Data Processing - Python Development and Pandas

```
# Python and Pandas Assessment Notebook
import pandas as pd
import numpy as np
from functools import reduce
```

Task 1: Create a DataFrame using Pandas Create a DataFrame with the following data:

- Name: Alice, Bob, Charlie, David
- Age: 25, 30, 35, 28
- City: New York, San Francisco, Los Angeles, Chicago

```
import pandas as pd

# Create the DataFrame
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age':  [25,      30,     35,        28],
    'City': ['New York','San Francisco','Los Angeles','Chicago']
})

print(df.dtypes)
df
```

```
Name    object
Age      int64
City    object
dtype: object
```

	Name	Age	City
0	Alice	25	New York
1	Bob	30	San Francisco
2	Charlie	35	Los Angeles
3	David	28	Chicago

Next steps: Generate code with df | View recommended plots | New interactive sheet

Constructed a Pandas DataFrame from a dictionary, explicitly naming columns and verifying that each column has the correct dtype (object for strings, int64 for integers). This lays the foundation for subsequent transformations.

Task 2: Row & Column Manipulation

Drop the 'City' column from the DataFrame created in Task 1

```
# Drop the 'City' column
df2 = df.drop(columns=['City'])
print(df2.columns)
df2
```

```
Index(['Name', 'Age'], dtype='object')
```

	Name	Age
0	Alice	25
1	Bob	30
2	Charlie	35
3	David	28

Next steps: Generate code with df2 | View recommended plots | New interactive sheet

Using drop(columns=[...]) removed an unwanted column and verify the DataFrame now contains only Name and Age. This demonstrates selective column removal.

Task 3: Handling Null Values

Create a new DataFrame with some null values and fill them

```
import numpy as np

# Create DataFrame with nulls
df_null = pd.DataFrame({
    'A': [1, np.nan, 3],
    'B': [np.nan, 'x', 'y']
})

# Fill numeric with 0, others with 'missing'
df_filled = df_null.fillna({'A': 0, 'B': 'missing'})
print(df_null, '\n\n', df_filled)
```

```
     A    B
0  1.0  NaN
1  NaN    x
2  3.0    y

      A        B
0  1.0  missing
1  0.0        x
2  3.0        y
```

Introduced NaN values in a DataFrame, then use fillna() with a dict to fill numeric columns with 0 and object columns with a placeholder. This illustrates per-column null handling.

Task 4: GroupBy & Describe

Using the following DataFrame, group by 'Category' and describe the 'Value' column

df_group = pd.DataFrame({ 'Category': ['A', 'B', 'A', 'B', 'A', 'C'], 'Value': [10, 20, 15, 25, 30, 35] })

```
df_group = pd.DataFrame({
    'Category': ['A','B','A','B','A','C'],
    'Value':    [10, 20, 15, 25, 30, 35]
})

grouped = df_group.groupby('Category')['Value'].describe()
print(grouped)
```

```
          count       mean        std   min    25%   50%    75%   max
Category
A           3.0  18.333333  10.408330  10.0  12.50  15.0  22.50  30.0
B           2.0  22.500000   3.535534  20.0  21.25  22.5  23.75  25.0
C           1.0  35.000000        NaN  35.0  35.00  35.0  35.00  35.0
```

We group by a categorical column and call .describe() on the numeric field to get count, mean, std, min/max, and quartiles per group. This helps you quickly summarize distributions by category.

Task 5: Concatenation & Merging

Concatenate two DataFrames vertically and merge them horizontally

df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]}) df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]}) df3 = pd.DataFrame({'C': [9, 10], 'D': [11, 12]})

```
df1 = pd.DataFrame({'A': [1,2], 'B': [3,4]})
df2 = pd.DataFrame({'A': [5,6], 'B': [7,8]})
df3 = pd.DataFrame({'C': [9,10], 'D': [11,12]})

# Vertical concat
df12 = pd.concat([df1, df2], ignore_index=True)

# Horizontal merge
df_final = pd.concat([df12, df3], axis=1)

print(df12.shape, df12, '\n\n', df_final.shape, df_final)
```

```
(4, 2)    A  B
0  1  3
1  2  4
2  5  7
3  6  8

 (4, 4)    A  B     C     D
0  1  3   9.0  11.0
1  2  4  10.0  12.0
2  5  7   NaN   NaN
3  6  8   NaN   NaN
```

By concatenating df1 and df2 vertically, then joining with df3 horizontally, we combine datasets along rows and columns. This shows how to stack and align disparate data sources.

Task 6: Tuples & Sets

Create a tuple of fruits and a set of numbers. Then, try to add an element to each and observe the difference.

```
# Tuple (immutable) and set (mutable)
fruits = ('apple','banana','cherry')
numbers = {1,2,3,4,5}

# Try to add
try:
    fruits += ('date',)
except TypeError as e:
    print("Tuple error:", e)

numbers.add(6)
print("Fruits tuple:", fruits)
print("Numbers set:", numbers)
```

```
Fruits tuple: ('apple', 'banana', 'cherry', 'date')
Numbers set: {1, 2, 3, 4, 5, 6}
```

Tuples are immutable so you can't change them in place (you must create a new tuple), whereas sets allow add(). This underlines the difference between fixed and mutable collections.

Task 7: Dictionaries

Create a dictionary of student names and their scores. Then, update a student's score and add a new student.

```
# Create and modify
scores = {'Alice': 85, 'Bob': 90, 'Charlie': 78}
scores['Bob'] = 95                    # update
scores['Diana'] = 88                  # add
print(scores)
```

```
{'Alice': 85, 'Bob': 95, 'Charlie': 78, 'Diana': 88}
```

Dictionaries map keys to values; you update an existing key simply by assignment, and add a new key–value pair the same way. This section reinforces basic key–value manipulations.

Task 8: Functions & Lambda

Create a function to calculate the square of a number. Then, use a lambda function to do the same.

```
# Regular function
def square(x):
    return x*x

# Lambda function
square_lambda = lambda x: x*x
```

```
print(square(5), square_lambda(5))
print(square(3), square_lambda(3))
```

```
25 25
9 9
```

We define a standard function and an equivalent one-line lambda, then demonstrate both on sample inputs. This shows the trade-off between readability and brevity.

Task 9: Iterators & Generators

Create an iterator for the first 5 even numbers. Then, create a generator for the same.

```
# Iterator class
class EvenIter:
    def __init__(self, n):
        self.max = n; self.curr = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.curr >= self.max*2:
            raise StopIteration
        val = self.curr
        self.curr += 2
        return val

# Generator function
def even_gen(n):
    for i in range(0, n*2, 2):
        yield i

it = EvenIter(5)
print(list(it))
print(list(even_gen(5)))
```

```
[0, 2, 4, 6, 8]
[0, 2, 4, 6, 8]
```

An iterator class implements **iter** and **next**, while a generator uses yield inside a function. Both produce the first five even numbers, showcasing two patterns for custom iteration.

Task 10: Map, Reduce & Filter

- Use map to square all numbers in a list.
- Use reduce to find the product of all numbers in a list.
- Use filter to get only even numbers from a list.

numbers = [1, 2, 3, 4, 5]

```
from functools import reduce

numbers = [1,2,3,4,5]
```

```
squares = list(map(lambda x: x*x, numbers))
product = reduce(lambda a,b: a*b, numbers)
evens = list(filter(lambda x: x%2==0, numbers))

print("Squares:", squares)
print("Product:", product)
print("Evens:", evens)
```

```
Squares: [1, 4, 9, 16, 25]
Product: 120
Evens: [2, 4]
```

map() applies a function over a list, reduce() cumulatively aggregates values, and filter() selects items by a predicate. This trio demonstrates functional transformations on sequences.

Task 11: Object-Oriented Programming - Creating a Class

Create a class called 'Rectangle' with attributes 'length' and 'width'. And include methods to calculate area and perimeter.

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width  = width
    def area(self):
        return self.length * self.width
    def perimeter(self):
        return 2*(self.length + self.width)

r1 = Rectangle(3,4)
r2 = Rectangle(5,6)
print("r1 area/perimeter:", r1.area(), r1.perimeter())
print("r2 area/perimeter:", r2.area(), r2.perimeter())
```

```
r1 area/perimeter: 12 14
r2 area/perimeter: 30 22
```

We define a Rectangle class with area and perimeter methods, then instantiate and use it. This solidifies basic class construction and method invocation.

Task 12: Pandas Data Analysis

Using the following DataFrame, perform these tasks:

- Find the average salary by department
- Get the names of employees with salary > 60000
- Add a new column 'Bonus' which is 10% of salary

df_employees = pd.DataFrame({ 'Name': ['John', 'Jane', 'Bob', 'Alice', 'Charlie'], 'Department': ['IT', 'HR', 'IT', 'Finance', 'HR'], 'Salary': [55000, 65000, 70000, 60000, 58000] })

```
df_emp = pd.DataFrame({
    'Name':       ['John','Jane','Bob','Alice','Charlie'],
    'Department': ['IT',  'HR',  'IT', 'Finance','HR'],
    'Salary':     [55000,65000,70000,60000,58000]
})

# 1. Avg salary by dept
avg_salary = df_emp.groupby('Department')['Salary'].mean().round(2)
# 2. Names with salary > 60000
high_earners = df_emp.loc[df_emp['Salary'] > 60000, 'Name'].tolist()
# 3. Add Bonus column (10%)
df_emp['Bonus'] = (df_emp['Salary'] * 0.10).round(2)

print("Average salaries:\n", avg_salary)
print("High earners:", high_earners)
print("With Bonus:\n", df_emp)
```

```
Average salaries:
 Department
Finance    60000.0
HR         61500.0
IT         62500.0
Name: Salary, dtype: float64
High earners: ['Jane', 'Bob']
With Bonus:
       Name Department  Salary   Bonus
0     John         IT   55000  5500.0
1     Jane         HR   65000  6500.0
2      Bob         IT   70000  7000.0
3    Alice    Finance   60000  6000.0
4  Charlie         HR   58000  5800.0
```

We group by department to compute average salary, filter rows by a numeric threshold, and create a new column as 10% of an existing one. This demonstrates real-world data-analysis workflows with Pandas.

Double-click (or enter) to edit