**Parking Lot Challenge:**

Create a parking lot class that takes in a square footage size as input and creates an array of

empty values based on the input square footage size. Assume every parking spot is 8x12 (96 ft2)

for this program, but have the algorithm that calculates the array size be able to account for

different parking spot sizes. For example, a parking lot of size 2000ft2 can fit 20 cars, but if the

parking spots were 10x12 (120 ft2), it could only fit 16 cars. The size of the array will determine

how many cars can fit in the parking lot.

Parking\_lot.py

import random

# This program simulates a parking lot where cars with 7-digit license plates can park in random spots.

# The ParkingLot class initializes a parking lot of a given size with a specified parking spot size.

# The Car class represents a car with a license plate and methods to park the car in the parking lot.

# The main function simulates the process of parking a list of cars in random spots until the parking lot is full.

class ParkingLot:

def \_\_init\_\_(self, size\_in\_sqft, spot\_length=8, spot\_width=12):

"""

Initialize the parking lot with a given size and spot dimensions.

Calculates the number of spots based on the size and spot dimensions.

"""

self.spot\_size = spot\_length \* spot\_width

self.num\_spots = size\_in\_sqft // self.spot\_size

self.spots = [None] \* self.num\_spots

if self.spot\_size > size\_in\_sqft:

raise ValueError("spot\_size cannot be more than size\_in\_sqft.")

def is\_full(self):

"""

Check if the parking lot is full.

Returns True if there are no empty spots, False otherwise.

"""

for spot in self.spots:

if spot is None:

return False

return True

def find\_random\_empty\_spot(self):

"""

Find a random empty spot in the parking lot.

Returns the index of an empty spot, or None if the lot is full.

"""

empty\_spots = [i for i, spot in enumerate(self.spots) if spot is None]

return random.choice(empty\_spots) if empty\_spots else None

class Car:

def \_\_init\_\_(self, license\_plate):

"""

Initialize the car with a given license plate.

Raises a ValueError if the license plate is not a 7 digit alphanumeric string.

"""

if len(license\_plate) != 7 or not license\_plate.isalnum():

raise ValueError("License plate must be a 7 digit alphanumeric string.")

self.license\_plate = license\_plate

def \_\_str\_\_(self):

"""

Return the license plate as the string representation of the car.

"""

return self.license\_plate

def park(self, parking\_lot, spot\_number):

"""

Attempt to park the car in the given spot number of the parking lot.

Returns a tuple (success, message) indicating whether the parking was successful and a message.

"""

if spot\_number < 0 or spot\_number >= parking\_lot.num\_spots:

return False, f"Spot number {spot\_number} is out of range."

if parking\_lot.spots[spot\_number] is None:

parking\_lot.spots[spot\_number] = self

return True, f"Car with license plate {self.license\_plate} parked successfully in spot {spot\_number}."

else:

return False, f"Spot {spot\_number} is already occupied."

def main(cars, parking\_lot):

"""

Simulate parking each car in the list of cars into random spots in the parking lot.

Continues until all cars are parked or the parking lot is full.

At the end, save the mapping of vehicles to spots in a JSON file and upload it to an S3 bucket.

"""

for car in cars:

if parking\_lot.is\_full():

print("Parking lot is full. Exiting program.")

break

while True:

spot\_number = parking\_lot.find\_random\_empty\_spot()

if spot\_number is None:

print("Parking lot is full. Exiting program.")

break

success, message = car.park(parking\_lot, spot\_number)

print(message)

if success:

break

if \_\_name\_\_ == "\_\_main\_\_":

# Example usage: Create a parking lot and a list of cars, then try to park them.

size\_sft = int(input("Enter the size of the parking lot in square feet: "))

spot\_length = int(input("Enter the length of each parking spot in feet: "))

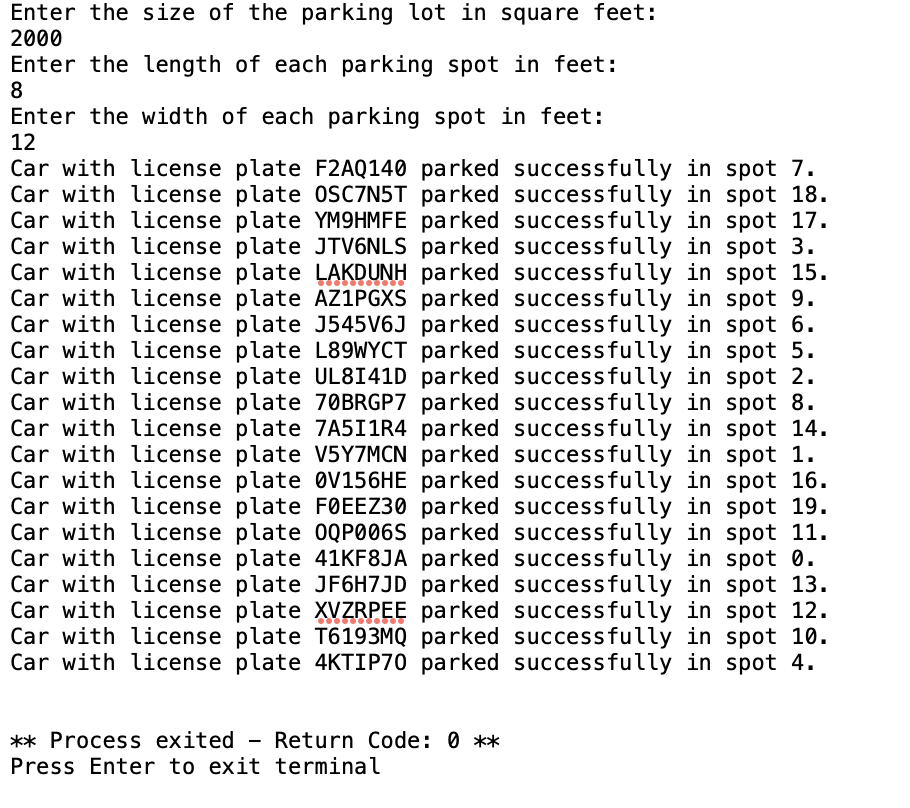
spot\_width = int(input("Enter the width of each parking spot in feet: "))

parking\_lot = ParkingLot(size\_in\_sqft=size\_sft, spot\_length=spot\_length, spot\_width=spot\_width)

cars = [Car("ABC 1234"), Car("XYZ567890"), Car("!LMN3456"), Car("QWE7890"),Car("DBC1234"), Car("WYZ5678"), Car("OMN3456"), Car("PWE7890"),Car("EBC1234"), Car("QYZ5678"), Car("JMN3456")]

#cars = [Car("".join(random.choices("ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789", k=7))) for \_ in range(parking\_lot.num\_spots)]

main(cars, parking\_lot)

**Output:**Case-1: Parked all the cars randomly  
  


Case-2: Limit the square feet area to 100

A screenshot of a computer screen

Description automatically generated

Case-3: Parking Lot Full

A screen shot of a parking code

Description automatically generated

Case-4: Invalid License Plate  
  
  
A screenshot of a computer program

Description automatically generated

Case-5: If spot size greater than total size area  
  
A screenshot of a computer code

Description automatically generated

**Test Cases**

**Positive Test Cases:**

**Valid Parking and Mapping:**

**Scenario:** Park cars in the parking lot and verify the vehicle-to-spot mapping.

**Steps:**

Initialize a parking lot and cars.

Park cars in specific spots.

Call map\_vehicles\_to\_spots and verify the generated mapping matches the expected JSON-like dictionary.

**Expected Outcome:** The mapping should accurately reflect which cars are parked in which spots.

def test\_positive\_valid\_parking\_and\_mapping():

parking\_lot = ParkingLot(size\_in\_sqft=480, spot\_length=8, spot\_width=12)

cars = [Car("ABC1234"), Car("XYZ5678"), Car("LMN3456")]

# Park cars in specific spots

parking\_lot.spots[0] = cars[0] # Car "ABC1234" in spot 0

parking\_lot.spots[2] = cars[1] # Car "XYZ5678" in spot 2

parking\_lot.spots[4] = cars[2] # Car "LMN3456" in spot 4

# Expected mapping

expected\_mapping = {

"0": "ABC1234",

"2": "XYZ5678",

"4": "LMN3456"

}

# Get actual mapping

actual\_mapping = parking\_lot.map\_vehicles\_to\_spots()

# Assert that actual mapping matches expected mapping

assert actual\_mapping == expected\_mapping

----------------------------------------------------------------------------------------------------------------

**Negative Test Cases**

**1.** **Parking Lot Full:**

Scenario: Attempt to park more cars than the parking lot can accommodate.

Steps:

Initialize a parking lot with a small number of spots.

Try to park more cars than there are spots.

Verify that no additional cars can be parked once the lot is full.

Expected Outcome: Cars should not be able to park once the parking lot is full.

def test\_negative\_parking\_lot\_full():

parking\_lot = ParkingLot(size\_in\_sqft=96, spot\_length=8, spot\_width=12) # Only 1 spot available

car1 = Car("ABC1234")

car2 = Car("XYZ5678")

# Park the first car

parking\_lot.spots[0] = car1

# Attempt to park the second car

success, message = car2.park(parking\_lot, 0)

# Assert that parking is not successful

assert not success

assert message == "Spot 0 is already occupied."

---------------------------------------------------------------

**2. Invalid License Plate:**

Scenario: Attempt to create a car with an invalid license plate (not 7 characters or not alphanumeric).

Steps:

Initialize a car with an invalid license plate.

Verify that a ValueError is raised during initialization.

Expected Outcome: Initialization should fail with a ValueError due to an invalid license plate.

def test\_negative\_invalid\_license\_plate():

try:

car = Car("ABC12345") # Invalid: More than 7 characters

assert False # Should not reach here

except ValueError as e:

assert str(e) == "License plate must be a 7 digit alphanumeric string."

try:

car = Car("ABC 123") # Invalid: Contains spaces

assert False # Should not reach here

except ValueError as e:

assert str(e) == "License plate must be a 7 digit alphanumeric string."