```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""

@author: fran-pellegrino
"""

# Question 2: standardizing predictor variables 2 and 3

import csv

file_path = "housingUnits.csv"

data = []
with open(file_path, "r") as f:
    reader = csv.reader(f)
    header = next(reader)  # save header row
    for row in reader:
        data.append([float(x) for x in row])  # convert all to float

# Create new standardized predictors
for row in data:
    total_rooms = row[1]          # predictor 2
    total_bedrooms = row[2]       # predictor 3
    households = row[4]           # predictor 5

    # Avoid division by zero
    if households != 0:
        rooms_per_household = total_rooms / households
        bedrooms_per_household = total_bedrooms / households
    else:
        rooms_per_household = 0
        bedrooms_per_household = 0

    # Append new values to each row
    row.append(rooms_per_household)
    row.append(bedrooms_per_household)

# Update header to include new variables
header.extend(["rooms_per_household", "bedrooms_per_household"])
# Chose not to write data to a new csv to keep on disk, limiting standardized var

# say index 1 = rooms_per_household, 2 = bedrooms_per_household if you added them
for row in data[1:6]:   # skip header row, show first 5 rows
    print("Rooms per household:", row[8], "| Bedrooms per household:", row[9])

# Question 2 plot of standardize predictors 2 and 3
import matplotlib.pyplot as plt

data = []
with open("housingUnits.csv", "r") as f:
```

```
    reader = csv.DictReader(f)
    for row in reader:
        households = float(row['households'])
        row['rooms_per_household'] = float(row['total_rooms']) / households
        row['bedrooms_per_household'] = float(row['total_bedrooms']) / households
        row['median_house_value'] = float(row['median_house_value'])
        data.append(row)

# Extract values for plotting
rooms_per_household = [float(row['rooms_per_household']) for row in data]
bedrooms_per_household = [float(row['bedrooms_per_household']) for row in data]
median_house_value = [row['median_house_value'] for row in data]

# Plot 1: Rooms per household vs median house value
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.scatter(rooms_per_household, median_house_value, alpha=0.3)
plt.xlabel("Rooms per Household")
plt.ylabel("Median House Value ($)")
plt.title("Rooms per Household vs Median House Value")

# Plot 2: Bedrooms per household vs median house value
plt.subplot(1, 2, 2)
plt.scatter(bedrooms_per_household, median_house_value, alpha=0.3, color="orange"
plt.xlabel("Bedrooms per Household")
plt.ylabel("Median House Value ($)")
plt.title("Bedrooms per Household vs Median House Value")

plt.tight_layout()
plt.show()

# Question 3 code below

import math
import numpy as np

with open(file_path, "r") as f:
    reader = csv.DictReader(f)
    fieldnames = reader.fieldnames[:]   # list of column names, in order
    # Confirm structure assumption: last col = target
    print("Detected columns (in order):", fieldnames)

    # We assume columns are in the order you showed:
    # index 0: median age
    # index 1: total_rooms  <-- predictor 2
    # index 2: total_bedrooms <-- predictor 3
    # index 3: population <-- predictor 4
    # index 4: households <-- predictor 5 (denominator)
    # index 5: median_income
    # index 6: ocean_prox
    # index 7: median_house_value (target, last column)
```

```
    rows = []
    for r in reader:
        # convert all fields to float where possible; skip rows with conversion p
        try:
            # read by index positions to match your dataset structure
            total_rooms = float(r[fieldnames[1]])
            total_bedrooms = float(r[fieldnames[2]])
            population = float(r[fieldnames[3]])
            households = float(r[fieldnames[4]])
            median_income = float(r[fieldnames[5]])
            ocean_prox = float(r[fieldnames[6]])
            median_house_value = float(r[fieldnames[-1]])
        except Exception as e:
            # skip malformed rows
            continue

        # compute normalized predictors (avoid divide-by-zero)
        if households > 0:
            rooms_per_household = total_rooms / households
            bedrooms_per_household = total_bedrooms / households
        else:
            rooms_per_household = float("nan")
            bedrooms_per_household = float("nan")

        rows.append({
            'median_age': float(r[fieldnames[0]]),
            'total_rooms': total_rooms,
            'total_bedrooms': total_bedrooms,
            'population': population,
            'households': households,
            'median_income': median_income,
            'ocean_prox': ocean_prox,
            'median_house_value': median_house_value,
            'rooms_per_household': rooms_per_household,
            'bedrooms_per_household': bedrooms_per_household
        })

print("Loaded {} rows (after cleaning)".format(len(rows)))

# ---- Build predictor list (use normalized versions for predictors 2 & 3) ----
# We'll use these seven predictors (in original order, but swapping 2 & 3 with ra
predictor_names = [
    'median_age',               # predictor 1
    'rooms_per_household',      # predictor 2 normalized
    'bedrooms_per_household',   # predictor 3 normalized
    'population',               # predictor 4
    'households',               # predictor 5
    'median_income',            # predictor 6
    'ocean_prox'                # predictor 7
]
```

```
target_name = 'median_house_value'

# ---- Extract arrays (filter NaNs) ----
def column_array(rows, name):
    arr = np.array([row[name] for row in rows], dtype=float)
    # mask out nans
    mask = ~np.isnan(arr)
    return arr[mask]

# Use common mask to ensure equal lengths? For univariate regressions we allow va
# but median_house_value must exist
y_all = column_array(rows, target_name)

results = []  # will hold tuples (predictor, r, r2, beta, intercept)

for pname in predictor_names:
    x = column_array(rows, pname)
    # Need y aligned to same rows as x: recreate pairwise arrays
    # Build arrays by scanning rows and skipping rows where either is nan
    xs = []
    ys = []
    for row in rows:
        xv = row[pname]
        yv = row[target_name]
        if (xv is None) or (yv is None):
            continue
        if math.isnan(xv) or math.isnan(yv):
            continue
        xs.append(xv)
        ys.append(yv)
    if len(xs) < 5:
        # not enough data to compute stable metrics
        continue
    xs = np.array(xs, dtype=float)
    ys = np.array(ys, dtype=float)

    # Pearson r
    if np.std(xs) == 0 or np.std(ys) == 0:
        r = 0.0
    else:
        r = np.corrcoef(xs, ys)[0,1]
    r2 = r**2

    # Simple linear regression slope and intercept via polyfit
    beta, intercept = np.polyfit(xs, ys, 1)

    results.append((pname, r, r2, beta, intercept, xs, ys))

# ---- Sort by R^2 and print a compact report ----
results_sorted = sorted(results, key=lambda t: t[2], reverse=True)
```

```
print("\nUnivariate linear regression summary (predictor, r, R^2, slope (betas)):
for pname, r, r2, beta, intercept, xs, ys in results_sorted:
    print(f"{pname:22s}  r = {r: .4f}   R^2 = {r2: .4f}   slope = {beta: .4f}")

best = results_sorted[0]
best_name, best_r, best_r2, best_beta, best_intercept, best_xs, best_ys = best
print("\nMost predictive single predictor (by R^2):", best_name, f"(R^2={best_r2:

# ---- Plotting: scatter + regression line for each predictor, annotate r/R^2/bet
n = len(results_sorted)
cols = 2
rows_plots = int(np.ceil(n / cols))
fig, axes = plt.subplots(rows_plots, cols, figsize=(12, 4 * rows_plots))
axes = axes.flatten()

for ax_i, res in enumerate(results_sorted):
    pname, r, r2, beta, intercept, xs, ys = res
    ax = axes[ax_i]
    ax.scatter(xs, ys, alpha=0.3, s=10)
    # regression line
    x_line = np.linspace(np.nanmin(xs), np.nanmax(xs), 100)
    y_line = beta * x_line + intercept
    ax.plot(x_line, y_line, linestyle='--')
    ax.set_xlabel(pname)
    ax.set_ylabel("median_house_value")
    ax.set_title(f"{pname}  (r={r:.3f}, R2={r2:.3f})")
    # annotate slope
    ax.text(0.02, 0.95, f"slope={beta:.3f}", transform=ax.transAxes, va='top')

# Turn off any empty subplots
for j in range(len(results_sorted), len(axes)):
    fig.delaxes(axes[j])

plt.tight_layout()
plt.show()

# Question 4 code below
# Multiple regression: all predictors together

from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

# Build a filtered list of rows that have no NaNs for any predictor or target
valid_rows = []
for row in rows:
    ok = True
    for name in predictor_names + [target_name]:
        v = row.get(name, None)
        if v is None or (isinstance(v, float) and math.isnan(v)):
            ok = False
            break
```

```
    if ok:
        valid_rows.append(row)

print(f"Rows usable for multiple regression: {len(valid_rows)}")

# Construct X (n x p) and y (n,)
X = np.array([[r[p] for p in predictor_names] for r in valid_rows], dtype=float)
y = np.array([r[target_name] for r in valid_rows], dtype=float)

# Fit linear regression (ordinary least squares)
model = LinearRegression(fit_intercept=True)
model.fit(X, y)
y_pred = model.predict(X)

# Metrics you requested
# Pearson correlation between observed and predicted (r)
if np.std(y) == 0 or np.std(y_pred) == 0:
    r_multi = 0.0
else:
    r_multi = np.corrcoef(y, y_pred)[0,1]

r2_multi = r2_score(y, y_pred)  # explained variance (R^2)

# Print results: r, R^2, and betas (coefficients)
print("\n--- Multiple regression results (all predictors) ---")
print(f"Pearson r (observed vs predicted): {r_multi:.4f}")
print(f"Explained variance R^2:           {r2_multi:.4f}")
print(f"Intercept:                        {model.intercept_:.4f}")
print("Betas (coefficients) by predictor:")
for name, coef in zip(predictor_names, model.coef_):
    print(f"  {name:22s}  {coef: .6f}")

# (Optional) If you want adjusted R^2:
n = X.shape[0]
p = X.shape[1]
if n > p + 1:
    adj_r2 = 1 - (1 - r2_multi) * (n - 1) / (n - p - 1)
    print(f"Adjusted R^2:                     {adj_r2:.4f}")
else:
    print("Adjusted R^2:                     n <= p+1, not defined.")
```