**Ejecutar Python**

En la terminal debo escribir Python y ahí ya estoy usando el asistente (Shell) de python, para salir de ese, escribo exit ()

**Variable declaration:** Es el proceso de asignar un nombre a una variable y reservar espacio en memoria para almacenar datos. Se puede realizar simplemente escribiendo el nombre de la variable seguido de un signo de igual (=) y el valor que deseas asignarle.

**Log statement:** Es una declaración de registro que se utiliza para imprimir información o mensajes en la consola o en un archivo de registro. En Python, puedes usar la función **print()** para imprimir mensajes en la consola.

**Type check:** Es el proceso de verificar el tipo de dato de una variable. Puedes utilizar la función **type()** para obtener el tipo de dato de una variable.

numero = 10

print(type(numero))  # Imprime <class 'int'>, que indica que la variable 'numero' es de tipo entero (integer).

**Length check:** Se utiliza para verificar la longitud de una secuencia, como una cadena, una lista o una tupla. Puedes utilizar la función **len()** para obtener la longitud de una secuencia.

cadena = "Hola"

print(len(cadena))  # Imprime 4, que es la longitud de la cadena "Hola".

**Comment:**

* Single line: Un comentario de una sola línea se utiliza para agregar anotaciones o explicaciones en el código. Se indica utilizando el símbolo numeral (#) seguido del comentario. Por ejemplo:
* Multiline: Un comentario multilineal se utiliza para comentarios extensos o para desactivar temporalmente bloques de código. Se puede hacer utilizando triple comilla (""" o ''') al comienzo y al final del comentario.

**Data Types:**

* Primitive:

Boolean: Representa un valor de verdad, que puede ser True (verdadero) o False (falso).

Numbers: Pueden ser enteros (int) o de punto flotante (float) para representar valores numéricos.

Strings: Representan secuencias de caracteres y se pueden definir utilizando comillas simples o dobles.

* Composite:

Listas (lists): Secuencias ordenadas y mutables de elementos.

Tuplas (tuples): Secuencias ordenadas e inmutables de elementos.

Diccionarios (dictionaries): Colecciones no ordenadas de pares clave-valor.

En Python, las tuplas, las listas y los diccionarios son ***tipos de datos*** utilizados para almacenar y organizar información de diferentes formas. A continuación, se explica cada uno de ellos:

Tuplas (tuples):

* Definición: Las tuplas son secuencias ***ordenadas e inmutables*** de elementos. Se crean ***utilizando paréntesis ()*** y los elementos ***se separan por comas.***
* Uso:

Almacenar datos que no deben modificarse, como coordenadas geográficas o fechas.

Devolver múltiples valores desde una función.

Utilizar tuplas como claves en un diccionario (debido a su inmutabilidad).

* Funciones:

Inicializar(initialize): Se pueden inicializar utilizando paréntesis () o simplemente separando los elementos por comas.

Acceder a un valor(acces value): Se puede acceder a los valores de una tupla mediante su índice utilizando corchetes [].

Cambiar un valor(change value): Las tuplas son inmutables, por lo que no se pueden cambiar valores individuales en una tupla después de su creación.

Agregar un valor(add value): No se pueden agregar valores a una tupla existente, ya que son inmutables.

Eliminar un valor(delete value): No se pueden eliminar valores de una tupla existente, ya que son inmutables.

Ejemplo: coordenadas = (10, 20)

meses\_del\_ano = ("enero", "febrero", "marzo")

Listas (lists):

* Definición: Las listas son ***secuencias ordenadas y mutables de elementos***. Se crean ***utilizando corchetes []*** y los elementos ***se separan por comas.***
* Uso:

Almacenar una colección de elementos que pueden modificarse a lo largo del programa.

Iterar sobre los elementos de una lista.

Realizar operaciones como búsqueda, inserción y eliminación de elementos.

* Funciones

Inicializar: Se pueden inicializar utilizando corchetes **[]** o utilizando la función **list()**.

Acceder a un valor: Se puede acceder a los valores de una lista mediante su índice utilizando corchetes **[]**.

Cambiar un valor: Se puede cambiar un valor de una lista asignando un nuevo valor a un índice específico.

Agregar un valor: Se puede agregar un valor a una lista utilizando el método **append()** o utilizando la concatenación de listas.

Eliminar un valor: Se puede eliminar un valor de una lista utilizando el método **remove()** o la palabra clave **del**.

Ejemplo: numeros = [1, 2, 3, 4, 5]

nombres = ["Juan", "María", "Luisa"]

Diccionarios (dictionaries):

* Definición: Los diccionarios son ***colecciones no ordenadas de pares clave-valor.*** Se crean utilizando ***llaves {}*** y los ***pares clave-valor se separan por comas.***
* Características:

Mutabilidad: Los valores asociados a las claves se pueden modificar, agregar o eliminar.

No tiene orden definido: Los elementos de un diccionario no mantienen un orden específico.

Claves únicas: Las claves en un diccionario deben ser únicas.

* Uso:

Almacenar y recuperar datos utilizando claves personalizadas.

Representar estructuras de datos complejas y anidadas.

Mapear información relacionada de manera eficiente.

* Funciones:

Inicializar: Se pueden inicializar utilizando llaves {} o utilizando el constructor dict().

Acceder a un valor: Se puede acceder a los valores de un diccionario utilizando una clave específica entre corchetes [] o utilizando el método get().

Cambiar un valor: Se puede cambiar el valor asociado a una clave en un diccionario asignándole un nuevo valor.

Agregar un valor: Se puede agregar un nuevo par clave-valor a un diccionario utilizando una nueva clave y asignándole un valor.

Eliminar un valor: Se puede eliminar un par clave-valor de un diccionario utilizando la palabra clave del o el método pop().

Ejemplo: persona = {"nombre": "Juan", "edad": 30, "ciudad": "Madrid"}

estudiantes = {"A001": "María", "A002": "Pedro", "A003": "Luisa"}

**Condicional:**

* **if** (si): Es una estructura condicional que permite ejecutar un bloque de código si una condición es verdadera. Si la condición no se cumple, el bloque de código no se ejecuta

if condicion:

    # Bloque de código si la condición es verdadera

* **else** (si no): Se utiliza junto con **if** para proporcionar una alternativa en caso de que la condición sea falsa.

if condicion:

    # Bloque de código si la condición es verdadera

else:

    # Bloque de código si la condición es falsa

* **elif (sino si):** Se utiliza para evaluar condiciones adicionales después de if. Si la condición anterior no se cumple, se evalúa la siguiente condición.

if condicion1:

    # Bloque de código si la condición1 es verdadera

elif condicion2:

    # Bloque de código si la condicion2 es verdadera

else:

    # Bloque de código si ninguna condición es verdadera

**Bucle for:**

* **start** (inicio): Es el valor inicial desde el cual comenzará el bucle.
* **stop** (parada): Es el valor en el cual el bucle dejará de ejecutarse.
* **increment** (incremento): Especifica cómo cambiará la variable de control en cada iteración del bucle.
* **break**: Se utiliza para salir del bucle antes de que se complete normalmente.
* **continue**: Se utiliza para omitir el resto del bloque de código en una iteración particular y pasar a la siguiente iteración.
* **sequence** (secuencia): Es una secuencia iterable (como una lista, tupla o cadena) que se utilizará en el bucle.

for variable in sequence:

    # Bloque de código a repetir

**Bucle while:**

* **start** (inicio): Es la condición inicial o valor de inicio antes de comenzar el bucle.
* **stop** (parada): Es la condición que se evalúa en cada iteración y determina cuándo se debe detener el bucle.
* **increment** (incremento): Se utiliza para actualizar o modificar la condición en cada iteración para evitar bucles infinitos.

while condicion:

    # Bloque de código a repetir mientras la condición sea verdadera

**Función:**

* **parameter** (parámetro): Es una variable utilizada para recibir valores o información en una función. Los parámetros se definen en la declaración de la función.
* **argument** (argumento): Es el valor real o dato proporcionado a un parámetro al llamar a una función.
* **return**: Es una palabra clave utilizada para devolver un resultado o valor desde una función al lugar donde se llamó. El valor devuelto puede ser utilizado en el código posterior.

def nombre\_de\_funcion(parametro1, parametro2):

    # Bloque de código de la función

    return resultado

**Para llamar a la función y pasar los argumentos:**

nombre\_de\_funcion(argumento1, argumento2)

**Errores:**

* **NameError**: Se produce cuando intentas utilizar una variable que no ha sido definida.
* **TypeError**: Ocurre cuando se realiza una operación o función en un tipo de dato incorrecto.
* **KeyError**: Sucede cuando intentas acceder a una clave que no existe en un diccionario.
* **IndexError**: Ocurre cuando intentas acceder a un índice que está fuera del rango válido en una lista o tupla.
* **IndentationError**: Se produce cuando hay un error en la indentación del código, como una falta de sangría o un nivel incorrecto de sangría.
* **AttributeError**: Sucede cuando intentas acceder a un atributo o método que no existe para un objeto en particular.

*def* add(a,b): # nombre de la función: 'add', parámetros: a y b

x = a + b # proceso

return x # devuelve valor: x

Hemos declarado una función con la palabra clave def la hemos nombrado *add*, y especificado que necesita dos entradas (parámetros). Si esto es todo lo que tenemos en nuestro archivo, no debería suceder nada si lo ejecutamos. Para realmente ejecutar la función, debemos ejecutarla al **invocarla**o **llamarla**. Esto se hace fuera de la función usando el nombre de la función seguido por paréntesis **( ).** Dentro de los paréntesis se encuentra cualquier valor (argumentos) que la función espera como entrada.

new\_val = add(3, 5) # llamar la función con los argumentos 3 y 5

print(new\_val) # el resultado de la función add se devuelve y guarda en new\_val, por lo que veremos 8

Espera, ¿cuál es la diferencia entre un parámetro y un argumento? Estas dos palabras se mezclan mucho en programación. Es este ejemplo,**'name' es un parámetro**, mientras que **"Michael", "Anna" y "Eli" son argumentos**. **Definimos parámetros.** **Pasamos argumentos** a funciones.

Es muy importante recordar lo siguiente: **la llamada a una función es igual a lo que sea que la función devuelva**. Es posible que esto no tenga sentido hasta que lo veamos en acción.

Modifiquemos la función *di\_hola* original y observemos las diferencias:

def di\_hola(nombre):

return "Hola " + nombre

saludo = di\_hola("Michael") # el valor devuelto por la función di\_hola se asigna a la variable 'saludo'

print(saludo) # esto dará como resultado 'Hola Michael'

**Parámetros por defecto**

# establece los valores predeterminados al declarar los parámetros

def sé\_alegre(name='', repeat=2):

print(f"buenos días {name}\n" \* repeat)

sé\_alegre()    # salida: buenos días (repetida en dos líneas)

sé\_alegre("tim")    # salida: buenos días tim (repetida en dos líneas)

sé\_alegre(name="john")    # salida: buenos días john (repetida en dos líneas)

sé\_alegre(repeat=6)    # salida: buenos días (repetida en 6 líneas)

sé\_alegre(name="michael", repeat=5)    # salida: buenos días michael (repetida en 5 líneas)

# NOTA: el nombre de los argumentos no importa si somos explícitos al enviarlos

sé\_alegre(repeat=3, name="kb")    # salida: buenos días kb (repetida en 3 líneas)

* no se proporcionan argumentos: se utilizan los valores por defecto
* se proporciona un argumento *unnamed:* el valor proporcionado se usa como valor para el primer parámetro, y se usa el valor predeterminado del segundo parámetro
* se proporciona un argumento *named*: el valor proporcionado se usa como el valor del parámetro del mismo nombre, y se usa el valor predeterminado del otro parámetro
* se proporcionan ambos argumentos *unnamed*: valores asignados a los parámetros en orden (es decir, lo que hemos estado haciendo hasta este momento)
* se proporcionan ambos argumentos *named*: los valores se asignan al parámetro asociado (¡y así el orden no importa!)

def multiplicar(num\_list, num):

print(num\_list, num)

for i in range(len(num\_list)):

num\_list[i] \*= num

return num\_list

a = [2, 4, 10, 16]

b = multiplicar(a, 5)

print(b)

Reflexión de jaz sobre el código de arriba, tenemos una función que se llama multiplicar, en donde se indica que se le dará dos parámetros, un num\_list y un num, se imprimen ambos para asegurarnos que funcióna como queremos. Después entra a un bucle for que tiene un rango que toma el largor de la lista, se indica que cada valor de la lista se multiplique por el valor de num y se retorne el valor de esa multiplicación.

**Diferencia entre argumento y parametro**

Se pasa un argumento cuando se llama a una función. Un parámetro se define durante la creación de la función y funciona de manera muy similar a una variable.

Un parámetro es un contenedor. Un argumento es un valor.

**Para comprobar si la variable x contiene algo, ¿cómo lo harías?**

En Python, la expresión **not x** se utiliza para comprobar si la variable **x** está vacía o no contiene nada. Si **x** es una variable que no tiene ningún valor asignado, o si es una cadena vacía, una lista vacía, un diccionario vacío, entre otros casos, la expresión **not x** evaluará como verdadera.

**Función comprensión de una lista en Python**

**Sintaxis** nueva\_lista = [expresion for elemento in iterable if condicion]

Donde:

* **expresion**: Es la expresión que se evalúa para cada elemento del iterable y define el valor que se agregará a la nueva lista.
* **elemento**: Es una variable que representa cada elemento del iterable (en este caso, los elementos de la lista original).
* **iterable**: Es el iterable del cual se obtienen los elementos (en este caso, la lista original).
* **condicion**: Es una condición que se evalúa para cada elemento. Si la condición es verdadera, el elemento se agrega a la nueva lista; de lo contrario, se omite.

**Código de línea en donde se usa esta función**

nueva\_lista = [valor for valor in lista if valor > segundo\_valor]

La variable **valor** representa cada elemento de la lista original (**lista**). Se evalúa la condición **valor > segundo\_valor**, y si es verdadera, el **valor** se agrega a la nueva lista **nueva\_lista**. En otras palabras, esta línea de código crea una lista que contiene solo los valores mayores que **segundo\_valor** de la lista original.

La variable **segundo\_valor** fue definida previamente en la función como el segundo elemento de la lista (**lista[1]**). Por lo tanto, la comprensión de lista filtra la lista original y crea una nueva lista que contiene los valores mayores que **segundo\_valor**.

**SINTAXIS PARA CREAR NUEVAS LISTAS BAJO CIERTAS CONDICIONES**

La línea **nueva\_lista = [valor] \* tamaño** crea una nueva lista llamada **nueva\_lista** que contiene **tamaño** elementos, y cada elemento tiene el valor **valor**. Por ejemplo, si llamamos a **length\_and\_value(4, 7)**, se creará una lista con 4 elementos, todos con el valor **7**, como **[7, 7, 7, 7]**.

INTRODUCCIÓN A POO

Clases

Siempre que declaramos una variable, estamos creando una instancia de una clase. Por ejemplo, al declarar x = [1,2,3], x es una instancia de una lista. Una instancia es simplemente un objeto que sigue el patrón definido por su clase.

A blue screen with text and symbols

Description automatically generated

Lo que hace

Lo que tiene

* **Atributos:** características compartidas por todas las instancias del tipo de clase.
* **Métodos:** Acciones que puede realizar un objeto. Un usuario, por ejemplo, debería poder hacer un depósito o un retiro, o tal vez enviar dinero a otro usuario.

**Atributos de instancia**Los atributos de instancia se definen en un "método mágico" llamado **\_\_init\_\_,** que se llama cuando se crea una instancia de un nuevo objeto.

El primer parámetro de un método de instancia dentro de una clase será **self,** y los atributos de instancia también se indican por **self.**

**self**es una referencia a la instancia, no a la clase.

**Atributos de clase**

Los atributos de clase se definen fuera de cualquier método de instancia y se comparten entre todas las instancias de la clase.

Los **métodos** solo son funciones que pertenecen a una clase. Esto significa que no podemos llamarlos de forma independiente como hemos llamado funciones anteriormente; más bien, los métodos deben llamarse desde una instancia de una clase. Por ejemplo, si un usuario quisiera hacer un depósito, querríamos poder llamar al método *desde la instancia del usuario*; debido a que un usuario específico está realizando un depósito, solo debería afectar el balance de ese usuario.

**Self**

Probablemente sea hora de hablar sobre **self**. El parámetro **self** incluye toda la información sobre el objeto individual que ha llamado al método. Pero, ¿cómo se transmite? Según la firma del método de depósito o el método \_\_init\_\_, requieren 2 y 3 argumentos, respectivamente. Sin embargo, cuando los llamamos, pasamos solo 1 y 2. ¿Qué está pasando aquí? Debido a que estamos invocando el método *desde la instancia*, esto se conoce como **paso implícito de self**. Cuando invocamos un método desde una instancia, esa instancia, junto con toda su información (nombre, email, balance), se pasa como *self*.

**@classmethod**

Los métodos de clase se definen con un decorador @classmethod. Pertenecen a la propia clase en lugar de a la instancia. En lugar de pasar implícitamente self al método, pasamos cls. Esta es una referencia a la clase.

**@staticmethod**

Los métodos estáticos son funciones definidas dentro de la clase con un decorador @staticmethod. No tienen acceso a atributos de instancia o clase, por lo que sí necesitamos pasarles argumentos.

Si quisiéramos que nuestra clase CuentaBancaria tuviera una función de utilidad para sumar o restar, podríamos implementar @staticmethod en la clase.

Sugerencia: cuando se utilizan valores predeterminados en los parámetros, ¡el orden de los parámetros es importante!).

**PROGRAMACIÓN ORIENTADA A OBJETOS**

¿Qué se logra?

Implementa D.R.Y. Código (No te repitas)

Hace que nuestra aplicación sea escalable.

Hace que nuestro código sea reutilizable.

Hace que nuestras aplicaciones sean fáciles de mantener.

A screen shot of a computer code

Description automatically generated

A screen shot of a computer code

Description automatically generated

A screen shot of a computer code

Description automatically generated

A screen shot of a computer code

Description automatically generated

**Módulos**

Los módulos son simplemente archivos de Python con la extensión .py que implementan un conjunto de funciones. Los módulos se importan mediante el comando import.

La primera vez que se carga un módulo en un script de Python en ejecución, se inicializa ejecutando el código en el módulo una vez. Si otro módulo en tu código importa el mismo módulo nuevamente, no se cargará dos veces, sino solo una vez, por lo que las variables locales dentro del módulo actúan como un "singleton", se inicializan solo una vez.

Ahora, si queremos importar el módulo urllib.request, que nos permite solicitar datos de las URL, podemos importar el módulo de manera simple:

# importar la biblioteca

import urllib.request

response = urllib.request.urlopen("http://www.codingdojo.com")

html = response.read()

print(html)

**Crear tus propios módulos**

Escribir tus propios módulos de Python es muy simple. Para crear un módulo, primero creamos un nuevo archivo .py con el nombre del módulo en el mismo directorio que el archivo que importará el módulo. Luego lo importamos usando el comando import y el nombre del archivo Python (sin la extensión .py). lista de módulos integrados:

<https://docs.python.org/3.6/library/index.html>

**Explorar módulos integrados**

Dos funciones muy importantes son útiles cuando se exploran módulos en Python: las funciones dir y help. Podemos buscar qué funciones se implementan en cada módulo usando la función dir

**Paquetes**

Un módulo es un solo archivo (o archivos) que se importa bajo una sola importación. Un **paquete** es una colección de módulos en directorios que dan una jerarquía de paquetes.

from mi\_paquete.subdirectorio import mis\_funciones

Los paquetes son namespaces que contienen múltiples paquetes y módulos. Son simplemente directorios, pero con una pequeña diferencia.

proyecto\_ejemplo

|\_\_\_\_\_ archivo\_python.py

|\_\_\_\_\_ mis\_módulos

|\_\_\_\_\_ \_\_init\_\_.py

|\_\_\_\_\_ módulo\_prueba.py

|\_\_\_\_\_ otro\_módulo.py

|\_\_\_\_\_ tercer\_módulo.py

copy

En el diagrama anterior, el nombre del paquete es *mis\_módulos*.

**Escribir paquetes**

Si creamos un directorio llamado *mis\_módulos*, que marca el nombre del paquete, podemos crear un módulo dentro de ese paquete llamado *módulo\_prueba*.

Para usar el módulo *módulo\_prueba*, podemos importarlo de dos maneras:

import mis\_módulos.módulo\_prueba

**O**

from mis\_módulos import módulo\_prueba

**Archivo \_\_init\_\_.py**

En Python 3.3+, solo necesitamos este archivo si necesitamos personalizar qué módulos están disponibles para cualquiera que intente importar el paquete. Por ejemplo, si no queremos que otro\_módulo o tercer\_módulo sean accesibles para importar, podríamos anular la variable *\_\_all\_\_* , así:

proyecto\_ejemplo/mis\_módulos/\_\_init\_\_.py

\_\_all\_\_ = ["módulo\_prueba"]

**Anulación**

Hablemos de otras características interesantes de la programación orientada a objetos. A veces, el problema con la herencia implícita es que quieres que el hijo se comporte de manera completamente diferente al padre. En estos casos, quieres *anular* la función, reemplazando efectivamente la funcionalidad. Para hacer esto, simplemente define una función con el mismo nombre en la clase secundaria.

# utilizaremos la clase Persona para demostrar polimorfismo

# en el que varias clases heredan de la misma clase pero se comportan de diferentes maneras

class Persona:

*def* pagar\_cuenta(self):

raise NotImplementedError

# Millonario hereda de Persona

*class* Millonario(*Persona*):

*def* pagar\_cuenta(self):

print("Aquí tienes. Quédate con el cambio.")

# Estudiante de posgrado también hereda de la clase Persona

*class* EstudiantePosgrado(*Persona*):

*def* pagar\_cuenta(self):

print("¿Puedo deberle diez dólares o lavar los platos?")

La clase base Persona tiene un método pagar\_cuenta() que está marcado como NotImplementedError. Esto significa que el método está presente en la clase base pero no se ha implementado, y se espera que las subclases lo implementen de manera específica.

Según este ejemplo, un millonario y un estudiante de posgrado son Personas. Sin embargo, cuando se trata de pagar una cuenta, la forma en que actúan es bastante diferente. Este patrón es útil cuando sabes que cada subclase de una clase principal debe definir un comportamiento específico en un método y no quieres definir un comportamiento predeterminado en la clase principal (de ahí la implementación virtual pura en la principal).

**ENTRADAS Y SALIDAS**

color\_favorito = input('¿Cuál es tu color favorito? ') # la entrada toma un prompt, que debe ser una cadena

print(f'Tu color favorito is: {colo\_favorito}') # salida, imprime el color dado en la consola

ARGUMENTOS MULTIPLES  
¿Qué pasa si queremos pasar un número variable de argumentos o queremos capturar varios argumentos en un solo parámetro? Colocar un asterisco antes del nombre del parámetro después de los parámetros "normales" hace precisamente eso. El asterisco se llama operador splat.

def varargs(arg1, \*args):

print("Tengo ", arg1, " and ", args)

varargs("uno") # salida: Tengo uno y ()

varargs("uno", "dos")         # salida: Tengo uno y ('dos',)

varargs("uno", "dos", "tres")  # salida: Tengo uno y ('dos', 'tres' )

En este ejemplo, el primer argumento arg1 se asigna al primer parámetro del método como de costumbre. Sin embargo, observa que todos y cada uno de los argumentos restantes pasados están en el parámetro args, que parece ser una tupla (como lo indica la sintaxis (). Debido a que anteponemos el parámetro final con un asterisco (el operador splat), todos los argumentos que no coinciden con un parámetro requerido se *empaquetan* en una sola **tupla**.

Recuerda que una tupla es iterable, como una lista. Eso significa que si queremos acceder a cada uno de los argumentos pasados individualmente, podemos usar un bucle:

def varargs(arg1, \*args):

for a in args:

print(a)

varargs("uno", "dos", "tres") # salida: dos, tres (en líneas separadas)

INTRODUCCION A ESTRUCTURAS DE DATOS

Las estructuras de datos son tipos de datos que nos permiten almacenar y administrar múltiples valores. Donde los tipos de datos primitivos son valores simples, como los números o las cadenas, las estructuras de datos nos permiten tener colecciones de valores en una variable. Los arreglos, listas y diccionarios son ejemplos de estructuras de datos.

Hemos estado utilizando estructuras de datos durante algunas semanas, pero ahora que sabemos cómo construir clases con métodos, ¡vamos a profundizar en cómo funcionan bajo el capó! Comenzaremos mirando específicamente a una estructura de datos conocida como lista simple enlazada. La lista es simplemente una clase que tiene métodos; así como la clase lista de Python tiene métodos como append(val) y pop(), vamos a escribir una clase con la misma funcionalidad.