P3. Programación en iOS: Storyboards

Alumna: Daniela Mendez Ramirez Número de Cuenta: 258331-9

Aplicaciones Móviles

Profesor: Omar Vázquez González

Fecha de Entrega: 28 de enero de 2025

Abstract:

Las pilas y colas son estructuras de datos fundamentales en la programación, utilizadas para organizar y manipular información de manera eficiente. En esta práctica, se implementaron ambas estructuras en Python, utilizando programación orientada a objetos para gestionar elementos complejos, como instancias de automóviles. Se exploraron los principios de funcionamiento de las pilas (LIFO: Last In, First Out) y colas (FIFO: First In, First Out), así como los métodos para insertar, eliminar y consultar elementos dentro de ellas. La implementación permitió comprender el manejo dinámico de datos y su aplicación en situaciones cotidianas dentro del desarrollo de software.

Introducción:

Las estructuras de datos juegan un papel fundamental en la programación, ya que permiten organizar y manipular información de manera eficiente. Entre ellas, las pilas y colas son ampliamente utilizadas debido a su simplicidad y utilidad en múltiples aplicaciones, como la gestión de tareas, el manejo de historial en navegadores o la ejecución de procesos en un sistema operativo.

En esta práctica, se explorará la implementación de pilas y colas en Python, enfocándose en la inserción, eliminación y manipulación de objetos. Para ello, se utilizarán clases y objetos, permitiendo representar elementos complejos, como autos, dentro de estas estructuras. A través de este ejercicio, se reforzará el concepto de estructuras dinámicas y su aplicación en la programación orientada a objetos, comprendiendo cómo los datos se organizan y procesan de manera eficiente.

Código	Resultado

class Pilas:
 	
  def __init__(self):
  # Declaramos nuestra lista para almacenar los elementos
  self.items = []
 	
  def apilar(self, item):
  # Agregar un elemento a la pila
  self.items.append(item)
 	
  def desapilar(self):
  # Elimar y mostrar el ultimo elemento guardado
  if not self.estavacia():
  return self.items.pop()
  return None # En el caso de que la lista este vacia
 	
  def estavacia(self):
  # Devolver True en caso de que la pila este vacia y False si no
  return len(self.items) == 0
 	
  def ultimoElemento(self):
  # Devolver el ultimo elemento sin necesidad de eliminarlo
  if not self.estavacia():
  return self.items[-1]
  return None
pila = Pilas()
pila.apilar(20)
pila.apilar(10)
pila.apilar(33)
 	
print(pila.desapilar())
print(pila.ultimoElemento())
print(pila.desapilar())
print(pila.ultimoElemento())
print(pila.desapilar())
print(pila.estavacia())
	PS Clase_10> python .\pilas.py
	33
	10
	10
	20
	20
	True
	PS Clase_10>
	
	Explicación:
	
	Des apilar, te muestra y a su vez elimina el último elemento guardado, en este caso el 33 fue el último elemento que se guardo en la lista, así que lo muestra pero también lo elimina, entonces, cuándo queremos ver el último elemento de la lista, este ya no es el 33, sino que es el 10, y así sucesivamente pasa con los demás números.

Código	Resultado

class Auto:
  def __init__(self, marca, modelo, color):
  self.marca = marca
  self.modelo = modelo
  self.color = color
 	
  def __str__(self):
  return f"{self.marca} {self.modelo} ({self.color})"
 	
class PilaAutos:
  def __init__(self):
  self.items = [] # Lista para almacenar los autos
 	
  def apilar(self, auto):
  """Agrega un auto a la pila."""
  self.items.append(auto)
 	
  def desapilar(self):
  """Elimina y devuelve el último auto agregado."""
  if not self.estavacia():
  return self.items.pop()
  return None # Si la pila está vacía, retorna None
 	
  def estavacia(self):
  """Devuelve True si la pila está vacía, False si no."""
  return len(self.items) == 0
 	
  def cima(self):
  """Devuelve el auto en la cima sin eliminarlo."""
  if not self.estavacia():
  return self.items[-1]
  return None
 	
# ◇ Creación de objetos Auto 	
ferrari = Auto("Ferrari", "488 Spider", "Rojo")
	PS Clase_10> python .\pilaAutos.py
	Auto desapilado: Volkswagen Sedán (Azul)
	Auto en la cima: Jeep Wrangler (Negro)
	¿Está vacía la pila? False
	PS Clase_10>
	
	Explicación:
	
	El código define una clase Auto que representa un automóvil con atributos como marca, modelo y color, además de un método __str__() que permite imprimir su información de manera legible. Luego, se implementa una clase PilaAutos, que funciona como una estructura de datos tipo pila (LIFO: Last In, First Out), donde los autos se almacenan en una lista y se pueden agregar (apilar), eliminar (desapilar) y consultar el último elemento (cima). Se crean tres objetos Auto (Ferrari, Jeep y Vocho), que son apilados en PilaAutos. Al desapilar, el último auto agregado (Vocho) es eliminado primero, demostrando la naturaleza **LIFO** de la pila. Finalmente, se verifica el auto en la cima (Jeep) y si la pila está vacía, asegurando su correcto funcionamiento.

```
jeep = Auto("Jeep", "Wrangler", "Negro")
vocho = Auto("Volkswagen", "Sedán",
"Azul")

#  ◇  Creación de la pila de autos
pila_autos = PilaAutos()

#  ◇  Apilamos los autos
pila_autos.apilar(ferrari)
pila_autos.apilar(jeep)
pila_autos.apilar(vocho)

#  ◇  Operaciones con la pila de autos
print("Auto desapilado:",
pila_autos.desapilar())  # Salida: Volkswagen
Sedán (Azul)
print("Auto en la cima:",
pila_autos.cima())       # Salida: Jeep
Wrangler (Negro)
print("¿Está vacía la pila?",
pila_autos.estavacia())  # Salida: False
```

Conclusión:

El uso de pilas y colas en la programación es esencial para la gestión eficiente de datos en diversos escenarios, desde la administración de memoria hasta el control de procesos. A través de esta práctica, se demostró cómo estas estructuras pueden ser implementadas en Python utilizando objetos, facilitando la organización de información compleja. Se observó que las pilas operan bajo el principio LIFO, mientras que las colas siguen el modelo FIFO, lo que influye en la forma en que los datos son procesados. Además, el enfoque orientado a objetos permitió una representación más clara y reutilizable de los elementos almacenados. Esta experiencia refuerza la importancia de seleccionar la estructura de datos adecuada según el problema a resolver, optimizando así el rendimiento y la lógica del programa.