**Numpy Exercise**

**An Advanced Flight Management Scenario**

**Section 1: Initial Data Setup**

We'll start by creating larger, more realistic datasets. Note the addition of new data points like Departure\_Time, Ticket\_Price, and a new array for Flight\_Routes.

1. **Flights**: [Flight\_ID, Capacity, Current\_Passengers, Departure\_Time (24h)]
2. **Passengers**: [Passenger\_ID, Flight\_ID, Baggage\_Weight, Ticket\_Price]
3. **Cargo**: [Cargo\_ID, Flight\_ID, Cargo\_Weight]
4. **Member\_Points**: [Passenger\_ID, Points]
5. **Flight\_Routes**: [Flight\_ID, Origin, Destination]

import numpy as np

# 1. More realistic and extensive data for the arrays

flights = np.array([

[101, 150, 148, 830],

[102, 180, 180, 1200],

[103, 120, 95, 1745],

[104, 200, 198, 2200],

[105, 150, 150, 600]

])

passengers = np.array([

[1001, 101, 15, 450],

[1002, 102, 22, 600],

[1003, 101, 18, 480],

[1004, 103, 20, 350],

[1005, 104, 25, 720],

[1006, 102, 30, 650],

[1007, 105, 12, 300]

])

cargo = np.array([

[2001, 101, 500],

[2002, 102, 800],

[2003, 103, 650],

[2004, 104, 1200],

[2005, 105, 400]

])

member\_points = np.array([

[1001, 5000],

[1002, 12000],

[1003, 3000],

[1004, 7500],

[1005, 15000],

[1006, 2500],

[1007, 8000]

])

flight\_routes = np.array([

[101, 'NYC', 'LAX'],

[102, 'SFO', 'JFK'],

[103, 'LAX', 'DEN'],

[104, 'MIA', 'ORD'],

[105, 'LAX', 'SFO']

])

**Section 2: Advanced Indexing, Sorting, and Broadcasting**

**Exercise 1: Sorting and Filtering with np.argsort**

**Problem:**

1. Sort the flights array first by Departure\_Time in ascending order, then by Current\_Passengers in descending order.
2. Find the Flight\_ID of the flight with the highest current number of passengers.

**Solution:**

# 1. Sort flights by departure time (ascending), then by passengers (descending)

# To do this, we need to combine the sorting key. NumPy's lexsort is perfect for this.

# Note: lexsort sorts by the last key provided, then the second to last, etc.

sorted\_indices = np.lexsort((-flights[:, 2], flights[:, 3]))

sorted\_flights = flights[sorted\_indices]

print("Flights sorted by departure time (asc) and then by passengers (desc):")

print(sorted\_flights)

# 2. Find the Flight\_ID of the flight with the most passengers

max\_passengers\_idx = np.argmax(flights[:, 2])

max\_passengers\_flight\_id = flights[max\_passengers\_idx, 0]

print(f"\nFlight with the most passengers: Flight {max\_passengers\_flight\_id}")

**Explanation:**

* np.lexsort is used for **lexicographical sorting**, which is sorting by multiple keys. The keys are passed in a tuple, and the sorting is done from the last key to the first. We use a negative sign on flights[:, 2] to sort Current\_Passengers in **descending** order.
* np.argmax returns the **index** of the maximum value in an array. We use this index to find the corresponding Flight\_ID.

**Exercise 2: Advanced Filtering and Mathematical Operations**

**Problem:**

1. A new policy charges a $25 fee for every kilogram of baggage over a 20 kg limit. Calculate the total **overage fees** collected for all flights.
2. Use a single np.where statement to update the member\_points array:
   * Add 500 points to all passengers with more than 10,000 points.
   * Add 250 points to all others.

**Solution:**

# 1. Calculate total overage fees

baggage\_over\_limit = passengers[:, 2] - 20

overage\_fees\_per\_passenger = np.where(baggage\_over\_limit > 0, baggage\_over\_limit \* 25, 0)

total\_overage\_fees = np.sum(overage\_fees\_per\_passenger)

print(f"Total overage fees collected: ${total\_overage\_fees}")

# 2. Update points based on condition

member\_points[:, 1] = np.where(member\_points[:, 1] > 10000,

member\_points[:, 1] + 500,

member\_points[:, 1] + 250)

print("\nUpdated member points:")

print(member\_points)

**Explanation:**

* np.where(condition, x, y) is a powerful function for **vectorized conditional logic**. It returns elements from x where the condition is True and elements from y otherwise. We use it to calculate overage fees only for passengers with baggage over the limit.
* The second np.where example shows how to perform a complete in-place update of an array based on a single condition, which is much more efficient than using a Python loop.

**Section 3: Set Operations and Data Persistence**

**Exercise 3: Set Operations with np.unique and np.intersect1d**

**Problem:**

1. Find all the unique origins and destinations from the flight\_routes array.
2. Find the intersection of cities that are both an Origin and a Destination.

**Solution:**

# 1. Find unique origins and destinations

unique\_origins = np.unique(flight\_routes[:, 1])

unique\_destinations = np.unique(flight\_routes[:, 2])

all\_cities = np.unique(np.hstack((unique\_origins, unique\_destinations)))

print("All unique cities (origin or destination):")

print(all\_cities)

# 2. Find cities that are both an origin and a destination

intersection\_cities = np.intersect1d(unique\_origins, unique\_destinations)

print("\nCities that are both an origin and a destination:")

print(intersection\_cities)

**Explanation:**

* np.unique returns the **unique elements** of an array. We use it on the origin and destination columns.
* np.hstack horizontally stacks arrays. We use it here to combine the unique origins and destinations into a single array before finding the unique values.
* np.intersect1d finds the **common unique elements** between two arrays. This is a fast and efficient way to perform set-like operations on NumPy arrays.

**Exercise 4: Saving and Loading Data**

**Problem:**

1. Save the flights and passengers arrays to separate files.
2. Load the passengers array from its file and verify its contents.

**Solution:**

# 1. Save arrays to files

np.save('flights.npy', flights)

np.save('passengers.npy', passengers)

print("Arrays saved to 'flights.npy' and 'passengers.npy'.")

# 2. Load the passengers array

loaded\_passengers = np.load('passengers.npy')

print("\nLoaded passengers array:")

print(loaded\_passengers)

# Verify the loaded data is the same as the original

if np.array\_equal(passengers, loaded\_passengers):

print("\nVerification successful: Loaded data is identical to the original.")

else:

print("\nVerification failed.")

**Explanation:**

* np.save(filename, array) saves a NumPy array to a binary file with a .npy extension. This is the standard way to store NumPy arrays for later use.
* np.load(filename) reads the .npy file and reconstructs the array in memory.
* np.array\_equal is a useful function for comparing two arrays element-wise, returning True only if they have the same shape and contents.

add dtypes to the above arrays

**Numpy Exercise Handbook: Data Types**

**Section 1: Initial Data Setup with Dtypes**

This section shows how to create the arrays with specific data types, a crucial step for working with large datasets. We'll use a **structured array** for the flight\_routes array to handle its mixed data types (integer and string).

Python

import numpy as np

# 1. Flights array with int16 dtype for memory efficiency

# [Flight\_ID, Capacity, Current\_Passengers, Departure\_Time]

flights = np.array([

[101, 150, 148, 830],

[102, 180, 180, 1200],

[103, 120, 95, 1745],

[104, 200, 198, 2200],

[105, 150, 150, 600]

], dtype=np.int16)

# 2. Passengers array with int32 dtype to accommodate larger ticket prices

# [Passenger\_ID, Flight\_ID, Baggage\_Weight, Ticket\_Price]

passengers = np.array([

[1001, 101, 15, 450],

[1002, 102, 22, 600],

[1003, 101, 18, 480],

[1004, 103, 20, 350],

[1005, 104, 25, 720],

[1006, 102, 30, 650],

[1007, 105, 12, 300]

], dtype=np.int32)

# 3. Cargo array with int16 dtype

# [Cargo\_ID, Flight\_ID, Cargo\_Weight]

cargo = np.array([

[2001, 101, 500],

[2002, 102, 800],

[2003, 103, 650],

[2004, 104, 1200],

[2005, 105, 400]

], dtype=np.int16)

# 4. Member\_Points array with int32 dtype for larger point values

# [Passenger\_ID, Points]

member\_points = np.array([

[1001, 5000],

[1002, 12000],

[1003, 3000],

[1004, 7500],

[1005, 15000],

[1006, 2500],

[1007, 8000]

], dtype=np.int32)

# 5. Flight\_Routes array using a structured dtype for mixed data types

# This is an important technique for handling different data types in one array.

flight\_route\_dtype = np.dtype([('flight\_id', 'i4'), ('origin', 'U3'), ('destination', 'U3')])

flight\_routes = np.array([

(101, 'NYC', 'LAX'),

(102, 'SFO', 'JFK'),

(103, 'LAX', 'DEN'),

(104, 'MIA', 'ORD'),

(105, 'LAX', 'SFO')

], dtype=flight\_route\_dtype)

print("Updated flights array with dtype:", flights.dtype)

print("Updated passengers array with dtype:", passengers.dtype)

print("Updated cargo array with dtype:", cargo.dtype)

print("Updated member points array with dtype:", member\_points.dtype)

print("Updated flight routes array with structured dtype:")

print(flight\_routes)

**Explanation of Dtypes:**

* np.int16: A 16-bit integer, suitable for values up to 32,767. This is sufficient for most of our data and uses less memory than the default int64.
* np.int32: A 32-bit integer, which can handle larger values than int16. We use this for ticket prices and member points to allow for bigger numbers if needed.
* **Structured Dtype**: For the flight\_routes array, we need to store a mix of numbers and strings. A **structured array** allows us to define named fields with different data types.
  + ('flight\_id', 'i4'): A field named flight\_id that is a 4-byte integer.
  + ('origin', 'U3'): A field named origin that is a Unicode string of length 3 ('U3'). This ensures consistency for our 3-letter airport codes.