**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.

Listas enlazadas (o vinculadas)

Arreglos

Ya conocemos los arreglos. Los arreglos realizan un seguimiento de valores múltiples y están optimizados para acceso aleatorio, lo que significa que podemos acceder a los valores en el arreglo por número de índice, por ejemplo, miArreglo[3]. Los arreglos son una buena opción para cuando necesitamos mantener los valores en un orden específico o acceder de forma aleatoria a un valor dado en algún lugar intermedio. Esto es posible porque un arreglo ocupa un conjunto completo de celdas consecutivas en la memoria. Sin embargo, los arreglos son menos eficientes cuando tenemos que insertar o eliminar mucho desde el medio, ya que debemos cambiar otros valores para mantener el orden.

Una lista enlazada es otra estructura de datos que almacena valores en orden secuencial, pero está más optimizada para una rápida inserción y eliminación. Una vez que hayas visto cómo funcionan, ¡esto tendrá más sentido!

Listas enlazadas (o vinculadas)

Ingresar listas vinculadas. ¿Qué diferencia hay entre las listas vinculadas y los arreglos? En lugar de almacenar todos los valores de forma consecutiva en la memoria, cada elemento de la colección contiene no solo su valor, sino también un enlace al elemento contiguo. Pero, ¿cómo podemos almacenar estos dos datos en una variable? Si estás pensando en clases, ¡estás en el camino correcto! Para seguir la convención, llamaremos **nodo** a este contenedor.

¿Qué es un nodo?

Un **nodo** es una *clase* que, como se describió anteriormente, tendrá dos atributos:

* **value:** el valor real que se almacenará (por ejemplo, "Alice", "Ben", "Chad", etc.).
* **next:** un enlace al nodo contiguo en la lista. Para una computadora, esta es una referencia o dirección de memoria de ese nodo. Para un desarrollador, este es un nombre de variable que *apunta* o *hace referencia* a otro nodo. Si no hay nada junto a él, **next**se puede establecer en **None**. Debido a que no necesariamente sabremos quién será el vecino de nuestro nodo en el momento de la creación, establezcamos el valor predeterminado en None.

class SLNode:

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

self.value = value

self.next = None

¿Cómo pasamos de un nodo a una lista?

Ahora que tenemos contenedores, o **nodos**, para cada elemento en nuestra lista, necesitamos una forma de *administrarlos* realmente (es decir, agregar nuevos valores a nuestra lista, eliminar valores de la lista, etc.). Si estás pensando que necesitamos otra clase, ¡tienes toda la razón!

Lo bueno de una lista vinculada es que, siempre que conozcamos el primer nodo, podemos llegar a todos los demás nodos comenzando por el frente y moviéndonos al nodo vecino siempre que haya uno. Pero nos estamos adelantando, ¡eso viene muy pronto!

Convencionalmente, este primer nodo se conoce como el **encabezado (head)** de la lista. Cuando creamos una lista por primera vez, estará vacía:

class SList:

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

self.head = None

Ahora que tenemos nuestras clases, hagamos *una instancia* de nuestra lista:

my\_list = SList()

Esto puede parecer bastante abstracto, pero recuerda que los principios siguen siendo los mismos que cuando creamos una clase User y CuentaBancaria. Es solo que el objetivo de nuestra clase es diferente: no estamos tratando de representar a un usuario, sino a una estructura de datos llamada **Lista simple enlazada**. En la siguiente tarea, agregaremos algunos métodos a nuestra clase SList.

Agregar un valor al comienzo

Una funcionalidad común de una lista es poder agregar valores, así que agreguemos dicho método a nuestra clase.

Así como pasaríamos un valor al método de adición de una lista de Python (append), nuestro método add\_to\_front debería aceptar un valor para agregarlo a la lista:

    class SList:

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

        self.head = None

*def* add\_to\_front(self, val): # agregó esta línea, toma un valorcopy

1. Lo primero que debe hacer nuestro método es crear un nodo:

*def* add\_to\_front(self, val):

new\_node = SLNode(val) # crea una nueva instancia de nuestra clase Node usando el valor dadocopy

Esto implica que ya hemos creado la clase Node. Hicimos esto en la pestaña anterior, pero como recordatorio, en una clase separada (el mismo archivo está bien):

    class SLNode:

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

        self.value = val

    self.next = Nonecopy

1. Queremos que este nuevo nodo sea el primero de nuestra lista, pero no nos apresuremos. Dado que el único nodo al que tenemos una referencia en nuestra lista es el encabezado, si lo reemplazamos de inmediato, perderemos nuestra referencia al encabezado actual, así que guardémoslo antes de seguir adelante:
2. *def* add\_to\_front(self, val):
3. new\_node = SLNode(val)

    current\_head = self.head # guarda el encabezado actual en una variablecopy

1. Actualmente nuestro nuevo nodo no tiene vecino. Debido a que estamos tratando de agregar este nuevo nodo al frente, ahora sabemos que su vecino debería ser el encabezado actual que acabamos de guardar:
2. *def* add\_to\_front(self, val):
3. new\_node = SLNode(val)
4. current\_head = self.head

    new\_node.next = current\_head # ESTABLECE el nuevo nodo JUNTO AL encabezado actual de la listacopy

1. Finalmente, necesitamos que este nuevo nodo sea el encabezado de nuestra lista:
2. *def* add\_to\_front(self, val):
3. new\_node = SLNode(val)
4. current\_head = self.head
5. new\_node.next = current\_head
6. self.head = new\_node # FIJA el encabezado de la lista AL nodo que creamos en el último paso

return self                 # return self para permitir el encadenamientocopy

Tómate un momento para comparar esta funcionalidad a agregar un valor al comienzo de un arreglo. Con suerte, podrás ver algunos de los beneficios: no es necesario desplazar/mover, ¡sin importar cuántos elementos tengamos en nuestra lista!

Recorrer una lista

Dado que esto es bastante abstracto, podría ser útil tener una función que pase por cada nodo e imprima su valor. Esta es una gran oportunidad para aprender a *recorrer una lista vinculada*. Para iterar a través de un arreglo, usamos un bucle for con un índice como iterador. Sin embargo, dado que nuestra lista vinculada no está indexada, tenemos que crear un iterador diferente. Usaremos un **puntero** que, en un bucle, apuntará a cada nodo.

1. Este método no requerirá ninguna entrada:

*def* print\_values(self):copy

1. Necesitamos comenzar al principio de nuestra lista, así que creemos un puntero a nuestro primer nodo:
2. *def* print\_values(self):

runner = self.head # un puntero al primer nodo de la listacopy

1. Siempre que la variable runner apunte a un nodo:
2. *def* print\_values(self):
3. runner = self.head
4. while (runner != None): # iterando mientras runner es un nodo y no None

</div>copy

1. Imprimamos su valor:
2. *def* print\_values(self):
3. runner = self.head
4. while (runner != None):

        print(runner.value) # imprime el valor del nodo actualcopy

1. Luego, necesitamos "incrementar" runner al siguiente nodo, o actualizarlo para que apunte a su vecino:
2. *def* print\_values(self):
3. runner = self.head
4. while (runner != None):
5. print(runner.value)
6. runner = runner.next # establece runner a su vecino

    return self                 # una vez terminado el ciclo, devuelve self para permitir el encadenamientocopy

Recorrer una lista y agregar un valor al final

Practiquemos recorrer listas una vez más. Si queremos agregar un nuevo nodo en cualquier lugar de nuestra lista, solo necesita convertirse en vecino de un nodo existente. Para convertirse en el último nodo de nuestra lista, el último nodo actual de la lista debe apuntar hacia este nuevo nodo.

1. Este método requerirá que se agregue un valor:

*def* add\_to\_back(self, val): # acepta un valorcopy

1. Luego querremos crear un nuevo nodo con el valor dado:
2. *def* add\_to\_back(self, val):

    new\_node = SLNode(val) # crea una nueva instancia de nuestra clase Node con el valor dado</span></copy

1. Inicia un iterador al principio de la lista
2. *def* add\_to\_back(self, val):
3. new\_node = SLNode(val)

    runner = self.head     # establecer un iterador para que comience al principio de la listacopy

1. Como queremos llegar al último nodo, queremos detenernos en el nodo que no tiene vecino:
2. *def* add\_to\_back(self, val):
3. new\_node = SLNode(val)
4. runner = self.head

    while (runner.next != None): # iterador hasta que el iterador no tenga un vecinocopy

1. Incrementa runner a su vecino (ya que acabamos de verificar para asegurarnos de que, de hecho, hay un vecino):
2. *def* add\_to\_back(self, val):
3. new\_node = SLNode(val)
4. runner = self.head
5. while (runner.next != None):

        runner = runner.next # incrementa runner al siguiente nodo de la listacopy

1. Cuando el bucle haya terminado de ejecutarse, runner apuntará al último nodo. Su next actualmente está configurado en None, pero queremos que el nuevo nodo que creamos al comienzo de este método sea su vecino:
2. *def* add\_to\_back(self, val):
3. new\_node = SLNode(val)
4. runner = self.head
5. while (runner.next != None):
6. runner = runner.next

    runner.next = new\_node # incrementa runner al siguiente nodo de la listacopy

1. Considera el caso extremo en el que nuestra lista está vacía. En ese caso, agregar al comienzo sería lo mismo que agregar al final. Como ya hemos escrito ese método, ¡usémoslo!
2. *def* add\_to\_back(self, val):
3. if self.head == None: # si la lista está vacía
4. self.add\_to\_front(val) # ejecuta el método add\_to\_front
5. return self # asegurémonos de que el resto de esta función no suceda si agregamos al comienzo
6. new\_node = SLNode(val)
7. runner = self.head
8. while (runner.next != None):
9. runner = runner.next
10. runner.next = new\_node # incrementa runner al siguiente nodo de la lista

            return self                 # return self para permitir el encadenamientocopy

¡Probemos nuestra clase!

my\_list = SList() # crea una nueva instancia de una lista

my\_list.add\_to\_front("son").add\_to\_front("Las listas enlazadas").add\_to\_back("divertidas!").print\_values()    # encadenando, yeah!

# la salida debería ser:

# Las listas enlazadas

# son

# divertidas!

copy

*Si te sientes desanimado, confundido o abrumado, no te preocupes. Te garantizamos que no eres el único. Este es un concepto realmente difícil de aprender la primera vez. Sigue practicando y dividiendo cada paso una línea a la vez. Trata de averiguar qué partes no tienen sentido y luego habla con un compañero de clase, asistente técnico o instructor.*

Una vez que tengas una buena comprensión de esta idea de nodos con punteros, tendrás los componentes básicos para construir algunas otras estructuras de datos realmente geniales como árboles binarios de búsqueda, tries, grafos y más. Como es un concepto tan importante, practica y revisa el código anterior para que puedas volver a escribir el código demostrado arriba sin mirar la plataforma. Asegúrate de sentirte muy cómodo al agregar un nuevo nodo, recorrer la lista vinculada, etc. Una vez que puedas crear SList y Node sin mirar los códigos anteriores, pasa a alguno de los desafíos adicionales.

Desafíos adicionales

¡Estos son un desafío! Súbete a una pizarra, busca un compañero si hay alguno disponible, y traten de resolverlos juntos.

1. remove\_from\_front(self): elimina el primer nodo y devuelve su valor
2. remove\_from\_back(self): elimina el último nodo y devuelve su valor
3. remove\_val(self, val): elimina el primer nodo con el valor dado

Considera los siguientes casos:

* + el nodo con el valor dado es el primer nodo
  + el nodo con el valor dado está en el medio de la lista
  + el nodo con el valor dado es el último nodo

1. insert\_at(self, val, n): inserta un nodo con valor *val* como el *n*-ésimo nodo en la lista

Considera los siguientes casos:

* + *n* es 0
  + *n* es la longitud de la lista
  + *n* está entre 0 y la longitud de la lista

Otro tipo de función mencionada anteriormente es la función *anónima*. En pocas palabras, una función anónima es una función sin nombre. En Python, las funciones anónimas se crean con la palabra clave **lambda**. Estas funciones se utilizan para diversos fines:

* son útiles en situaciones en las que solo necesitamos usar la función una vez
* Son livianas cuando necesitamos dividir tareas complejas en tareas pequeñas y específicas
* son convenientes como argumentos para funciones que requieren funciones como parámetros

Anteriormente, definimos la función square() que toma un parámetro (num), lo eleva al cuadrado y luego lo devuelve:

*def* square(num):

x = num \*\* 2

return x

Ahora podemos reescribir esta función como una función lambda anónima (sin nombre):

lambda num: num \*\* 2

Esto significa *"aquí hay una función anónima (sin nombre) que toma un argumento, llamado num, y devuelve num \*\* 2.*

¿Qué pasa si quisieras crear una función anónima (sin nombre) que toma dos argumentos y devuelve la suma de los dos argumentos?

lambda num1, num2: num1+num2copy

Esto significa *"aquí hay una función anónima (sin nombre) que toma dos argumentos: num1 y num2; y devuelve num 1+num2*

Una lambda podría ser:

**un elemento en una lista:**

# crea una nueva lista, con una lambda como elemento

my\_list = ['test\_string', 99, lambda x : x \*\* 2]

# acceder al valor en la lista

print(my\_list[2]) # imprimirá un objeto lambda almacenado en la memoria

# invocar la función lambda, pasando 5 como argumento

my\_list[2](5)

**pasada a otra función como devolución de llamada:**

# definir una función que tome una entrada que sea una función

*def* invoker(callback):

# invocar la entrada pasar el argumento 2

print(callback(2))

invoker(lambda x: 2 \* x)

invoker(lambda y: 5 + y)

**almacenada en una variable:**

add10 = lambda x: x + 10 # almacenar la expresión lambda en una variable

add10(2) # devuelve 12

add10(98) # devuelve 108

**devuelta por otra función:**

*def* incrementor(num):

start = num

return lambda x: num + x

Lambda y otras funciones

A veces, cuando invocas una función, es posible que uno de los argumentos deba ser una función. Aquí es donde las lambdas son útiles, un lugar donde es posible que no quieras declarar una función completamente nueva con la palabra clave def. Por ejemplo, hablemos de la función map que toma una función y una lista como parámetros, luego aplica la función a cada elemento de la lista y devuelve un objeto map, que se puede convertir fácilmente en una lista. Una forma en la que podríamos hacer esto es así:

# crear una lista

my\_arr = [1,2,3,4,5]

# define una función que eleve al cuadrado los valores

*def* square(num):

return num \*\* 2

# invocar función map

print(list(map(square, my\_arr)))

La salida es:

[1,4,9,16,25]

De esta manera está bien y funciona, pero digamos que nunca volvemos a usar la función *square* en nuestro programa. Definir una función simplemente para ese propósito no es necesario, y es por eso que las *lambdas son útiles* ***cuando solo necesitamos una función una vez***. De esta manera no necesitamos definir una función y consumir memoria innecesariamente y complicar nuestro código, solo para producir el mismo resultado:

my\_arr = [1,2,3,4,5]

print(list(map(lambda x: x \*\* 2, my\_arr))) # invocar map, pasa un lambda como primer argumento

Funciones donde entra en juego esta implementación:

* map()
* reduce()
* sort(): lambda es opcional
* filter()

## ¿Qué son las secuencias?

Las **secuencias**son cualquier cosa sobre la que podamos iterar secuencialmente, incluidas listas, tuplas y cadenas. Aquí aprenderemos sobre algunas cosas interesantes que podemos hacer con instancias de estas clases.

## Rebanado

Si estamos interesados en obtener subconjuntos de valores de una secuencia, Python proporciona una forma de rebanar con [:]. Indicamos el índice inicial a la izquierda y el índice final (exclusivo) a la derecha. Si no indicamos un valor a la izquierda, comenzará en el índice 0; si no se especifica el valor de la derecha, asumirá la longitud de la secuencia. Al igual que el método de rebanada en JavaScript, el uso de esta sintaxis devuelve una copia del tipo de datos con los valores especificados.

my\_list = [99,4,2,5,-3]

my\_tuple = (99,4,2,5,-3)

my\_str = "sequoia"

print(my\_list[:])

# output: [99,4,2,5,-3]

print(my\_tuple[1:])

# output: (4,2,5,-3)

print(my\_str[:3])

# output: "seq"

print(my\_tuple[2:4])

# output: (2,5)

print(my\_list, my\_tuple, my\_str)

# output: [99,4,2,5,-3] (99,4,2,5,-3) 'sequoia' -- nota que los valores originales no han cambiado

Otros métodos de secuencia integrados

A continuación, se muestran algunas funciones integradas de uso común para secuencias:

* max(sequence) devuelve el valor más grande en la secuencia
* sum(sequence) devuelve la suma de todos los valores en la secuencia
* map(función, sequence) aplica la función a cada elemento de la secuencia que pasas. Devuelve una lista de los resultados.
* min(sequence) devuelve el valor más bajo de una secuencia.
* sorted(sequence) devuelve una lista ordenada de los valores de una secuencia

Haz clic <https://www.linuxtopia.org/online_books/programming_books/python_programming/python_ch14s07.html> para obtener más información sobre otras funciones integradas que puedes usar con secuencias.

<https://realpython.com/face-recognition-with-python/>

<https://realpython.com/face-detection-in-python-using-a-webcam/>

El marco de pruebas unitarias de Python, que antes se conocía como *PyUnit*, se incluye en la biblioteca estándar de Python. Comencemos con una "unidad" simple y sus pruebas. Crea un archivo nuevo:

test\_even.py

# importar el marco de prueba de Python

import unittest

# nuestra "unidad"

# esto es en lo que estamos ejecutando nuestra prueba

*def* isEven(n):

if n % 2 == 0:

return True

else:

return False

# nuestras "pruebas unitarias"

# inicializado creando una clase que hereda de unittest.TestCase

class IsEvenTests(unittest.TestCase):

# cada método de esta clase es una prueba que se ejecutará

*def* testTwo(self):

self.assertEqual(isEven(2), True)

        # otra forma de escribir lo de arriba es

self.assertTrue(isEven(2))

*def* testThree(self):

self.assertEqual(isEven(3), False)

# otra forma de escribir lo de arriba es

self.assertFalse(isEven(3))

# cualquier tarea que quieras ejecutar antes de que se ejecute cualquier método anterior, ponlas en el método setUp

*def* setUp(self):

# agregar las tareas setUp

print("running setUp")

# cualquier tarea que quieras ejecutar después de que se ejecuten las pruebas, ponlas en el método tearDown

*def* tearDown(self):

     # agrega las tareas de tearDown

print("running tearDown tasks")

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main() # esto ejecuta nuestras pruebas

En el ejemplo anterior usamos assertTrue() y assertFalse(), que introducen el primer par de muchas funciones/aserciones disponibles en el marco unittest. Tocaremos algunos de estos a medida que avancemos, pero aquí hay una buena referencia: [funciones unittest](https://docs.python.org/3.6/library/unittest.html#unittest.TestCase).

Ejecutar unittest

Antes que nada, examinemos cómo ejecutamos nuestras pruebas:

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

Al incluir estas dos líneas, podemos ejecutar nuestro código de prueba ejecutando nuestro archivo Python. Ejecutarlo sin opciones da como resultado una salida simple, pero ejecutar este archivo con el indicador -v te dará un resultado detallado con información sobre cada prueba que se ejecutó:

Resultados de la prueba

Cuando ejecutamos nuestras pruebas, hay 3 resultados posibles:

1. OK: todas las pruebas han pasado
2. FAIL: una o más de las pruebas han fallado y genera una excepción AssertionError
3. ERROR: la prueba genera una excepción distinta de AssertionError

En el flujo de trabajo típico de TDD:

Piensa en Característica-> Escribir pruebas-> Ejecutar y fallar-> Código-> Ejecutar y aprobar-> Refactorizar-> Repetir

Otras aserciones útiles

Si entendiste el código anterior, implementar las otras aserciones es bastante sencillo. Aquí hay una lista de algunas de las otras aserciones útiles directamente del documento de Python:

A screenshot of a computer

Description automatically generated