**SINTAXIS BÁSICA**

Si quiero saber el resto de la división hay que usar: 10%3-> Así me devuelve el resto de esa división.

La potencia es:\*\*; así 5\*\*3=125

Si quiero el valor entero de la división: 9//2=4, porque si pongo 9/2 saldrá 4´5

Para saber la clase de una variable se utiliza type (nombre de la variable)

Funciones:

**LISTAS**

miLista=["María", "Pepe", "Marta", "Antonio"]

print(miLista[:]) para imprimir la lista, si haces print(miLista[:2]) o print(miLista[2:]) estás haciendo que se imprima desde el principio hasta ese número o desde ese número hasta el final.

print(miLista[0:2]) El primer número es desde donde se imprime y el segundo es donde se corta (sin incluir)

miLista.apend sirve para añadir valores al final de la lista

miLista.insert(2,"Sandra") para incluir un valor a la lista en el lugar que queramos

miLista.extend(["Sandra", "Ana", "Lucía"]) para incluir más valores a la lista

print(miLista.index("Antonio")) para saber en que posición se encuentra el valor que estamos buscando, si hay dos o varios siempre nos devolverá la primera posición en la que encuentre ese valor que buscamos

print("Pepe" in miLista) y saldrá true o false según si ese valor está o no en la lista

miLista.remove("Pepe") para eliminar algún valor de la lista

miLista.pop() elimina el último valor de la lista

miLista3=miLista+miLista2 así se unen dos listas

miLista=["María", "Pepe", "Marta", "Antonio"] \*3 hace que esta lista se repita ese número de veces

**Tuplas**

Es una lista inmutable, no se pueden modificar con las expresiones de arriba en las listas, salvo extraer porciones y comprobar si un elemento se encuentra en la tupla, permiten búsquedas (index). Se pueden escribir sin ( ) pero es recomendable ponerlos. Si en una función o generador pongo: def devuelveciudades(\*ciudades): ese asterisco quiere decir que va a recibir un número indeterminado de elementos y que los va a recibir en forma de tupla

Se ejecutan más rápido que las listas, ya que ocupa menos espacio de memoria, permiten formatear strings y se pueden utilizar como clave en un diccionario.

mitupla =("Juan", 5,45,1997) para escribir una tupla

print(mitupla[2]) para ver la posición en la tupla

milista=list(mitupla) pasar la tupla a una lista

mitupla=tuple(milista) para pasar la lista a una tupla

print(mitupla.count(5)) para que me cuente el número de veces que aparece dentro de la tupla el elemento que queramos

print(len(mitupla)) para saber el número de elementos que hay en la tupla

mitupla=("Juan",) para hacer una tupla unitaria (no olvidar la ,)

nombre, dia, mes, agno = mitupla asignar cada uno de los valores de la tupla por orden a las variables declaradas (desempaquetado de tupla)

**Diccionarios**

El orden es indiferente. Los datos se almacenan asociados a una clave de tal forma que se crea una asociación clave : valor.

midiccionario={"Alemania":"Berlín", "Francia":"París", 23:"Jordan"} para escribirlo

print(midiccionario) para ver el diccionario

print(midiccionario["Alemania"]) para que me devuelva el valor asociado a esa clave

midiccionario["Italia"] = "Roma" para añadir una nueva asociación clave : valor al diccionario

del midiccionario ["Inglaterra"] para borrar una parte de ese diccionario

**mitupla=["España", "Francia", "Alemania"]**

**midiccionario={mitupla[0]: "Madrid", mitupla[1]: "París", mitupla[2]: "Berlín"}**

Para poder pasar una tupla a un diccionario

midiccionario={23:"Jordan", "Nombre":"Michael", "Anillos":[1991,1992,1993,1996]} para asignarle más valores a una clave

midiccionario={23:"Jordan", "Anillos":{"temporadas":[1991,1992,1993,1996]}} diccionario dentro de un diccionario

print(midiccionario.keys( )) nos da las claves del diccionario

print(midiccionario.values( )) nos da los valores del diccionario

print(len(midiccionario)) para ver el número de calve : valor que hay en el diccionario

**Condicionales**

if nota < 5: los dos puntos se usan para saber dónde acaba la condición

elif para poner entre if y else para hacer varias de esas

Si quieres poner más condiciones en el if puedes usar el **and** u **or** para añadirlas

Tambien se puede usar **in (aquí separado por comas lo que queremos comprobar)** para ver si lo que he escrito coincide con lo que debe de escribirse o no (ejemplo mas\_condicionales)

**Bucles**

Determinado: **for** variable **in** elemento a recorrer (lista, tupla, cadena...):

Recordar que el bucle recorre todos los parámetros que haya en la cadena o los numeros, por ejemplo, **for** i **in** "manuurodriguez96@gmail.com": lo que ocurre es que la variable i va cogiendo cada uno de los caracteres de la cadena, por lo que empezará valiendo m, a, n...

Si quieres que al imprimir por pantalla aparezca todo en una misma linea debes escribir dentro del cuerpo del bucle print("lo que sea", end=" ")

**for** i **in** range(5): para hacer que se ejecute el cuerpo del bucle 5 veces (empieza en 0,1,2,3,4)

Si escribimos: print(f"valor de la variable {i} ") aparece antes del valor de la variable eso que hemos escrito

**for** i **in** range(4,50,4): quiere decir que empieza en el 4, hasta el 49 (empieza en el 0) y va contando de 4 en 4.

**for** i **in** range(len(email)): En este caso len de email responde con el número de caracteres del email, la longitud del string, y el range hace que de tantas vueltas en el bucle como numero de caracteres del email

Bucles indeterminados: **While** condición:

Condiciones continue (se salta una vuelta de bucle), pass (devolver null cuando se lee, poco usada, al definir clases o dejar un bucle para hacer después) y else (su uso es el mismo que dentro de un condicional)

**Generadores**

Devuelven un solo valor de lo que queremos generar, al contrario que las funciones tradicionales que generan todos los valores para después devolverlo. Son más eficientes que las funciones tradicionales. Muy útiles con listas de valores infinitos. Por ejemplo, para hacer una lista de números pares primero generaría el 2, luego el 4, 6, 8... y así.

**Def** generaNumeros(): **yield** nuemeros (abajo e indexado para devolverlo, como el return de las funciones)

**Excepciones**

Son errores que ocurren durante la ejecución del programa. La sintaxis del código es correcta pero durante la ejecución ha ocurrido algo inesperado. Este tipo de errores se pueden "saltar" y que por lo menos continúe el resto del código.

**try:** lo que quieres que intente y ya más abajo se pone **except** ZeroDivisionError (o el error que sea): y lo que quieres que haga el programa si ocurre este error.

Lanzar la excepción de forma intencionada: **raise TypeError o el que sea ("mensaje de error")** (para crear excepciones propias)

**Programación orientada a objetos**

Clases (coche) --> Propiedad del obejto (chasis, ancho, ruedas...) -->

Las funciones no pertenecen a ninguna clase, si queremos que pertenezcan a una clase entonces habrá que crear un método: def function(self )-> objeto perteneciente a la clase): pass

Para acceder a propiedades de objeto desde código se usa el "."

self.\_\_largochasis=250 --> Esos \_\_ es para encapsular esa propiedad y que no se pueda modificar desde fuera