UNIVERSIDAD JUAREZ AUTONOMA DE TABASCO

DIVISION ACADEMICA DE INGENIERIA Y ARQUITECTURA

ASIGNATURA:
ALGORITMOS YU ESTRUCTURAS DE DATOS

PROFESOR:
MARIA ELENA GARCIA ULIN

GRUPO:
E4A

TRABAJO:
PILAS Y COLAS 3

ALUMNO:
JOSE JESUS RAMIREZ ALVAREZ

Pilas y colas usando listas

Python incorporado List La estructura de datos viene con métodos para simular operaciones tanto de pila como de cola.
consideremos una pila de letras:

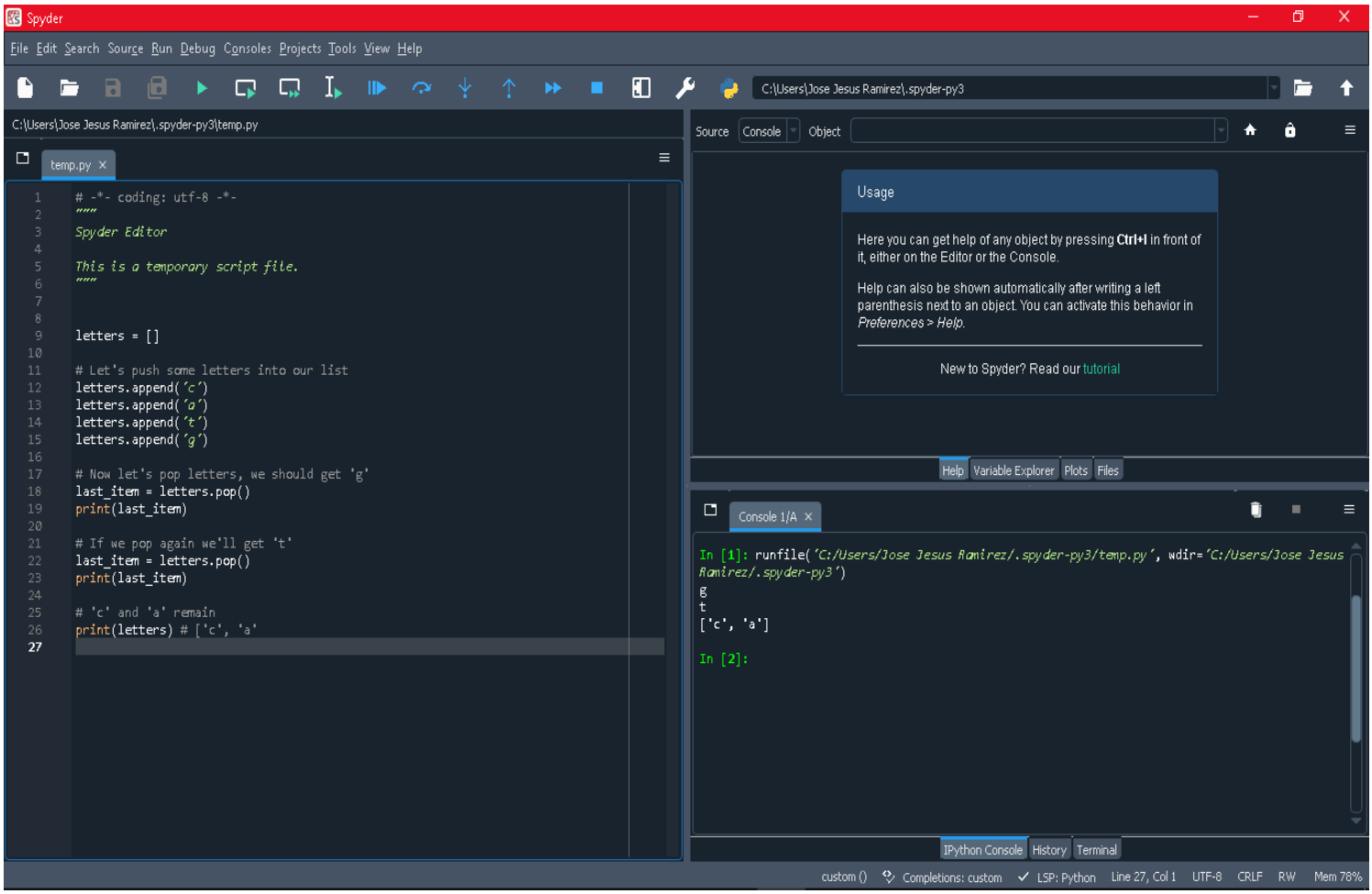
```
letters = []

# Let's push some letters into our list
letters.append('c')
letters.append('a')
letters.append('t')
letters.append('g')

# Now let's pop letters, we should get 'g'
last_item = letters.pop()
print(last_item)

# If we pop again we'll get 't'
last_item = letters.pop()
print(last_item)

# 'c' and 'a' remain
```

print(letters) # ['c', 'a'
Podemos usar las mismas funciones para implementar una cola. los pop La función opcionalmente toma el índice del elemento que queremos recuperar como argumento.

Entonces podemos usar pop con el primer índice de la lista, es decir 0, para obtener un comportamiento similar al de una cola.

Considere una «cola» de frutas:

```
fruits = []

# Let's enqueue some fruits into our list
fruits.append('banana')
fruits.append('grapes')
fruits.append('mango')
fruits.append('orange')

# Now let's dequeue our fruits, we should get 'banana'
first_item = fruits.pop(0)
print(first_item)

# If we dequeue again we'll get 'grapes'
first_item = fruits.pop(0)
print(first_item)
```



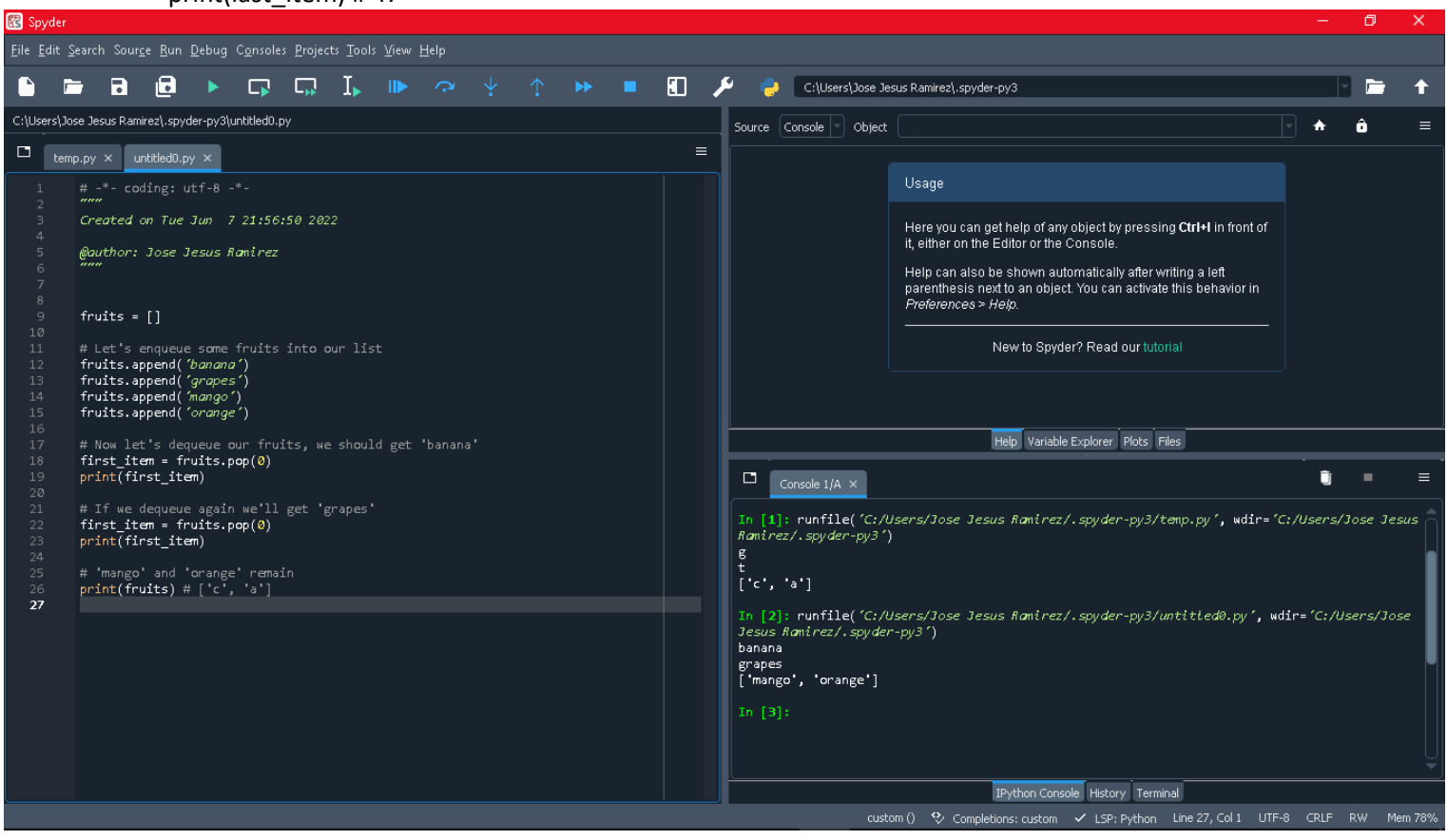
```
# 'mango' and 'orange' remain
print(fruits) # ['c', 'a']
```

Pilas y colas con la biblioteca Deque

Python tiene un deque (pronunciado ‘deck’) biblioteca que proporciona una secuencia con métodos eficientes para trabajar como una pila o una cola.

deque es la abreviatura de Double Ended Queue: una cola generalizada que puede obtener el primer o el último elemento almacenado:

```
from collections import deque

# you can initialize a deque with a list
numbers = deque()

# Use append like before to add elements
numbers.append(99)
numbers.append(15)
numbers.append(82)
numbers.append(50)
numbers.append(47)

# You can pop like a stack
last_item = numbers.pop()
print(last_item) # 47
```



```
print(numbers) # deque([99, 15, 82, 50])

# You can dequeue like a queue
first_item = numbers.popleft()
print(first_item) # 99
print(numbers) # deque([15, 82, 50])
```

Implementaciones más estrictas en Python

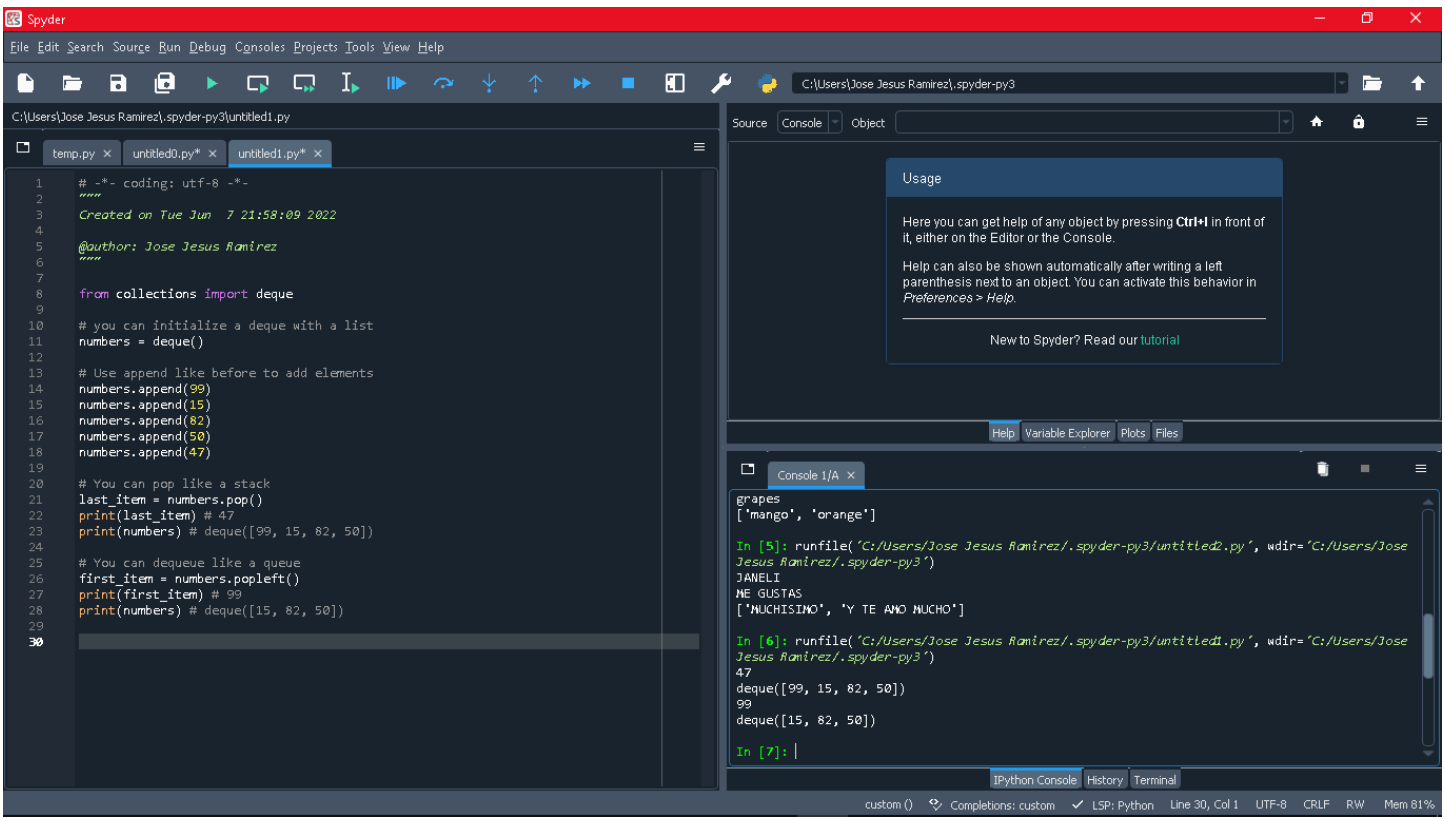
Si su código necesita una pila y proporciona un List, no hay nada que impida que un programador llame insert, remove u otras funciones de lista que afectarán el orden de su pila. Esto arruina fundamentalmente el punto de definir una pila, ya que ya no funciona como debería.

Hay ocasiones en las que nos gustaría asegurarnos de que solo se puedan realizar operaciones válidas con nuestros datos.

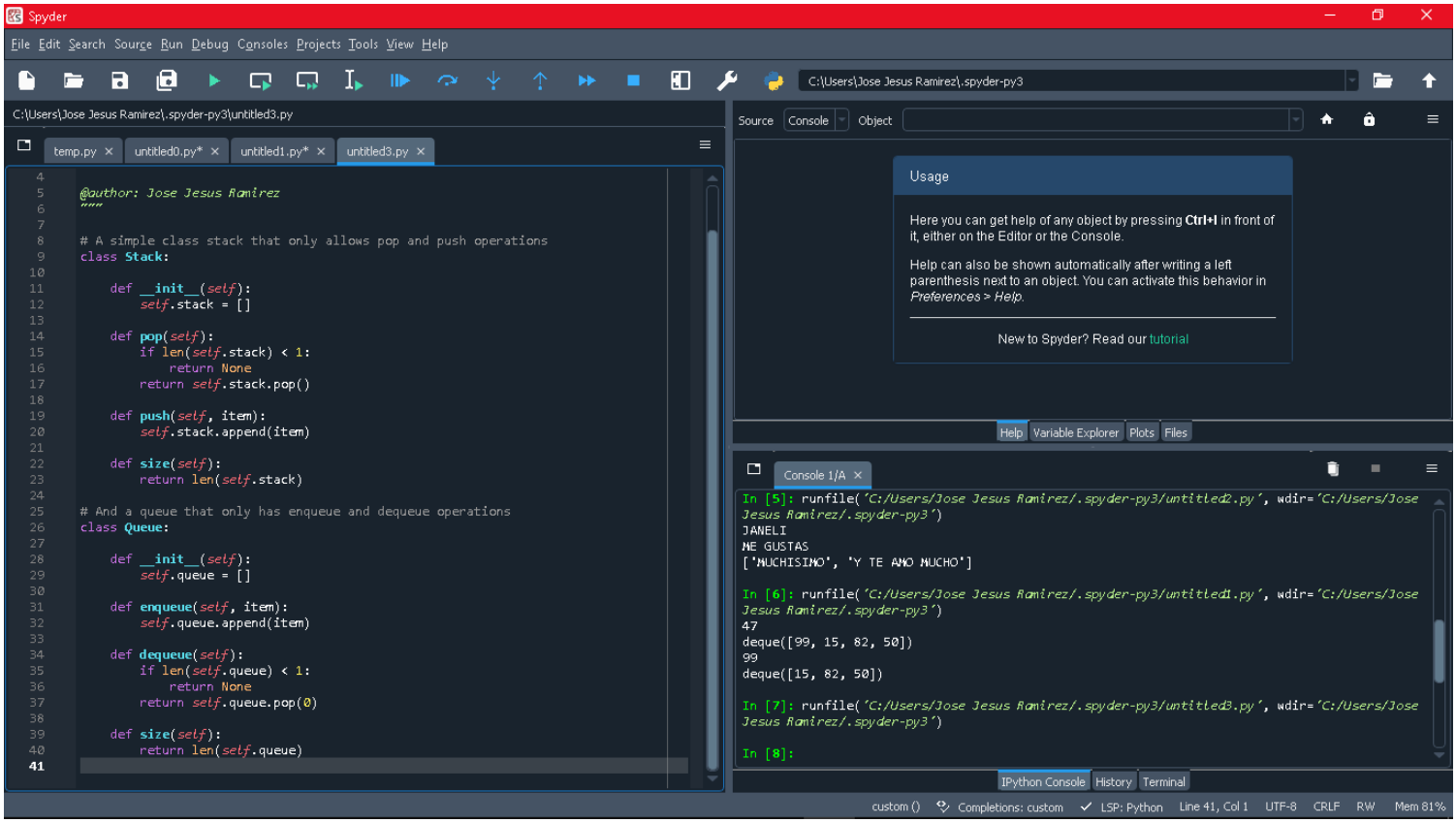
Podemos crear clases que solo exponen los métodos necesarios para cada estructura de datos.

Para hacerlo, creemos un nuevo archivo llamado stack_queue.py y definir dos clases:

```
# A simple class stack that only allows pop and push operations
class Stack:

    def __init__(self):
        self.stack = []

    def pop(self):
        if len(self.stack) < 1:
```



```
            return None
        return self.stack.pop()

    def push(self, item):
        self.stack.append(item)

    def size(self):
        return len(self.stack)

# And a queue that only has enqueue and dequeue operations
class Queue:

    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if len(self.queue) < 1:
            return None
        return self.queue.pop(0)

    def size(self):
        return len(self.queue)
```



Ejemplos

Imagina que eres un desarrollador que trabaja en un nuevo procesador de textos. Tiene la tarea de crear una función de deshacer, lo que permite a los usuarios retroceder en sus acciones hasta el comienzo de la sesión.

Una pila es ideal para este escenario. Podemos registrar cada acción que realiza el usuario empujándola a la pila. Cuando el usuario quiera deshacer una acción, la sacará de la pila. Podemos simular rápidamente la función de esta manera:

```
document_actions = Stack()

# The first enters the title of the document
document_actions.push('action: enter; text_id: 1; text: This is my favourite document')
# Next they center the text
document_actions.push('action: format; text_id: 1; alignment: center')
# As with most writers, the user is unhappy with the first draft and undoes the center alignment
document_actions.pop()
# The title is better on the left with bold font
document_actions.push('action: format; text_id: 1; style: bold')
```

Ahora imagina que eres un desarrollador que trabaja en un nuevo juego de lucha. En su juego, cada vez que se presiona un botón, se activa un evento de entrada. Un evaluador notó que si los botones se presionan demasiado rápido, el juego solo procesa el primero y los movimientos especiales no funcionarán.
Puedes arreglar eso con una cola. Podemos poner en cola todos los eventos de entrada a medida que ingresan. De esta manera, no importa si los eventos de entrada vienen con poco tiempo entre ellos, todos se almacenarán y estarán disponibles para su procesamiento. Cuando procesamos los movimientos, podemos quitarlos de la cola. Se puede realizar un movimiento especial de la siguiente manera:

```
input_queue = Queue()

# The player wants to get the upper hand so pressing the right combination of buttons quickly
input_queue.enqueue('DOWN')
input_queue.enqueue('RIGHT')
input_queue.enqueue('B')

# Now we can process each item in the queue by dequeueing them
key_pressed = input_queue.dequeue() # 'DOWN'

# We'll probably change our player position
key_pressed = input_queue.dequeue() # 'RIGHT'

# We'll change the player's position again and keep track of a potential special move to perform
key_pressed = input_queue.dequeue() # 'B'

# This can do the act, but the game's logic will know to do the special move
```