Módulo 3 (Intermedio): Programación Orientada a Objetos - Un viaje por la OOP

¿Qué es una pila?

Una pila es una estructura desarrollada para almacenar datos de una manera muy específica. Imagina una pila de monedas. No puedes poner una moneda en ningún otro lugar sino en la parte superior de la pila.

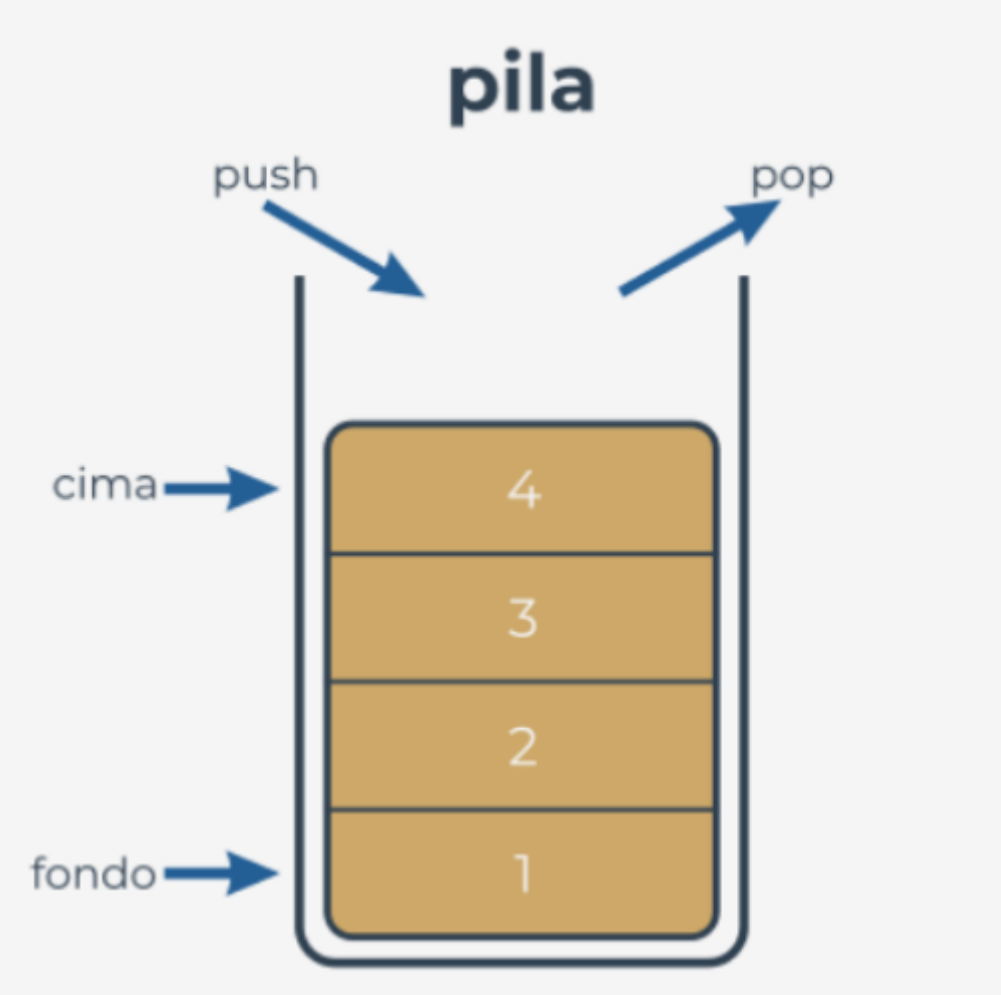
Del mismo modo, no puedes sacar una moneda de la pila desde ningún lugar que no sea la parte superior de la pila. Si deseas obtener la moneda que se encuentra en la parte inferior, debes eliminar todas las monedas de los niveles superiores.

El nombre alternativo para una pila (pero solo en la terminología de TI) es **UEPS (LIFO son sus siglas en inglés)**.

Es una abreviatura para una descripción muy clara del comportamiento de la pila: **Último en Entrar - Primero en Salir (Last In - First Out)**. La moneda que quedó en último lugar en la pila saldrá primero.

Una pila es un objeto con dos operaciones elementales, denominadas convencionalmente **push** (cuando un nuevo elemento se coloca en la parte superior) y **pop** (cuando un elemento existente se retira de la parte superior).

Las pilas se usan muy a menudo en muchos algoritmos clásicos, y es difícil imaginar la implementación de muchas herramientas ampliamente utilizadas sin el uso de pilas.



Implementemos una pila en Python. Esta será una pila muy simple, y te mostraremos como hacerlo en dos enfoques independientes: de manera procedimental y orientado a objetos.

La pila: el enfoque procedimental

Primero, debes decidir como almacenar los valores que llegarán a la pila. Sugerimos utilizar el método más simple, y **emplear una lista** para esta tarea. Supongamos que el tamaño de la pila no está limitado de ninguna manera. Supongamos también que el último elemento de la lista almacena el elemento superior.

La pila en sí ya está creada:

```
stack = []
```

Estamos listos para **definir una función que coloca un valor en la pila**. Aquí están las presuposiciones para ello:

- El nombre para la función es push.
- La función obtiene un parámetro (este es el valor que se debe colocar en la pila).
- La función no retorna nada.
- La función agrega el valor del parámetro al final de la pila.

Así es como lo hemos hecho, echa un vistazo:

```
def push(val):
```

```
    stack.append(val)
```

Ahora es tiempo de que una **función quite un valor de la pila**. Así es como puedes hacerlo:

- El nombre de la función es pop.
- La función no obtiene ningún parámetro.
- La función devuelve el valor tomado de la pila.
- La función lee el valor de la parte superior de la pila y lo elimina.

La función esta aqui:

```
def pop():
    val = stack[-1]
    del stack[-1]
    return val
```

Nota: la función no verifica si hay algún elemento en la pila.

Armemos todas las piezas juntas para poner la pila en movimiento. El programa completo empuja (push) tres números a la pila, los saca e imprime sus valores en pantalla.

```
stack = []


def push(val):
    stack.append(val)


def pop():
    val = stack[-1]
    del stack[-1]
    return val


push(3)
push(2)
push(1)

print(pop())
print(pop())
print(pop())
```

El programa muestra el siguiente texto en pantalla:

```
1
2
3
```

La pila: el enfoque procedimental frente al enfoque orientado a objetos

La pila procedimental está lista. Por supuesto, hay algunas debilidades, y la implementación podría mejorarse de muchas maneras (aprovechar las excepciones es una buena idea), pero en general la pila está completamente implementada, y puedes usarla si lo necesitas.

Pero cuanto más la uses, más desventajas encontrarás. Éstas son algunas de ellas:

- La variable esencial (la lista de la pila) es altamente vulnerable; cualquiera puede modificarla de forma incontrolable, destruyendo la pila; esto no significa que se haya hecho de manera maliciosa; por el contrario, puede ocurrir como resultado de un descuido, por ejemplo, cuando alguien confunde nombres de variables; imagina que accidentalmente has escrito algo como esto:

  ```
  stack[0] = 0
  ```

  El funcionamiento de la pila estará completamente desorganizado.

- También puede suceder que un día necesites más de una pila; tendrás que crear otra lista para el almacenamiento de la pila, y probablemente otras funciones push y pop.
- También puede suceder que no solo necesites funciones push y pop, pero también algunas otras funciones; ciertamente podrías implementarlas, pero intenta imaginar qué sucedería si tuvieras docenas de pilas implementadas por separado.

El enfoque orientado a objetos ofrece soluciones para cada uno de los problemas anteriores. Vamos a nombrarlos primero:

- La capacidad de ocultar (proteger) los valores seleccionados contra el acceso no autorizado se llama **encapsulamiento; no se puede acceder a los valores encapsulados ni modificarlos si deseas utilizarlos exclusivamente**.
- Cuando tienes una clase que implementa todos los comportamientos de pila necesarios, puedes producir tantas pilas como desees; no necesitas copiar ni replicar ninguna parte del código.
- La capacidad de enriquecer la pila con nuevas funciones proviene de la herencia; puedes crear una nueva clase (una subclase) que herede todos los rasgos existentes de la superclase y agregar algunos nuevos.

Ahora escribamos una nueva implementación de pila desde cero. Esta vez, utilizaremos el enfoque orientado a objetos, que te guiará paso a paso en el mundo de la programación de objetos.

La pila, el enfoque orientado a objetos

Por supuesto, la idea principal sigue siendo la misma. Usaremos una lista como almacenamiento de la pila. Solo tenemos que saber como poner la lista en la clase.

Comencemos desde el principio: así es como comienza la pila orientada a objetos:

```
class Stack:
```

Ahora, esperamos dos cosas de la clase:

- Queremos que la clase tenga **una propiedad como el almacenamiento de la pila**, tenemos que «**instalar» una lista dentro de cada objeto de la clase** (nota: cada objeto debe tener su propia lista; la lista no debe compartirse entre diferentes pilas).
- Despues, queremos que **la lista esté oculta** de la vista de los usuarios de la clase.

¿Cómo se hace esto?

A diferencia de otros lenguajes de programación, Python no tiene medios para permitirte declarar una propiedad como esa.

En su lugar, debes agregar una instrucción específica. Las propiedades deben agregarse a la clase manualmente.

¿Cómo garantizar que dicha actividad tiene lugar cada vez que se crea una nueva pila?

Existe una manera simple de hacerlo, tienes que **equipar a la clase con una función específica**:

- Tiene que ser nombrada de forma estricta.
- Se invoca implícitamente cuando se crea el nuevo objeto.

Dicha función es llamada el **constructor**, ya que su propósito general es **construir un nuevo objeto**. El constructor debe saber todo acerca de la estructura del objeto y debe realizar todas las inicializaciones necesarias.

Agreguemos un constructor muy simple a la nueva clase. Echa un vistazo al código:

```
class Stack:
    def __init__(self):
        print("¡Hola!")


stack_object = Stack()
```

Expliquemos más a detalle:

- El nombre del constructor es siempre __init__.
- Tiene que tener **al menos un parámetro** (discutiremos esto más adelante); el parámetro se usa para representar el objeto recién creado: puedes usar el parámetro para manipular el objeto y enriquecerlo con las propiedades necesarias; harás uso de esto pronto.
- Nota: el parámetro obligatorio generalmente se denomina *self*, es solo **una sugerencia, pero deberías seguirla**, simplifica el proceso de lectura y comprensión de tu código.

```
class Stack:  # Definiendo la clase de la pila.
    def __init__(self):  # Definiendo la función del constructor.
        print("¡Hola!")


stack_object = Stack()  # Instanciando el objeto.
```

Aquí está su salida:

```
¡Hola!
```

Nota: no hay rastro de la invocación del constructor dentro del código. Ha sido invocado implícita y automáticamente. Hagamos uso de eso ahora.

Cualquier cambio que realices dentro del constructor que modifique el estado del parámetro *self* se verá reflejado en el objeto recien creado.

Esto significa que puedes agregar cualquier propiedad al objeto y la propiedad permanecerá allí hasta que el objeto termine su vida o la propiedad se elimine explícitamente.

Ahora **agreguemos solo una propiedad al nuevo objeto**, una lista para la pila. La nombraremos stack_list.

```
class Stack:
    def __init__(self):
        self.stack_list = []


stack_object = Stack()
print(len(stack_object.stack_list))
```

Nota:

- Hemos usado la **notación punteada**, al igual que cuando se invocan métodos. Esta es la manera general para acceder a las propiedades de un objeto: debes nombrar el objeto, poner un punto (.) después de el, y especificar el nombre de la propiedad deseada, ¡no uses paréntesis! No deseas invocar un método, deseas **acceder a una propiedad**.
- Si estableces el valor de una propiedad por primera vez (como en el constructor), lo estás creando; a partir de ese momento, el objeto tiene la propiedad y está listo para usar su valor.
- Hemos hecho algo más en el código: hemos intentado acceder a la propiedad stack_list desde fuera de la clase inmediatamente después de que se haya creado el objeto; queremos verificar la longitud actual de la pila, ¿lo hemos logrado?

Si, por supuesto: el código produce el siguiente resultado:

```
0
```

Esto no es lo que queremos de la pila. Nosotros queremos que stack_list este escondida del mundo exterior. ¿Es eso posible?

Si, y es simple, pero no muy intuitivo.

Echa un vistazo: hemos agregado dos guiones bajos antes del nombre stack_list, nada mas:

```
class Stack:
    def __init__(self):
        self.__stack_list = []

stack_object = Stack()
print(len(stack_object.__stack_list))
```

El cambio invalida el programa.

¿Por qué?

Cuando cualquier componente de la clase tiene un **nombre que comienza con dos guiones bajos (__), se vuelve privado**, esto significa que solo se puede acceder desde dentro de la clase.

No puedes verlo desde el mundo exterior. Así es como Python implementa el concepto de **encapsulación**.

Ejecuta el programa para probar nuestras suposiciones: una excepción *AttributeError* debe ser generada.

El enfoque orientado a objetos: una pila desde cero

Ahora es el momento de que las dos funciones (métodos) implementen las operaciones push y pop. Python supone que una función de este tipo debería estar **inmersa dentro del cuerpo de la clase**, como el constructor.

Queremos invocar estas funciones para agregar(push) y quitar(pop) valores de la pila. Esto significa que ambos deben ser accesibles para el usuario de la clase (en contraste con la lista previamente construida, que está oculta para los usuarios de la clase ordinaria).

Tal componente es llamado **público**, por ello **no puede comenzar su nombre con dos (o más) guiones bajos**. Hay un requisito más el nombre **no debe tener más de un guión bajo**.

Las funciones en sí son simples. Echa un vistazo:

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


stack_object = Stack()

stack_object.push(3)
stack_object.push(2)
stack_object.push(1)

print(stack_object.pop())
print(stack_object.pop())
print(stack_object.pop())
```

Sin embargo, hay algo realmente extraño en el código. Las funciones parecen familiares, pero tienen más parámetros que sus contrapartes procedimentales.

Aquí, ambas funciones tienen un parámetro llamado **self** en la primera posición de la lista de parámetros.

¿Es necesario? Si, lo es.

Todos los métodos deben tener este parámetro. Desempeña el mismo papel que el primer parámetro constructor.

Permite que el método acceda a entidades (propiedades y actividades / métodos) del objeto. No puedes omitirlo. Cada vez que Python invoca un método, envía implícitamente el objeto actual como el primer argumento.

Esto significa que el **método está obligado a tener al menos un parámetro, que Python mismo utiliza**, no tienes ninguna influencia sobre el.

Si tu método no necesita ningún parámetro, este debe especificarse de todos modos. Si está diseñado para procesar solo un parámetro, debes especificar dos, ya que la función del primero sigue siendo la misma.

Hay una cosa más que requiere explicación: la forma en que se invocan los métodos desde la variable __stack_list.

Afortunadamente, es mucho más simple de lo que parece:

- La primera etapa entrega el objeto como un todo → self.
- A continuación, debes llegar a la lista __stack_list → self.__stack_list.
- Con __stack_list lista para ser usada, puedes realizar el tercer y último paso → self.__stack_list.append(val).

La declaración de la clase está completa y se han enumerado todos sus componentes. La clase está lista para usarse.

Tener tal clase abre nuevas posibilidades. Por ejemplo, ahora puedes hacer que más de una pila se comporte de la misma manera. Cada pila tendrá su propia copia de datos privados, pero utilizará el mismo conjunto de métodos.

Esto es exactamente lo que queremos para este ejemplo.

Analiza el código:

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


stack_object_1 = Stack()
stack_object_2 = Stack()

stack_object_1.push(3)
stack_object_2.push(stack_object_1.pop())

print(stack_object_2.pop())
```

Existen **dos pilas creadas a partir de la misma clase base**. Trabajan independientemente. Puedes crear más si quieres.

Ejecuta el código en el editor y observa que sucede. Realiza tus propios experimentos.

Analiza el fragmento de código a continuación: hemos creado tres objetos de la clase Stack. Después, hemos hecho malabarismos. Intenta predecir el valor que se muestra en la pantalla.

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


little_stack = Stack()
another_stack = Stack()
funny_stack = Stack()

little_stack.push(1)
another_stack.push(little_stack.pop() + 1)
funny_stack.push(another_stack.pop() - 2)

print(funny_stack.pop())
```

Ahora vamos un poco mas lejos. Vamos a **agregar una nueva clase para manejar pilas**.

La nueva clase debería poder **evaluar la suma de todos los elementos almacenados actualmente en la pila**.

No queremos modificar la pila previamente definida. Ya es lo suficientemente buena en sus aplicaciones, y no queremos que cambie de ninguna manera. Queremos una nueva pila con nuevas capacidades. En otras palabras, queremos construir una subclase de la ya existente clase Stack.

El primer paso es fácil: **solo define una nueva subclase que apunte a la clase que se usará como superclase**.

Así es como se ve:

```
class AddingStack(Stack):
    pass
```

La clase aún no define ningún componente nuevo, pero eso no significa que esté vacía. **Obtiene (hereda) todos los componentes definidos por su superclase**, el nombre de la superclase se escribe después de los dos puntos, después del nombre de la nueva clase.

Esto es lo que queremos de la nueva pila:

- Queremos que el método push no solo inserte el valor en la pila, sino que también sume el valor a la variable sum.
- Queremos que la función pop no solo extraiga el valor de la pila, sino que también reste el valor de la variable sum.

En primer lugar, agreguemos una nueva variable a la clase. Será una **variable privada**, al igual que la lista de pila. No queremos que nadie manipule el valor de la variable sum.

Como ya sabes, el constructor agrega una nueva propiedad a la clase. Ya sabes como hacerlo, pero hay algo realmente intrigante dentro del constructor. Echa un vistazo:

```
class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0
```

La segunda línea del cuerpo del constructor crea una propiedad llamada **__sum**, almacenará el total de todos los valores de la pila.

Pero la línea anterior se ve diferente. ¿Qué hace? ¿Es realmente necesaria? Sí lo es.

Al contrario de muchos otros lenguajes, Python te obliga a **invocar explícitamente el constructor de una superclase**. Omitir este punto tendrá efectos nocivos: el objeto se verá privado de la lista __stack_list. Tal pila no funcionará correctamente.

Esta es la única vez que puedes invocar a cualquiera de los constructores disponibles explícitamente; se puede hacer dentro del constructor de la superclase.

Ten en cuenta la sintaxis:

- Se especifica el nombre de la superclase (esta es la clase cuyo constructor se desea ejecutar).
- Se pone un punto (.) después del nombre.
- Se especifica el nombre del constructor.
- Se debe señalar al objeto (la instancia de la clase) que debe ser inicializado por el constructor; es por eso que se debe especificar el argumento y utilizar la variable self aquí; recuerda: **invocar cualquier método (incluidos los constructores) desde fuera de la clase nunca requiere colocar el argumento self en la lista de argumentos**, invocar un método desde dentro de la clase exige el uso explícito del argumento self, y tiene que ser el primero en la lista.

Nota: generalmente es una práctica recomendada invocar al constructor de la superclase antes de cualquier otra inicialización que desees realizar dentro de la subclase. Esta es la regla que hemos seguido en el código.

En segundo lugar, agreguemos dos métodos. Pero, ¿realmente estamos agregándolos? Ya tenemos estos métodos en la superclase. ¿Podemos hacer algo así?

Si podemos. Significa que vamos a **cambiar la funcionalidad de los métodos**, no sus nombres. Podemos decir con mayor precisión que la interfaz (la forma en que se manejan los objetos) de la clase permanece igual al cambiar la implementación al mismo tiempo.

Comencemos con la implementación de la función push. Esto es lo que esperamos de la función:

- Agregar el valor a la variable __sum.
- Agregar el valor a la pila.

Nota: la segunda actividad ya se implementó dentro de la superclase, por lo que podemos usarla. Además, tenemos que usarla, ya que no hay otra forma de acceder a la variable __stackList.

Así es como se mira el método push dentro de la subclase:

```
def push(self, val):
    self.__sum += val
    Stack.push(self, val)
```

Toma en cuenta la forma en que hemos invocado la implementación anterior del método push (el disponible en la superclase):

- Tenemos que especificar el nombre de la superclase; esto es necesario para indicar claramente la clase

que contiene el método, para evitar confundirlo con cualquier otra función del mismo nombre.
- Tenemos que especificar el objeto de destino y pasarlo como primer argumento (no se agrega implícitamente a la invocación en este contexto).

Se dice que el método push ha sido anulado, el mismo nombre que en la superclase ahora representa una funcionalidad diferente.

Esta es la nueva función pop:

```
def pop(self):
    val = Stack.pop(self)
    self.__sum -= val
    return val
```

Hasta ahora, hemos definido la variable __sum, pero no hemos proporcionado un método para obtener su valor. Parece estar escondido. ¿Cómo podemos mostrarlo y que al mismo tiempo que se proteja de modificaciones?

Tenemos que definir un nuevo método. Lo nombraremos **get_sum**. **Su única tarea será devolver el valor de __sum**.

Aquí está:

```
def get_sum(self):
    return self.__sum
```

Entonces, veamos el programa en el editor. El código completo de la clase está ahí. Podemos ahora verificar su funcionamiento, y lo hacemos con la ayuda de unas pocas líneas de código adicionales.

Como puedes ver, agregamos cinco valores subsiguientes en la pila, imprimimos su suma y los sacamos todos de la pila.

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0

    def get_sum(self):
        return self.__sum

    def push(self, val):
        self.__sum += val
        Stack.push(self, val)

    def pop(self):
```

```
        val = Stack.pop(self)
        self.__sum -= val
        return val


stack_object = AddingStack()

for i in range(5):
    stack_object.push(i)
print(stack_object.get_sum())

for i in range(5):
    print(stack_object.pop())
```

Puntos Clave

1. Una **pila** es un objeto diseñado para almacenar datos utilizando el modelo **LIFO**. La pila normalmente realiza al menos dos operaciones, llamadas **push()** y **pop()**.

2. La implementación de la pila en un modelo procedimental plantea varios problemas que pueden resolverse con las técnicas ofrecidas por la **POO** (**P**rogramación **O**rientada a **O**bjetos).

3. Un **método** de clase es en realidad una función declarada dentro de la clase y capaz de acceder a todos los componentes de la clase.

4. La parte de la clase en Python responsable de crear nuevos objetos se llama **constructor** y se implementa como un método de nombre __init__.

5. Cada declaración de método de clase debe contener al menos un parámetro (siempre el primero) generalmente denominado self, y es utilizado por los objetos para identificarse a sí mismos.

6. Si queremos ocultar alguno de los componentes de una clase del mundo exterior, debemos comenzar su nombre con __. Estos componentes se denominan **privados**.

ejercicio

Objetivos

- Mejorar las habilidades del estudiante para definir clases.
- Emplear clases existentes para crear nuevas clases equipadas con nuevas funcionalidades.

Escenario

Recientemente te mostramos cómo extender las posibilidades de Stack definiendo una nueva clase (es decir, una subclase) que retiene todos los rasgos heredados y agrega algunos nuevos.

Tu tarea es extender el comportamiento de la clase Stack de tal manera que la clase pueda contar todos los elementos que son agregados (push) y quitados (pop). Emplea la clase Stack que proporcionamos en el editor.

Sigue las sugerencias:

Introduce una propiedad diseñada para contar las operaciones pop y nombrarla de una manera que garantice que esté oculta. Inicializala a cero dentro del constructor. Proporciona un método que devuelva el valor asignado actualmente al contador (nómbralo get_counter()).

```
class Stack:
    def __init__(self):
        self.__stk = []

    def push(self, val):
        self.__stk.append(val)

    def pop(self):
        val = self.__stk[-1]
        del self.__stk[-1]
        return val


class CountingStack(Stack):
    def __init__(self):
    #
    # Llena el constructor con acciones apropiadas.
    #

    def get_counter(self):
    #
    # Presenta el valor actual del contador al mundo.
    #

    def pop(self):
    #
    # Haz un pop y actualiza el contador.
    #


stk = CountingStack()
for i in range(100):
    stk.push(i)
    stk.pop()
print(stk.get_counter())
```

Completa el código en el editor. Ejecútalo para comprobar si tu código da como salida 100.

ejercicio

Objetivos

- Mejorar las habilidades del estudiante para definir clases desde cero.
- Implementar estructuras de datos estándar como clases.

Escenario

Como ya sabes, una pila es una estructura de datos que realiza el modelo LIFO (último en entrar, primero en salir). Es fácil y ya te has acostumbrado a ello perfectamente.

Probemos algo nuevo ahora. Una cola (queue) es un modelo de datos caracterizado por el término FIFO: primero en entrar, primero en salir. Nota: una cola (fila) regular que conozcas de las tiendas u oficinas de correos funciona exactamente de la misma manera: un cliente que llegó primero también es el primero en ser atendido.

Tu tarea es implementar la clase Queue con dos operaciones básicas:

- put(elemento), que coloca un elemento al final de la cola.
- get(), que toma un elemento del principio de la cola y lo devuelve como resultado (la cola no puede estar vacía para realizarlo correctamente).

Sigue las sugerencias:

- Emplea una lista como tu almacenamiento (como lo hicimos con la pila).
- put() debe agregar elementos al principio de la lista, mientras que get() debe eliminar los elementos del final de la lista.
- Define una nueva excepción llamada QueueError (elige una excepción de la cual se derivará) y generala cuando get() intentes operar en una lista vacía.

Completa el código que te proporcionamos en el editor. Ejecútalo para comprobar si tu salida es similar a la nuestra.

Salida Esperada

```
1
perro
False
Error de Cola
```

```
class QueueError(???):  # Eligir la clase base para la nueva excepción.
    #
    #  Escribe código aquí.
    #


class Queue:
    def __init__(self):
        #
        # Escribe código aquí.
        #

    def put(self, elem):
        #
        # Escribe código aquí.
        #

    def get(self):
        #
        # Escribe código aquí.
        #


que = Queue()
que.put(1)
que.put("perro")
que.put(False)
```

```
try:
    for i in range(4):
        print(que.get())
except:
    print("Error de Cola")
```

ejercicio

Objetivos

- Mejorar las habilidades del estudiante para definir subclases.
- Agregar nueva funcionalidad a una clase existente.

Escenario

Tu tarea es extender ligeramente las capacidades de la clase Queue. Queremos que tenga un método sin parámetros que devuelva True si la cola está vacía y False de lo contrario.

Completa el código que te proporcionamos en el editor. Ejecútalo para comprobar si genera un resultado similar al nuestro.

Salida esperada:

```
1
perro
False
Cola vacía
```

```
class QueueError(???):
    pass


class Queue:
    #
    # Código del laboratorio anterior.
    #


class SuperQueue(Queue):
    #
    # Escribe código nuevo aquí.
    #


que = SuperQueue()
que.put(1)
que.put("perro")
que.put(False)
for i in range(4):
    if not que.isempty():
        print(que.get())
    else:
```

```
        print("Cola vacía")
```