.

Para declarar una lista de utilizan los **CORCHETES []**

Cuando hablamos de objetos conjuntos, para recorrerlos tenemos dos opciones:

1. Mediante su longitud utilizando **len()**

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

1. Mediante un bucle de referencia

Un bucle de referencia es un bucle que recorre objetos, no recorre índices.

Dicho bucle lo que hace es utilizar cada objeto en una variable hasta que no existen objetos.

Perdemos la capacidad de tener un índice.

Solamente lo utilizo si deseo recorrer los objetos

Si nos fijamos en la imagen de arriba, estamos utilizando una variable de referencia.

Una variable de referencia es una variable que apunta al mismo objeto de memoria en dos sitios distintos.

nombre = listaNombres[1] LUCIA

print(nombre) LUCIA

listaNombres[1] LUCIA

Sintaxis:

for (VariableReferencia in CONJUNTO):

//INSTRUCCIONES

Texto

El contenido generado por IA puede ser incorrecto.

Un bucle de referencia es de solo lectura, es decir, nunca debemos tocar el CONJUNTO dentro del Bucle.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

En el bucle contador, podemos modificar nuestro CONJUNTO. No nos dará ninguna excepción ni tendremos bucle infinito

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.

Creamos un programa nuevo llamado **python10listas.py**

**print("Ejemplo de listas")**

**#la declaracion de una lista puede ser ya**

**#con los objetos o sin objetos**

**listaNumeros = [12, 88, 11, 21, 33, 22, 31, 2]**

**#ORDEN ASCENDENTE**

**#TODOS LOS METODOS DE LA LISTA MODIFICAN EL OBJETO**

**#ASCENDENTE**

**#listaNumeros.sort()**

**#Tenemos una opción para indicar que el orden sera DESCENDENTE**

**listaNumeros.sort(*reverse*=True)**

**#REALIZAMOS UN BUCLE PARA DIBUJAR LOS DATOS**

**# for i in range(len(listaNumeros)):**

**#     print(listaNumeros[i])**

**#CREAMOS UNA NUEVA LISTA DE NOMBRES**

**listaNombres = ["Adrian", "Lucia", "Antonia", "Diana", "Carlos", "Adrian"]**

**listaNombres.append("Carlos")**

**listaNombres.insert(17, "El nuevo")**

**#PROBAMOS EL METODO remove()**

**#Remove, si no encuentra el nombre, dará Excepción**

**#listaNombres.remove("Carlitos")**

**#Probamos con pop a ver que sucede**

**#Al eliminar, tendremos excepción si no existe**

**#listaNombres.pop(17)**

**print(*f*"Longitud: {len(listaNombres)}")**

**#DIBUJAMOS TODOS LOS NOMBRES**

**for i in range(len(listaNombres)):**

**nombre = listaNombres[i]**

**listaNombres.append("Paco")**

**print(*f*"{i}: {nombre}")**

**# for nombre in listaNombres:**

**#     listaNombres.append("Paco")**

**#     print(nombre)**

**TUPLAS**

Una Tupla es como un Array, es un objeto estático. Se declara con objetos constantes, es decir, no puede crecer y es de solo lectura.

**Las tuplas no se pueden modificar**

Una tupla se declara mediante **paréntesis ()**

Creamos un nuevo programa llamado **python11tuplas.py**

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

**DICCIONARIOS**

Un diccionario son un tipo de estructuras que tienen formato Key:Value

Las claves de un diccionario deben ser únicas, es decir, no podemos repetirlas

Un diccionario es mutable, es decir, podemos agregar elementos a su interior.

Un diccionario es un JSON.

{

“nombre”: “Paco”, “edad”: 44, “email”: [paco@email.com](mailto:paco@email.com)

}

Un diccionario se debe instanciar/crear como objeto.

Para crear un diccionario se realiza con la siguiente sintaxis:

diccionario = dict()

diccionario = {

“nombre”: “Paco”, “edad”: 44, “email”: [paco@email.com](mailto:paco@email.com)

}

También podemos recuperar datos por la clave de un diccionario con el método **get()**

diccionario.get(“nombre”) 🡪 Paco

Existe una forma de recorrer todas las Keys/Values a la vez utilizando un método llamado **items()**

Sintaxis:

for key, value in diccionario.items():

print(f“clave: {key}, valor: {value}”)

Por supuesto, como es un objeto conjunto, también podemos utilizar len() que nos devuelve el número de Keys que existen en el diccionario.

**RECORRER SOLAMENTE LAS KEYS**

Existe una forma de recorrer todas las Keys **keys()**

Sintaxis:

for key in diccionario.keys():

print(f“clave: {key}”)

**RECORRER SOLAMENTE LOS VALUES**

Existe una forma de recorrer todos las Valores **values()**

Sintaxis:

for value in diccionario.values():

print(f“Valor: {value}”)

Podemos eliminar el diccionario y dejarlo vacío utilizando **clear()**

Tenemos la posibilidad de eliminar una Key utilizando el método **pop()**

diccionario.pop(“nombre”)

Por supuesto, podemos agregar elementos Keys/Value a nuestro diccionario mediante el método **setDefault()**

diccionario = dict()

diccionario = {

“nombre”: “Paco”, “edad”: 44, “email”: [paco@email.com](mailto:paco@email.com)

}

diccionario.setDefault(“profesion”: “Profesor”)

Vamos a crear un diccionario de provincias para visualizar cómo podemos trabajar con un diccionario.

Creamos un nuevo programa llamado **python12diccionario.py**

**print("Ejemplo de diccionarios")**

**#CREAMOS EL DICCIONARIO DE PROVINCIAS**

**provincias = *dict*()**

**provincias = {**

**924 : "Badajoz",**

**956:  "Cádiz",**

**958 : "Granada",**

**959 : "Huelva",**

**91 : "Madrid",**

**95 : "Málaga",**

**968 : "Murcia"**

**}**

**#AÑADIMOS UNA NUEVA PROVINCIA, SI LA KEY NO EXISTE LO INSERTA**

**provincias.setdefault(976, "Zaragoza")**

**#QUE SUCEDE SI EXISTE????**

**provincias.setdefault(968, "Albacete")**

**provincias.pop(958)**

**print(*f*"Número de provincias: {len(provincias)}")**

**for key, value in provincias.items():**

**print(*f*"key: {key}, valor: {value}")**

**print("--------------KEYS------------------")**

**for key in provincias.keys():**

**print(*f*"key: {key}")**

**print("--------------VALUES------------------")**

**for value in provincias.values():**

**print(*f*"Valor: {value}")**

## CONTROL DE EXCEPCIONES

Un control de excepciones nos permite controlar el código cuando sucede algo imprevisto.

Cuando sucede un error, el programa se detiene y no sigue adelante, pero, pudiera ser que yo necesitase que el programa siguiera por la razón que sea.

Por supuesto, el control de excepciones se debería realizar al final de cualquier desarrollo y con elementos del cuales no tengamos control, por ejemplo, si estamos leyendo un Api, puede ser que nuestro código sea perfecto y todo funcione, pero también puede ser y no tenemos control sobre ello que nos quedemos sin internet.

Para esta teoría vamos a ir realizando un ejemplo sencillo y capturando las excepciones y viendo cómo funcionan los bloques try…catch

Comenzamos creando un nuevo programa llamado **python13excepciones.py**

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Si introducimos una letra, veremos una excepción

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Cada excepción tiene un NOMBRE único y propio, no será lo mismo una excepción de conversión que una excepción de abrir un fichero.

En nuestro ejemplo estamos viendo una excepción llamada **ValueError** y podemos personalizar qué deseamos hacer cuando suceda dicho error para que el programa no se detenga

**CONTROL DE EXCEPCIONES**

**try:**

#CODIGO A EJECUTAR

**except NOMBRE\_EXCEPCION:**

#CODIGO SI SALTA LA EXCEPCION

Vamos a controlar nuestra excepción actual.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

También tenemos la posibilidad de volver a pedir un número si lanza la excepción.

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

Podemos capturar más de una excepción.

Cada excepción para capturar debemos incluirla dentro de un bloque **except** con el nombre de cada excepción.

Por ejemplo, existe una Excepción que es lanzada cuando dividimos entre cero.

Texto

El contenido generado por IA puede ser incorrecto.

La siguiente situación es tener una excepción GENERAL, es decir, yo solamente quiero capturar excepciones independientemente del tipo, si tenemos una excepción, mostramos un mensaje simple.

Para esta sintaxis se utiliza la palabra **except** sin indicar el tipo.

Para realizar esto, tenemos varias posibilidades:

1. Capturar varias excepciones y, si sucede algo más que no sé lo que puede ser, también capturo la excepción, da igual el tipo. Por ejemplo, en nuestro bloque de código actual.

**El bloque except debe ir al final**

Texto

El contenido generado por IA puede ser incorrecto.

1. Capturamos todas las excepciones a la vez, no importa el tipo.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Por último, cuando tenemos un bloque except, no sabemos el tipo, si queremos mostrar un mensaje con el tipo de lo qué sucede en una excepción general, debemos hacerlo mediante una función que está dentro de la librería **sys** y que se llama **exc\_info()**

Texto

El contenido generado por IA puede ser incorrecto.

Por último, este ejemplo yo nunca lo utilizaría en entorno real, simplemente hacemos IF y si el número es CERO, preguntamos de nuevo.

Las excepciones están para cuando NO tenemos control de código.

Tenemos un último bloque llamado **finally** y que es opcional completamente dentro de **try..catch**

Sintaxis:

**try:**

#CODIGO A EJECUTAR

**except NOMBRE\_EXCEPCION:**

#CODIGO SI SALTA LA EXCEPCION

**finally:**

#CODIGO QUE SIEMPRE SE EJECUTA

Finally es un bloque opcional

Imaginemos que estamos leyendo de una base de datos.

1. Abrir conexión con BBDD
2. Ejecutar una consulta
3. Leer datos
4. Cerrar conexión

conexion.abrir()

consulta.execute()

consulta.read()

conexion.cerrar()

Suceda lo que suceda, una conexión NUNCA debe permanecer abierta.

Ejemplo real de Finally:

**try:**

conexion.abrir()

consulta.execute()

consulta.read()

**except NOMBRE\_EXCEPCION:**

#CODIGO SI SALTA LA EXCEPCION

print(“Tenemos un problema, Houston”)

**finally:**

#CODIGO QUE SIEMPRE SE EJECUTA

conexion.cerrar()

## PROGRAMACION ORIENTADA A OBJETOS

Dentro de Python, debemos separar los métodos de las clases.

Un programa de Python, como el que hicimos de la calculadora con los métodos es una librería.

Una librería está compuesta por funciones. Una función también es un método.

Nosotros podemos tener librerías que sean solamente de métodos.

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.

Otra arquitectura distinta es tener clases.

Una clase nos permite almacenar información de un objeto compuesto y que tenga métodos dependiendo del objeto.

Por ejemplo, pongamos que deseamos almacenar el nombre de una ciudad, declaramos un string con dicho nombre

En cuanto sea más complejo, necesitamos más variables.

También quiero la temperatura

Necesito la población

ciudad = “Alicante”

temperatura = 30

población = 310000

Tenemos tres variables, si necesitamos almacenar más ciudades en una lista, necesitamos tres listas:

Ciudades = [“Alicante”, “Cartagena”, “Oviedo”]

Temperaturas = [30, 28, 18]

Población = [310000, 150000, 160000]

En este supuesto, estaría bien tener un objeto que tuviera propiedades tales como nombre ciudad, temperatura y población.

ciudad = new Ciudad()

ciudad.nombre = Alicante

ciudad.temperatura = 30

ciudad.poblacion = 310000

ciudad.quemarHogueras()

Una clase está compuesta por propiedades y por métodos.

Al crear una clase en Python se utiliza la siguiente instrucción:

class NombreClase:

#METODOS/PROPIEDADES

Dentro de una clase existen multitud de características además de propiedades y métodos.

Dentro de POO existen varias características dependiendo lo que deseemos realizar.

* Encapsulación: Solamente veremos lo que deseemos en una clase. Este concepto sirve para “proteger” nuestras variables internas de una clase y que otros programas NO puedan ver dichas variables.
* Polimorfismo: Un método que realiza la misma función puede tener múltiples formas. Este es el ejemplo que os he comentado antes sobre el pedal de freno
* Herencia: Una clase hereda de otra y, a su vez, recupera todas las características de la clase que hereda y las implementa.
* Abstracción: Podemos abstraernos para manejar un objeto de una misma forma con su clase **super()**

**CONCEPTO CONSTRUCTOR**

Un constructor es un método de la clase.

Es un código que nos permite realizar acciones cuando se crea el objeto por primera vez.

En Python un constructor se define con la palabra **def** y con el nombre **\_\_init\_\_**

Es el primer código que se lee en una clase.

Un constructor es opcional.

**PALABRA CLAVE SELF**

La palabra clave **self** es obligatoria en los métodos de una clase.

Nos permite acceder al objeto que están utilizando.

Debe ser el primer parámetro que tengamos dentro de una clase.

Ejemplo de una clase llamada **Lámpara**

class Lampara:

#AQUI PODEMOS DECLARAR LAS PROPIEDADES

Encendida

Color

Iluminacion

#DECLARAR UN CONSTRUCTOR PARA INICIAR EL ESTADO DE LAS PROPIEDADES

def \_\_init\_\_(**self**):

**self**.Encendida = false

**self**.Color = Azul

**self**.Iluminacion = 0

#METODOS PARA LA LAMPARA

def encender(**self**):

**self**.Encendida = true

def aumentarIluminacion(**self**, brillo):

self.Iluminacion = brillo

Cuando “alguien” utilice mi lámpara desde otro programa de Python

miLampara = new Lampara()

¿Cómo está la lámpara ahora mismo? Encendida a False, de color azul y cero de iluminación

miLampara.enceder()

miLampara.aumentarIluminacion(20)

Vamos a realizar un ejemplo, pero esta vez, ya vamos a ser organizados.

Las clases las tenemos ahora mismo por ahí tiradas, lo suyo es organizar nuestras clases por carpetas/funcionalidad.

Por ejemplo, si tenemos librerías, deberíamos tener una carpeta llamada **libraries** y tener ahí las clases de librería.

En Python nos dan la posibilidad de tener las clases en cualquier programa, no importa.

Incluso, podemos tener un solo fichero de Python con todas las clases en su interior.

En mi cerebro me gusta separar las clases por su acción.

Tendremos nuestras clases dentro de una carpeta llamada **models**

Dentro de la carpeta creamos una nueva clase llamada **persona.py**

Por otro lado, vamos a tener una clase de ejecución en la que probaremos lo que hagamos con Personas.

Creamos un programa llamado **python14usopersonas.py**

Estamos haciendo un Coche con 3 pedales de freno

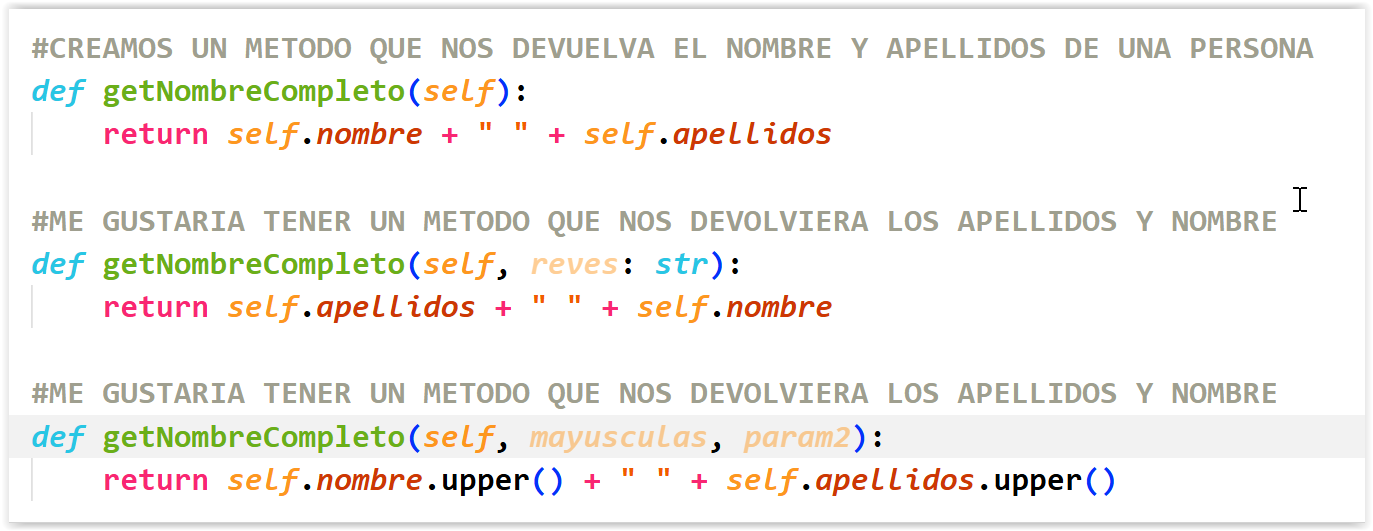
Texto

El contenido generado por IA puede ser incorrecto.

Yo solo quiero un método para el nombre completo y que me devuelva los datos dependiendo cómo actúe sobre dicho método

Para poder tener un solo método necesitamos que cada método que tengamos reciba parámetros distintos en su número.

No podemos tener el mismo nombre de método con los mismos parámetros.



Esto no es nativo en Python y necesitamos instalar alguna librería de **overloads**

<https://github.com/Hernanatn/sobrecargar.py>

***class* Persona:**

**#TENEMOS DOS FORMAS DE DECLARAR PROPIEDADES**

**#1) AQUI AL INICIO DE LA CLASE**

**# nombre = ""**

**# apellidos = ""**

**# edad = 0**

**# pais = "lo que sea"**

**#EL CONSTRUCTOR ES UN METODO PARA CREAR PROPIEDADES**

**#E INICIAR DICHAS PROPIEDADES**

**#2) CREAR LAS PROPIEDADES EN EL CONSTRUCTOR**

***def* \_\_init\_\_(*self*):**

**#PODEMOS TENER PROPIEDADES PRIVADAS**

**#UNA PROPIEDAD PRIVADA SE CREA MEDIANTE EL DOBLE GUION BAJO**

**# private:  \_\_propiedadPrivada**

***self*.*\_\_dni* = "12345678X"**

***self*.*pais* = "Alemania"**

***self*.*edad* = 0**

***self*.*nombre* = ""**

***self*.*apellidos* = ""**

**#CREAMOS UN METODO PARA VER QUE PODEMOS ACCEDER AL DNI**

***def* getPrivateDni(*self*):**

**return *self*.*\_\_dni***

**#CREAMOS UN METODO QUE NOS DEVUELVA EL NOMBRE Y APELLIDOS DE UNA PERSONA**

***def* getNombreCompleto(*self*, *dato*: *int*):**

**return *self*.*nombre* + " " + *self*.*apellidos***

**# #ME GUSTARIA TENER UN METODO QUE NOS DEVOLVIERA LOS APELLIDOS Y NOMBRE**

**# def getNombreCompleto(self, \*\*kwargs):**

**#     return self.apellidos + " " + self.nombre**

**# #ME GUSTARIA TENER UN METODO QUE NOS DEVOLVIERA LOS APELLIDOS Y NOMBRE**

**# def getNombreCompleto(self, mayusculas, param2):**

**#     return self.nombre.upper() + " " + self.apellidos.upper()**

El siguiente concepto es utilizar la HERENCIA en PYTHON

La herencia dentro de programación orientada a objetos implica que un objeto recupera las propiedades de otro objeto en la herencia.

El ejemplo que os he puesto sobre un Coche y un Deportivo.

Sintaxis para la herencia en Python es la siguiente:

class ClaseHereda(claseDeLaQueHeredamos):

#METODOS Y PROPIEDADES DE LA HERENCIA

Tenemos una nueva palabra para cuando estamos heredando, que es la palabra **super()** que hace referencia a la clase de la que estamos heredando.

Existe otro concepto que es sobrescribir, que simplemente lo que hace es adapta un método de la clase super() y lo escribe de otro forma en la clase que hereda.

Vamos a crear una nueva clase llamada **Empleado.py** dentro de **models**

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Si probamos nuestra clase Empleado accediendo a propiedades que hemos inicializado y que están en Persona, podemos ver el contenido.

Si no las hemos inicializado, es como si No existieran. Para Python, las propiedades de persona no existen hasta que no las **utilizamos** en el objeto Empleado.

Conclusión: Recupera sus características pero no las implementa.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Las que sí que implementa son las que yo he puesto de forma explicita en el constructor y no hace falta que las inicie dentro de nuestro Empleado, por ejemplo, el Salario.

Creamos otra clase llamada **Director.py** dentro de **models**

La herencia nos permite un nuevo concepto llamado **override** que es sobrescribir un método que tiene una clase **super()** y adaptarlo a nuestra clase a nuestra conveniencia.

En Python no es necesaria ninguna palabra clave para sobrescribir, simplemente que el método se llame igual.

Un Empleado tiene 22 días de vacaciones y un Director tendrá 30 días de vacaciones.

Implementamos la clase Director.

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.

Actualmente, un Director y un Empleado ahora mismo tienen los mismos días de vacaciones.

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

Los dos tienen 22 días

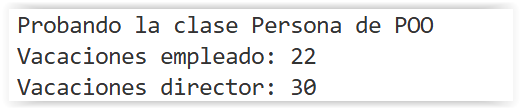
Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.

Es un concepto de sustitución



Otro concepto es, en lugar de sustituir, poder **reutilizar**, es decir:

Quiero que un Director tenga **8 días más** de vacaciones que un empleado.

Ahora mismo, esto no se cumple.

Para poder hacer esto, debemos llamar al método de la clase **super()** del Empleado llamado **GetVacaciones()**, recuperar el valor que tenga un Empleado y sumar 8 días más.

**DIRECTOR**

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.

Texto

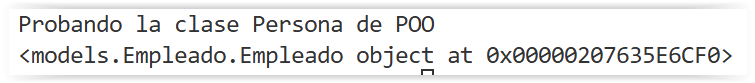
El contenido generado por IA puede ser incorrecto.

Con el mismo concepto, tenemos dentro de las clases un método llamado **\_\_str\_\_** nativo que podemos sobrescribir e indicar cómo queremos que se dibuje un objeto determinado en el momento de pintarlo por pantalla.

Por ejemplo, pongamos que dibujamos un objeto Empleado.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.



Si sobrescribimos el método **\_\_str\_\_** podremos dibujar al empleado como deseemos.

**EMPLEADO**

Texto

El contenido generado por IA puede ser incorrecto.

Y podremos comprobar el resultado.

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

Por último, como vimos en la anterior clase, una clase que HEREDA no recupera directamente los datos de otra clase en Python.

Esto sucede porque en el lenguaje Python, NO se llama de forma implícita al constructor de su clase SUPER.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

## ENVIRONMENTS

Un entorno es un lugar aislado de nuestro SO de Python.

Nos permite instalar librerías o características de un determinado Framework cuando no tenemos permisos o no queremos que las librerías de un proyecto interfieran en otro proyecto.

Cada entorno es aislado y permite trabajar de forma independiente.

En algunos frameworks es imprescindible crear un entorno aislado de trabajo.

Por ejemplo: Jupyter, Django, Fast Api, IA

**PIP**

Pip es el repositorio dónde se encuentran nuestras clases de Python, las clases externas que podemos agregar.

Dichas clases nos permiten utilizar Jupyter, Django, MySql, Api, Files

Vamos a probar la funcionalidad de nuestro entorno en un proyecto nuevo.

En dicho proyecto haremos lo siguiente:

1. Tendremos acceso a datos mediante un API.
2. Dividiremos nuestro proyecto en módulos
3. Crearemos un entorno para instalar las librerías necesarias
4. Tendremos un fichero **main** que será el encargado de dibujar y hacer todas las características que necesitemos.