**Operations with NumPy arrays**

The following blocks of code create new lists given input lists input\_list1, input\_list2, input\_list3 (you can check their values in the console). If you had analogous NumPy arrays with the same values input\_array1, input\_array2, input\_array3 (you can check their values in the console), how would you create similar output as NumPy arrays using the knowledge on broadcasting, accessing element in NumPy arrays, and performing element-wise operations?

**Block 1**

list(map(lambda x: [5\*i for i in x], input\_list1)

**Block 2**

list(filter(lambda x: x % 2 == 0), input\_list2)

**Block 3**

[[i\*i for i in j] for j in input\_list3]

**Instructions 1/3**

**0 XP**

**Instructions 1/3**

**0 XP**

* [1](javascript:void(0))
  + Substitute the code in the block 1 given the input\_array1.

# Substitute the code in the block 1 given the input\_array1

output\_array1 = 5 \* input\_array1

print(list(map(lambda x: [5\*i for i in x], input\_list1)))

print(output\_array1)

* Substitute the code in the block 2 given input\_array2.

# Substitute the code in the block 2 given the input\_array2

output\_array2 = input\_array2 [input\_array2 % 2 ==0 ]

print(list(filter(lambda x: x % 2 == 0, input\_list2)))

print(output\_array2)

* Substitute the code in the block 3 given input\_array3.

# Substitute the code in the block 3 given the input\_array3

output\_array3 = input\_array3 \* input\_array3

print([[i\*i for i in j] for j in input\_list3])

print(output\_array3)

<script.py> output:

[[5, 10, 15], [20, 25, 30], [35, 40, 45]]

[[ 5 10 15]

[20 25 30]

[35 40 45]]

<script.py> output:

[0, 2, 4, 6, 8]

[0 2 4 6 8]

<script.py> output:

[[1, 4], [9, 16], [25, 36]]

[[ 1 4]

[ 9 16]

[25 36]]

Well done! Of course, decision to use a list or a NumPy array depends on a concrete task. As you can see, Python is very flexible for both scenarios. But if you need performance and scalability, then NumPy arrays is the choice!

**Simple use of .apply()**

Let's get some handful experience with .apply()!

You are given the full scores dataset containing students' performance as well as their background information.

Your task is to define the prevalence() function and apply it to the groups\_to\_consider columns of the scores DataFrame. This function should retrieve the most prevalent group/category for a given column (*e.g.* if the most prevalent category in the lunch column is standard, then prevalence() should return standard).

The reduce() function from the functools module is already imported.

Tip: pd.Series is an Iterable object. Therefore, you can use standard operations on it.

**Instructions**

**100 XP**

* Create a tuple list with unique items from passed object series and their counts.
* Extract a tuple with the highest counts using reduce().
* Return the item with the highest counts.
* Apply the prevalence function on the scores DataFrame using columns specified in groups\_to\_consider.

def prevalence(series):

vals = list(series)

# Create a tuple list with unique items and their counts

itms = [(x, vals.count(x)) for x in set(series)]

# Extract a tuple with the highest counts using reduce()

res = reduce(lambda x, y: x if x[1] > y[1] else y, itms)

# Return the item with the highest counts

return res[0]

# Apply the prevalence function on the scores DataFrame

result = scores[groups\_to\_consider].apply(prevalence)

print(result)

In [5]: def prevalence(series):

vals = list(series)

# Create a tuple list with unique items and their counts

itms = [(x, vals.count(x)) for x in set(series)]

# Extract a tuple with the highest counts using reduce()

res = reduce(lambda x, y: x if x[1] > y[1] else y, itms)

# Return the item with the highest counts

return res[0]

# Apply the prevalence function on the scores DataFrame

result = scores[groups\_to\_consider].apply(prevalence)

print(result)

gender female

race/ethnicity group C

parental level of education some college

lunch standard

test preparation course none

dtype: object

Very well! Actually, we can use the mode() method instead defining the prevalence() function and using it instead of the apply() method (scores[groups\_to\_consider].mode()). However, it's always good to practice several approaches.

**Additional arguments**

Let's use additional arguments in the .apply() method!

Your task is to create two new columns in scores:

* mean is the row-wise mean value of the math score, reading score and writing score
* rank defines how high the mean score is:
  + 'high' if the mean value >90>90
  + 'medium' if the mean value >60>60 but ≤≤ 90
  + 'low' if the mean value ≤60≤60

To accomplish this task, you'll need to define the function rank that, given a series, returns a list with two values: the mean of the series and a string defined by the aforementioned rule.

The module numpy is already imported for you as np.

**Instructions**

**100 XP**

* Calculate the mean of the input series.
* Return the mean and its rank as a list.
* Insert the output of rank() into new columns of scores.

def rank(series):

# Calculate the mean of the input series

mean = np.mean(series)

# Return the mean and its rank as a list

if mean > 90:

return [mean, 'high']

if mean > 60:

return [mean, 'medium']

return [mean, 'low']

# Insert the output of rank() into new columns of scores

cols = ['math score', 'reading score', 'writing score']

scores[['mean', 'rank']] = scores[cols].apply(rank, axis=1,

result\_type='expand')

print(scores[['mean', 'rank']].head())

<script.py> output:

mean rank

0 80.666667 medium

1 48.666667 low

2 47.666667 low

3 91.666667 high

4 82.333333 medium

Good job! The .apply() method is very useful when you need to create new columns in your DataFrame.

**Functions with additional arguments**

Let's add some arguments to the function definition!

Numeric data in scores represent students' performance scaled between 0 and 100. Your task is to rescale this data to an arbitrary range between low and high. Rescaling should be done in a linear fashion, *i.e.* for any data point xx in a column:

xnewxnew = xhigh−low100+lowxhigh−low100+low

To do rescaling, you'll have to define the function rescale(). Remember, the operation written above can be applied to Series directly. After defining the function, you'll have to apply it to the specified columns of scores.

**Instructions 1/2**

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* Define the expression to rescale input series.
* Rescale the data in cols to lie between 1 and 10.

def rescale(series, low, high):

# Define the expression to rescale input series

return series \*((high - low) / 100) + low

# Rescale the data in cols to lie between 1 and 10

cols = ['math score', 'reading score', 'writing score']

scores[cols] = scores[cols].apply(rescale , args = [1 , 10])

print(scores[cols].head())

<script.py> output:

math score reading score writing score

0 7.66 8.74 8.38

1 4.96 5.41 5.77

2 5.86 5.14 4.87

3 8.92 9.55 9.28

4 8.65 8.29 8.29

**Instructions 2/2**

**50 XP**

* [2](javascript:void(0))
* Redefine the function to accept keyword arguments with 0 and 100 as the default lower and upper limit, respectively.
* Rescale the data in cols to lie between 1 and 10.

# Redefine the function to accept keyword arguments

def rescale(series, low=0, high=100):

return series \* (high - low)/100 + low

# Rescale the data in cols to lie between 1 and 10

cols = ['math score', 'reading score', 'writing score']

scores[cols] = scores[cols].apply(rescale, low=1, high=10)

print(scores[cols].head())

<script.py> output:

math score reading score writing score

0 7.66 8.74 8.38

1 4.96 5.41 5.77

2 5.86 5.14 4.87

3 8.92 9.55 9.28

4 8.65 8.29 8.29

Wow, that was a lot work with the .apply() method! Congratulations!

**Standard DataFrame methods**

You are given the diabetes dataset storing information on female patients tested for diabetes. You will focus on blood glucose levels and the test results. Subjects, tested positively, usually have higher blood glucose levels after performing the so-called glucose tolerance test. Your task is to investigate whether it is true for this specific dataset.

The plasma glucose column corresponds to the glucose levels. The test result column corresponds to the diabetes test results.

You must use standard DataFrame methods (the numpy module is not imported for you).

**Instructions**

**100 XP**

* Load the data from the diabetes.csv file.
* Calculate the mean glucose level in the entire dataset.
* Group the data according to the diabetes test results.
* Calculate the mean glucose levels per group.

import numpy as np

import pandas as pd

# Load the data from the diabetes.csv file

diabetes = pd.read\_csv('diabetes.csv')

print(diabetes.info())

# Calculate the mean glucose level in the entire dataset

print(diabetes['plasma glucose'].mean())

# Group the data according to the diabetes test results

diabetes\_grouped = diabetes.groupby('test result')

# Calculate the mean glucose levels per group

print(diabetes\_grouped['plasma glucose'].mean())

<script.py> output:

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 768 entries, 0 to 767

Data columns (total 9 columns):

n pregnant 768 non-null int64

plasma glucose 763 non-null float64

blood pressure 733 non-null float64

skin thickness 541 non-null float64

serum insulin 394 non-null float64

bmi 757 non-null float64

pedigree function 768 non-null float64

age 768 non-null int64

test result 768 non-null object

dtypes: float64(6), int64(2), object(1)

memory usage: 54.1+ KB

None

121.6867627785059

test result

negative 110.643863

positive 142.319549

Name: plasma glucose, dtype: float64

Good job! As it was expected, the subjects tested positive for diabetes have higher blood glucose levels.

**BMI of villains**

Let's return to the heroes dataset containing the information on different comic book heroes. We added a bmi column to the dataset calculated as Weight divided by (Height/100)\*\*2. This index helps define whether an individual has weight problems.

Your task is to find out what is the mean value and standard deviation of the BMI index depending on the character's 'Alignment' and the 'Publisher' whom this character belongs to. However, you'll need to consider only those groups that have more than 10 valid observations of the BMI index.

Tip: use .count() to calculate the number of valid observations.

**Instructions**

**100 XP**

* Group the data by the two factors specified above.
* Filter groups having more than 10 valid bmi observations.
* Group the filtered data again by the same factors.
* Calculate the mean and standard deviation of the BMI index.

import numpy as np

# Group the data by two factors specified in the context

groups = heroes.groupby(['Publisher', 'Alignment'])

# Filter groups having more than 10 valid bmi observations

fheroes = groups.filter(lambda x: x['bmi'].count() > 10)

# Group the filtered data again by the same factors

fgroups = fheroes.groupby(['Publisher', 'Alignment'])

# Calculate the mean and standard deviation of the BMI index

result = fgroups[['bmi']].agg([np.mean, np.std])

print(result)

<script.py> output:

bmi

mean std

Publisher Alignment

DC Comics bad 71.460251 250.642060

good 25.028215 6.879312

Marvel Comics bad 62.512261 258.935100

good 28.968076 16.550686

Magnificent! Interestingly, villains have quite a big BMI index in average (an individual is considered overweight when it is > 30), with a considerable standard deviation. Good characters are pretty normal (BMI index between 25 and 30 is considered normal).

**NaN value imputation**

Let's try to impute some values. In the previous task you created a DataFrame fheroes where all the groups with insufficient amount of bmi observations were removed. Our bmi column has a lot of missing values (NaNs) though. Given two copies of the fheroes DataFrame (imp\_globmean and imp\_grpmean), your task is to impute the NaNs in the bmi column with the overall mean value and with the mean value per group defined by Publisher and Alignment factors, respectively.

Tip: pandas Series and NumPy arrays have a special .fillna() method which substitutes all the encountered NaNs with a value specified as an argument.

**Instructions**

**100 XP**

* Define a lambda function that imputes NaN values in series.
* Impute NaNs in the bmi column of imp\_globmean with the overall mean value.
* Impute NaNs in the bmi column of imp\_grpmean with the mean value per group.

# Define a lambda function that imputes NaN values in series

impute = lambda series: series.fillna(np.mean(series))

# Impute NaNs in the bmi column of imp\_globmean

imp\_globmean['bmi'] = imp\_globmean['bmi'].transform(impute)

print("Global mean = " + str(fheroes['bmi'].mean()) + "\n")

groups = imp\_grpmean.groupby(['Publisher', 'Alignment'])

# Impute NaNs in the bmi column of imp\_grpmean

imp\_grpmean['bmi'] = groups['bmi'].transform(impute)

print(groups['bmi'].mean())

**My Code OP :**

<script.py> output:

Global mean = 38.98033229674414

Publisher Alignment

DC Comics bad 71.460251

good 25.028215

Marvel Comics bad 62.512261

good 28.968076

Name: bmi, dtype: float64

**Ans Code OP :**

<script.py> output:

Global mean = 38.98033229674414

Publisher Alignment

DC Comics bad 71.460251

good 25.028215

Marvel Comics bad 62.512261

good 28.968076

Name: bmi, dtype: float64

**(Same OP diff code)**

Great! Imputation was successful! Grouping the data is a very useful technique required in almost any study dealing with complex datasets. Therefore, it is good to keep practicing!

**Explore feature relationships**

Let's do some exploratory analysis. You are already familiar with the retinol dataset. It stores the information on the relationship between different factors (background information, clinical data) and the levels of retinol (plasma retinol) and beta-carotene (plasma B-carotene) in blood plasma. Low levels of these compounds were reported to be associated with higher risk of cancer.

Which **numeric** features may have strong linear relationship to the plasma retinol feature?

The matplotlib.pyplot module is already imported as plt.

**Instructions**

**50 XP**

**Possible Answers**

* 

Age 

bmi, calories, fat, and cholesterol

* 

diet retinol and diet B-carotene

* 

plasma B-carotene

* 

neither of the numeric features **(A)**

Correct. Neither of the numeric features show strong linear relationship with plasma retinol. This question serves as an example that interviewers might try to confuse you by making you try to find some regularity in the data, which is deliberately absent.

**Plot a histogram**

Let's further investigate the retinol dataset. Your task now is to create a histogram of the plasma retinol feature.

**Instructions 1/4**

**25 XP**

* [1](javascript:void(0))

Plot a simple histogram of the plasma retinol feature.

# Plot a simple histogram of the plasma retinol feature

import matplotlib.pyplot as plt

plt.hist(retinol['plasma retinol'])

plt.show()

Redefine the histogram to have 20 bins.

# Redefine the histogram to have 20 bins

plt.hist(retinol['plasma retinol'] , bins = 20)

plt.show()

Add a title to the plot (choose any name you want).

plt.hist(retinol['plasma retinol'], bins=20)

# Add a title to the plot

plt.title('Karthik')

plt.show()

Add other missing parts to the plot.

plt.hist(retinol['plasma retinol'], bins=20)

plt.title('Histogram of Plasma Retinol')

# Add other missing parts to the plot

plt.xlabel('X label')

plt.ylabel('Y label')

plt.show()

**Creating boxplots**

Let's get back to our heroes dataset. As we previously discovered, the BMI index is in average much higher for villains than for good characters (taking into account only Marvel and DC publishers). Your task is to plot the corresponding distributions of BMI indices using boxplots.

Tip: to select rows in a DataFrame, for which a specific column follows a certain condition, use this expression dataframe[condition for column\_name] (*e.g.* heroes[heroes['Alignment'] == 'good'] selects rows that have a 'good' Alignment in the heroes dataset).

**Instructions 1/3**

**35 XP**

* [1](javascript:void(0))

Create a boxplot of BMI indices for good and bad sides.

import seaborn as sns

# Create a boxplot of BMI indices for 'good' and 'bad' sides

sns.boxplot(x='Alignment', y='bmi', data=heroes)

plt.show()

Select rows from heroes for which the BMI index is smaller than 1000.

import seaborn as sns

# Select rows from heroes for which the BMI index < 1000

heroes\_filtered = heroes[heroes['bmi'] < 1000]

Create a new boxplot of BMI indices for good and bad sides using heroes\_filtered.

import seaborn as sns

# Select rows from 'heroes' for which the BMI index < 1000

heroes\_filtered = heroes[heroes['bmi'] < 1000]

# Create a new boxplot of BMI indices

sns.boxplot(x = "Alignment" , y = 'bmi' , data = heroes\_filtered)

plt.show()

Very good job! You saw that there are two outliers (Giganta and Utgard-Loki from Marvel comics) which affect the average value of BMI index for villains. If you filter them out, the villains are only slightly higher in their values of BMI index.