Q1. Python objects, mutability, and function defaults (4 marks)

Question:

- Explain mutable vs immutable objects with one short code example each, then show how name binding impacts lists, and finally write a function that safely uses a list as an optional parameter using a None default.

Q1. Python objects, mutability, and function defaults (4 marks)

Question:

- Explain mutable vs immutable objects with one short code example each, then show how name binding impacts lists, and finally write a function that safely uses a list as an optional parameter using a None default.

- Name binding means variable names point to objects; for mutable lists, two names can refer to the same list, so mutating via one name is visible through the other

The safe default-argument pattern uses y=None and initializes a new list inside the function to avoid sharing state across calls

CODE

```
# Immutable
    x = (1, 2, 3)
# x[0] = 10  # TypeError if uncommented
# Mutable + name binding
    a = [1, 2]
    b = a
    a.append(3)
# b is now [1, 2, 3] as well
# Safe default for list parameter
    def push(val, arr=None):
        if arr is None:
            arr = []
     arr.append(val)
      return arr
```

Q2. NumPy arrays: slicing, broadcasting, and basic stats (4 marks)

Question: Given a = [4 6 8, 2 9 5,3 1 10] as a NumPy array, extract the last column, set every alternate row to zero via slicing, compute elementwise y=2x+1 for x=np.linspace(0,1,5), and report mean and standard deviation of x using unbiased std

Answer:

- Use a:, 2 for the last column, and a::2, : = 0 for alternating rows via strides as covered in the arrays and slicing sections.

Broadcasting supports y = 2*x + 1 elementwise without loops, and basic stats are available as np.mean(x) and np.std(x, ddof=1) for the unbiased estimator per the statistics notes .

CODE:

```
import numpy as np

a = np.array([[1,2,3],[4,5,6],[7,8,9]])

last_col = a[:, 2]

a[::2, :] = 0  # alternate rows zeroed in-place

x = np.linspace(0, 1, 5)

y = 2*x + 1

mu = np.mean(x)

sigma_unbiased = np.std(x, ddof=1)
```

Q3. pandas Series/DataFrame essentials and indexing (4 marks)

Question:

- Create a DataFrame with columns x=4 6, y='a','b','c', add column z=x**2, show df.loc[:1, 'x','z'], explain how NaN arises from misaligned labels in Series addition, and mention one method to summarize numeric columns

Answer:

- pandas aligns by labels, constructs Series and DataFrames with explicit or implicit indexes, and supports vectorized column operations such as z = df'x'**2
- df.loc uses label-based indexing, so df.loc[:1, 'x','z'] selects rows up to label 1 and the specified columns, whereas df.iloc provides integer-position-based selection
- Adding two Series with different indexes produces the union of indexes and NaN for missing labels, and df.describe() summarizes numeric columns while df.dtypes helps verify parsing . Code:

```
import pandas as pd

df = pd.DataFrame(dict(x=[0,1,2], y=['a','b','c']))

df['z'] = df['x']**2

view = df.loc[:1, ['x','z']]

s1 = pd.Series([1,2], index=['a','b'])
```

s2 = pd.Series([10,20], index=['b','c'])

sum_ = s1 + s2 # index union => NaN where labels don’t align

summary = df.describe()

Q4. Probabilistic programming: RNGs, seeding, and biased coin (4 marks)

Question:

- Using NumPy’s RNG, simulate 10,000 biased coin flips with P(heads)=0.3 via np.random.choice, estimate the empirical probability of heads, and explain why seeding is used and how to set it.

Answer:

- NumPy provides pseudo-random variates through np.random.choice, supporting probabilities via p=, enabling discrete random experiments fundamental to Monte Carlo ideas in the notes
- np.random.seed(123) fixes the RNG state to ensure repeatability for debugging and consistent results across runs as emphasized in the seeding section
- The empirical probability is mean(outcomes == ‘H’), and histogram-based intuition connects to empirical distributions discussed in the statistics material

CODE

import numpy as np

np.random.seed(123)

events = ['H','T']

x = np.random.choice(events, size=10_000, p=[0.3, 0.7])

p_hat = np.mean(np.array(x) == 'H')

Q5. Linear regression: formulas, fitting, and R^2 (4 marks)

Question:

- State the OLS slope and intercept in simple linear regression in terms of $S_{xx}, S_{xy}, \bar{x}, \bar{y}$, define R^2, and show how to fit a straight line using np.polyfit and compute R^2 using sums of squares as in the notes [3].

Answer:

- With $S_{xx} = \sum_i (x_i - \bar{x})^2$ and $S_{xy} = \sum_i (x_i - \bar{x})(y_i - \bar{y})$, the slope $b_1 = S_{xy}/S_{xx}$ and intercept $b_0 = \bar{y} - b_1\bar{x}$ as derived under normal equations in the regression slides [3].
- The coefficient of determination is $R^2 = 1 - SS_R/S_{YY}$ with $SS_R = \sum_i (y_i - \hat{y}i)^2$ and $SYY = \sum_i (y_i - \bar{y})^2$, and standardized residuals should roughly lie within ± 2 with no pattern if the linear model is adequate [3].
- np.polyfit(x,y,1) returns slope, intercept, and a helper get_rsq can compute R^2 using these sums as demonstrated in class [3].

Code:

import numpy as np

```
def fit(x, y):
    return np.polyfit(x, y, 1)


def predict(coef, x):
    return coef[0]*x + coef[1]


def get_rsq(y, yp):
    ssr = np.sum((y - yp)**2)
    ybar = np.mean(y)
    syy = np.sum((y - ybar)**2)
    return 1.0 - ssr/syy


x = np.linspace(0, 1, 20)
y = 2*x + 0.25 + np.random.normal(0, 0.2, size=len(x))
coef = fit(x, y)
yp = predict(coef, x)
r2 = get_rsq(y, yp)
```

Mini-project prompt

Prompt:

- Generate a synthetic dataset following a linear model with Gaussian noise, assemble a pandas DataFrame, fit a simple linear regression using NumPy's polyfit, report the slope, intercept, and R^2, then use bootstrap resampling on rows to estimate 90% confidence intervals for the coefficients and the fitted curve, and finally outline a residual check consistent with the regression notes.
- The deliverables are: printed coefficients with 90% CIs, R^2 , two arrays giving the 5th and 95th percentile fitted values across x, and code that can be adapted later to real CSVs with pandas read_csv and df.sample for bootstrapping as shown in the pandas and regression sections .

Implementation

The steps below mirror the course's NumPy, pandas, statistics, and regression patterns, including np.polyfit for slope/intercept, sums-of-squares for R^2, and DataFrame-based bootstrap via df.sample with replace=True

```
import numpy as np

import pandas as pd


# 1) Synthetic data with known linear relation and Gaussian noise

np.random.seed(123)              # repeatability

n = 60

x = np.linspace(0, 3, n)

true_slope, true_intercept, noise_sd = 3.5, 1.2, 0.6

y = true_slope*x + true_intercept + np.random.normal(0, noise_sd, size=n)


# 2) Assemble DataFrame (aligns with pandas practice)

df = pd.DataFrame(dict(x=x, y=y))


# 3) Fit linear regression via NumPy polyfit (degree 1)

coef = np.polyfit(df.x, df.y, 1)    # [slope, intercept]

slope, intercept = coef


# 4) Predictions and R^2 using sums of squares

yp = slope*df.x + intercept

ssr = np.sum((df.y - yp)**2)

syy = np.sum((df.y - np.mean(df.y))**2)

r2 = 1.0 - ssr/syy


print(f"Fit: y = {slope:.3f} x + {intercept:.3f}")

print(f"R^2  = {r2:.4f}")


# 5) Bootstrap coefficients and fitted curves (90% CI)

B = 1000

coefs = []
```

```
yps = []
for _ in range(B):
    # sample rows with replacement; keep (x,y) pairs intact
    df_b = df.sample(n=len(df), replace=True)
    cb = np.polyfit(df_b.x, df_b.y, 1)
    coefs.append(cb)
    yps.append(cb[0]*df.x + cb[1])

coefs = np.asarray(coefs)      # shape: (B, 2)
yps = np.asarray(yps)          # shape: (B, n)

coef_low, coef_high = np.percentile(coefs, [5, 95], axis=0)
y_low, y_high = np.percentile(yps, [5, 95], axis=0)

print(f"Slope 90% CI     : [{coef_low[0]:.3f}, {coef_high[0]:.3f}]")
print(f"Intercept 90% CI : [{coef_low[1]:.3f}, {coef_high[1]:.3f}]")

# 6) Residual check scaffold (standardized residuals)
sigma = np.sqrt(np.sum((df.y - yp)**2) / (len(df) - 2))
std_res = (df.y - yp) / sigma
# A quick sanity: most std_res should lie roughly within ±2 if model adequate
frac_within_2 = np.mean(np.abs(std_res) <= 2)
print(f"Std residuals within ±2: {frac_within_2*100:.1f}%")
```

This implementation follows the exact techniques shown in the notes: vectorized NumPy operations and np.polyfit for regression, DataFrame-centric bootstrapping with df.sample, using sums of squares for , and standardized residuals as an assessment cue

Set 1 Detailed Answers

Q1. Mutable vs Immutable Types

- Immutable objects cannot be changed after creation: numbers, strings, tuples. Example: `x = (1,2,3)` cannot be modified by `x[0]=10`.
- Mutable objects can be changed in-place: lists, dicts, sets. Example:

```python
a = [1,2]; b = a; a.append(3)  # b is now also [1,2,3]
```

- Names bind to objects; mutable mutations via one name are seen through others.

Q2. NumPy slicing & stats

```python
import numpy as np
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
last_col = a[:, 2]          # array([3,6,9])
a[::2, :] = 0              # zero alternating rows
x = np.linspace(0,1,5)
y = 2*x + 1
mu = np.mean(x)
std = np.std(x, ddof=1)
```

Q3. pandas DataFrame and indexing

```python
import pandas as pd
df = pd.DataFrame({'x':[0,1,2], 'y':['a','b','c']})
df['z'] = df['x']**2
subset = df.loc[:1, ['x','z']]
s1 = pd.Series([1,2], index=['a','b'])
s2 = pd.Series([10,20], index=['b','c'])
sum_ = s1 + s2  # NaN at labels not common in both
summary = df.describe()
```

Q4. Biased coin flips

```python
np.random.seed(123)
outcomes = np.random.choice(['H','T'], size=10_000, p=[0.3,0.7])
p_heads = np.mean(outcomes == 'H')
```

- Seeding fixes pseudo-randomness for reproducibility.

Q5. Linear regression formulas and R^2

Slope and intercept:

$$b_1 = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2}, \quad b_0 = \bar{y} - b_1 \bar{x}$$

Coefficient of determination:

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$$

Code:

```python
coef = np.polyfit(x, y, 1)
yp = coef[0]*x + coef[1]
ssr = np.sum((y - yp)**2)
syy = np.sum((y - np.mean(y))**2)
r2 = 1 - ssr/syy
```

Q1 - Mutable vs Immutable

Python's **immutable** types like `int` and `tuple` cannot be changed after they are created, so operations that try to change them will fail or create new objects.

```python
t = (1, 2, 3)
# t[0] = 10  # Would raise a TypeError
```

Mutable types like lists can be changed in place, affecting all references to that object:

```python
lst1 = [1, 2]
lst2 = lst1
lst1.append(3)
print(lst2)  # [1, 2, 3]
```

Safe mutable default argument example:

```python
def safe_append(x, lst=None):
    if lst is None:
        lst = []
    lst.append(x)
    return lst

print(safe_append(1))  # [1]
print(safe_append(2))  # [2], fresh list each call
```

Q2 - NumPy slicing & broadcasting

Extract last column and zero alternate rows in a 3x3 array:

```python
import numpy as np
a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("Last column:", a[:, 2])  # [3 6 9]
a[::2, :] = 0  # Zero every alternate row (0,2)
print(a)
```

Broadcast vectorized operations without loops:

```python
x = np.linspace(0, 1, 5)
y = 2*x + 1  # Broadcast multiply and add
print(y)  # [1. 1.5 2. 2.5 3.]
```

Q3 - pandas indexing and alignment

Create DataFrame and derive column:

```python
import pandas as pd
df = pd.DataFrame({'x': [0, 1, 2], 'y': ['a', 'b', 'c']})
df['z'] = df['x'] ** 2
print(df.loc[:1, ['x', 'z']])
```

Add Series with different indexes showing NaN for missing keys:

```python
s1 = pd.Series([1, 2], index=['a', 'b'])
s2 = pd.Series([10, 20], index=['b', 'c'])
print(s1 + s2)
# Output:
# a     NaN
# b    12.0
# c     NaN
```

Q4 - Simulate biased coin flips

```python
np.random.seed(123)
flips = np.random.choice(['H', 'T'], size=10000, p=[0.3, 0.7])
prob_H = np.mean(flips == 'H')
print("Empirical P(H):", prob_H)
```

Q5 - Linear regression and R^2

Synthetic

```python
x = np.linspace(0, 10, 50)
y = 2.5*x + 1 + np.random.normal(0, 1, 50)
coef = np.polyfit(x, y, 1)
y_pred = np.polyval(coef, x)
ss_res = ((y - y_pred)**2).sum()
ss_tot = ((y - y.mean())**2).sum()
r2 = 1 - ss_res/ss_tot
print(f"Slope: {coef[0]}, Intercept: {coef[1]}, R²: {r2:.3f}")
```

Set 2 Examples and Explanation

Q1 - Safe Default Mutable Argument Function

Problem: Using a mutable default argument leads to the argument being shared across calls, causing unexpected behavior.

Example of problematic code:

```python
def add_number(num, num_list=[]):
    num_list.append(num)
    return num_list

print(add_number(1))  # [1]
print(add_number(2))  # [1, 2], shared list!
```

Safe code with None default:

```python
def add_number(num, num_list=None):
    if num_list is None:
        num_list = []
    num_list.append(num)
    return num_list

print(add_number(1))  # [1]
print(add_number(2))  # [2], new list each time
```

Q2 - NumPy Array Creation, Slicing, Broadcasting

Create and slice arrays:

```python
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])
print("Original array:\n", arr)

# Slice to get second column
second_col = arr[:, 1]
print("Second column:", second_col)

# Set all values in first row to zero
arr[0, :] = 0
print("After zeroing first row:\n", arr)

# Broadcasting example
x = np.array([1, 2, 3])
y = x * 2 + 1
print("Broadcasted result:", y)
```

Q3 - pandas loc vs iloc

Label based indexing `.loc`:

```python
import pandas as pd
df = pd.DataFrame({'A': [10, 20, 30], 'B': [100, 200, 300]}, index=['a', 'b', 'c'])

print(df.loc['a':'b', ['A']])
# Output:
#    A
# a 10
# b 20
```

Position based indexing `.iloc`:

```python
print(df.iloc[0:2, 0])
# Output:
# a    10
# b    20
# Name: A, dtype: int64
```

Q4 - Probabilistic Programming Dice Roll

Random 6-sided die roll(s):

```python
import numpy as np

def roll_dice(n=1):
    return np.random.randint(1, 7, size=n)

print("Single roll:", roll_dice())
print("Five rolls:", roll_dice(5))
```

- Probabilistic programming uses randomness within code to simulate outcomes and model uncertainty.

Q5 - Linear Regression Fit and Residuals

Fit a simple linear model and compute residuals:

```python
import numpy as np

x = np.linspace(0, 5, 10)
y = 2*x + 1 + np.random.normal(0, 0.5, 10)

coef = np.polyfit(x, y, 1)
y_pred = np.polyval(coef, x)
residuals = y - y_pred

print("Coefficients: slope =", coef[0], ", intercept =", coef[1])
print("Residuals:", residuals)
```

Set 3 Examples and Explanation

Q1 - Python Object Model and Mutability

Mutability is a core object property:

```python
s = "hello"        # Immutable string
s2 = s.upper()     # Returns new string, original unchanged
print(s, s2)       # hello HELLO

lst = [1, 2, 3]    # Mutable list
lst.append(4)      # Modified in-place
print(lst)         # [1, 2, 3, 4]
```

Q2 - pandas Derived Columns

Creating new columns from existing ones:

```python
import pandas as pd

df = pd.DataFrame({'a': [1, 2], 'b': [3, 4]})
df['c'] = df['a'] + df['b']
print(df)
# Output:
#    a  b  c
# 0  1  3  4
# 1  2  4  6
```

Q3 - Plot sine curve with matplotlib

Plot sinusoidal function:

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 100)
y = np.sin(x)

plt.plot(x, y)
plt.title('Sine Wave')
plt.xlabel('x')
plt.ylabel('sin(x)')
plt.show()
```

Q4 - RNG Seeding Example

Random sequences reproducible using seed:

```python
import numpy as np

np.random.seed(10)
print(np.random.rand(3))
# This will output the same sequence every time you run it.
```

Q5 - Residual Analysis Importance

Residuals show difference between observed and predicted values:

```python
residuals = y - y_pred
print("Mean residual:", np.mean(residuals))
print("Residuals standard deviation:", np.std(residuals))
```

Set 4 Examples and Explanation

Q1 - Name Binding and Shared Mutation

```python
a = [10, 20]
b = a  # both reference same list
a.append(30)
print(b)  # [10, 20, 30]
```

Q2 - pandas Selection and Missing Data

```python
import pandas as pd

df = pd.DataFrame({'x': [1, 2, None], 'y': [4, None, 6]})
print(df.loc[:, ['x', 'y']])  # Selecting columns

df_filled = df.fillna(0)
print(df_filled)
```

Q3 - f-string Formatting

```python
name = "John"
age = 28
print(f"My name is {name} and I am {age} years old.")
```

Q4 - Monte Carlo Dice Rolls

```python
rolls = np.random.randint(1, 7, 1000)
print("First 10 rolls:", rolls[:10])
```

Set 5 Examples and Explanation

Q1 - Python Modules and Imports

```python
# module1.py
def greet():
    return "Hello, World!"

# main.py
import module1
print(module1.greet())

from module1 import greet
print(greet())
```

Q2 - NumPy Multi-dimensional Indexing and Slicing

```python
import numpy as np

arr = np.arange(24).reshape(4,6)
print(arr[:2, :3])  # first two rows and first three columns
```

Q3 - File I/O and String Methods

```python
with open('test.txt', 'w') as f:
    f.write("Hello Python")

with open('test.txt', 'r') as f:
    content = f.read()
print(content.upper())  # HELLO PYTHON
```

Q4 - Statistics Role in ML

- Variance quantifies data spread.
- Covariance measures joint variability of two variables, foundation for multidimensional ML models.

Q5 - Calculating R^2 in Regression

```python
ssr = np.sum((y - y_pred)**2)       # residual sum of squares
sst = np.sum((y - np.mean(y))**2)   # total sum of squares
r2 = 1 - ssr / sst
print("R-squared:", r2)
```

CONCEPTS TO CODE

Mutable and Immutable Objects in Python

- **Immutable types** cannot change after creation: int, float, string, tuple. Attempting to modify them raises errors.
  Example:

```python
x = (1, 2, 3)
# x[0] = 4  # Raises TypeError: tuple does not support item assignment
print(x)    # Output: (1, 2, 3)
```

- **Mutable types** can be updated in-place: list, dict, set. Updates affect all references to the object.
  Example:

```python
a = [1, 2, 3]
b = a   # b references same list as a
a.append(4)
print(b)  # Output: [1, 2, 3, 4], b reflects change
```

This shows Python variables bind to objects by reference; mutable object changes via any reference are visible.

Safe Default Mutable Arguments in Functions

Mutable defaults persist across calls, which can cause unexpected shared state.

Example problem:

```python
def add_element(value, lst=[]):
    lst.append(value)
    return lst

print(add_element(1))  # [1]
print(add_element(2))  # [1, 2] - shared list across calls unexpectedly!
```

Safe solution uses `None` and creates new list within:

```python
def add_element(value, lst=None):
    if lst is None:
        lst = []
    lst.append(value)
    return lst

print(add_element(1))  # [1]
print(add_element(2))  # [2] - fresh list each call as expected
```

NumPy Array Slicing and Broadcasting

Create a 3x3 NumPy array:

```python
import numpy as np

a = np.array([[1,2,3],
              [4,5,6],
              [7,8,9]])
```

- Extract last column:

```python
last_col = a[:, 2]
print(last_col)  # Output: [3 6 9]
```

- Zero every alternate row:

```python
a[::2, :] = 0
print(a)
# Output:
# [[0 0 0]
#  [4 5 6]
#  [0 0 0]]
```

Broadcasting example:

Create array x and compute y = 2x + 1:

```python
x = np.linspace(0, 1, 5)
y = 2*x + 1
print(y)  # Output: [1. 1.5 2. 2.5 3.]
```

pandas DataFrame Indexing and Alignment

Create a DataFrame and add a derived column:

```python
import pandas as pd

df = pd.DataFrame({'x': [0, 1, 2], 'y': ['a', 'b', 'c']})
df['z'] = df['x'] ** 2
print(df)
#    x  y  z
# 0  0  a  0
# 1  1  b  1
# 2  2  c  4
```

Use `.loc` label-based indexing:

```python
subset = df.loc[:1, ['x', 'z']]
print(subset)
#    x  z
# 0  0  0
# 1  1  1
```

Adding Series with overlapping but non-identical indexes creates NaN for labels missing in one:

```python
s1 = pd.Series([1,2], index=['a','b'])
s2 = pd.Series([10,20], index=['b','c'])
sum_series = s1 + s2
print(sum_series)
# a     NaN
# b    12.0
# c     NaN
```

Simple Linear Regression Fit and R² Calculation

Generate synthetic

```python
import numpy as np

np.random.seed(42)
x = np.linspace(0, 10, 50)
true_slope = 2.5
true_intercept = 1.0
noise = np.random.normal(0, 1, size=x.shape)
y = true_slope * x + true_intercept + noise
```

Fit regression line:

```python
coef = np.polyfit(x, y, 1)  # Returns array: [slope, intercept]
y_pred = np.polyval(coef, x)
```

Calculate R^2:

```python
ss_res = np.sum((y - y_pred)**2)
ss_tot = np.sum((y - np.mean(y))**2)
r2 = 1 - (ss_res / ss_tot)
print(f"Slope: {coef[0]:.3f}, Intercept: {coef[1]:.3f}, R²: {r2:.3f}")
```

Simulating Biased Coin Flips with NumPy

Generate 10,000 biased coin flips with $P(H) = 0.3$:

```python
np.random.seed(123)  # For reproducibility
outcomes = np.random.choice(['H', 'T'], size=10000, p=[0.3, 0.7])
p_heads_empirical = np.mean(outcomes == 'H')
print(f"Empirical P(heads): {p_heads_empirical:.3f}")
```

- Setting seed ensures repeated results for consistent testing.