2/27/2023

Mathematics Association of Nairobi University
isaak@students.uonbi.ac.ke

INTRODUCTION TO DATA ANALYSIS WITH PANDAS

Pandas Series and DataFrames are designed for fast data analysis and manipulation, as well as being flexible and easy to use. Below are just a few features that makes Pandas an excellent package for data analysis:

1. Allows the use of labels for rows and columns
2. Can calculate rolling statistics on time series data
3. Easy handling of NaN values
4. Is able to load data of different formats into DataFrames
5. Can join and merge different datasets together
6. It integrates with NumPy and Matplotlib

For these and other reasons, Pandas DataFrames have become one of the most commonly used Pandas object for data analysis in Python.

Creating Pandas Series

Pandas Series

A Pandas series is a one-dimensional array-like object that can hold many data types, such as numbers or strings, and has an option to provide axis labels. Difference between NumPy ndarrays and Pandas Series

1. One of the main differences between Pandas Series and NumPy ndarrays is that you can assign an index label to each element in the Pandas Series. In other words, you can name the indices of your Pandas Series anything you want.
2. Another big difference between Pandas Series and NumPy ndarrays is that Pandas Series can hold data of different data types.

Let's start by importing Pandas into Python. It has become a convention to import Pandas as `pd`, therefore, you can import Pandas by typing the following command in your Jupyter notebook:

```
import pandas as pd
```

Let's begin by creating a Pandas Series. You can create Pandas Series by using the command `pd.Series(data, index)`, where index is a list of `index` labels. Let's use a Pandas Series to store a grocery list. We will use the food items as index labels and the quantity we need to buy of each item as our data.

Example 1 - Create a Series

In [26]:
```
# We import Pandas as pd into Python
import pandas as pd

# We create a Pandas Series that stores a grocery list
groceries = pd.Series(data = [30, 6, 'Yes', 'No'], index = ['eggs', 'apples', 'milk', 'bread'])

# We display the Groceries Pandas Series
groceries
```

Out[26]:
```
eggs       30
apples      6
milk      Yes
bread      No
dtype: object
```

We see that Pandas Series are displayed with the indices in the first column and the data in the second column. Notice that the data is not indexed 0 to 3 but rather it is indexed with the names of the food we put in, namely eggs, apples, etc... Also, notice that the data in our Pandas Series has both integers and strings.

Just like NumPy ndarrays, Pandas Series have attributes that allow us to get information from the series in an easy way. Let's see some of them:

Example 2 - Print attributes - shape, ndim,and size

In [3]:
```
# We print some information about Groceries
print('Groceries has shape:', groceries.shape)
print('Groceries has dimension:', groceries.ndim)
print('Groceries has a total of', groceries.size, 'elements')
```

```
Groceries has shape: (4,)
Groceries has dimension: 1
Groceries has a total of 4 elements
```

We can also print the index labels and the data of the Pandas Series separately. This is useful if you don't happen to know what the index labels of the Pandas Series are.

Example 3 - Print attributes - values, and index

In [4]:
```
# We print the index and data of Groceries
print('The data in Groceries is:', groceries.values)
print('The index of Groceries is:', groceries.index)
```

```
The data in Groceries is: [30 6 'Yes' 'No']
The index of Groceries is: Index(['eggs', 'apples', 'milk', 'bread'], dtype='object')
```

If you are dealing with a very large Pandas Series and if you are not sure whether an index label exists, you can check by using the in command

Example 4 - Check if an index is available in the given Series

In [5]:
```
# We check whether bananas is a food item (an index) in Groceries
x = 'bananas' in groceries

# We check whether bread is a food item (an index) in Groceries
y = 'bread' in groceries

# We print the results
print('Is bananas an index label in Groceries:', x)
print('Is bread an index label in Groceries:', y)
```

```
Is bananas an index label in Groceries: False
Is bread an index label in Groceries: True
```

Now let's look at how we can access or modify elements in a Pandas Series. One great advantage of Pandas Series is that it allows us to access data in many different ways. Elements can be accessed using index labels or numerical indices inside square brackets, [ ], similar to how we access elements in NumPy ndarrays. Since we can use numerical indices, we can use both positive and negative integers to access data from the beginning or from the end of the Series, respectively. Since we can access elements in various ways, in order to remove any ambiguity to whether we are referring to an index label or numerical index, Pandas Series have two attributes, `.loc` and `.iloc` to explicitly state what we mean. The attribute `.loc` stands for location and it is used to explicitly state that we are using a labeled index. Similarly, the attribute `.iloc` stands for integer location and it is used to explicitly state that we are using a numerical index. Let's see some examples:

Example 1. Access elements using index labels

In [6]:
```
# We access elements in Groceries using index labels:

# We use a single index label
print('How many eggs do we need to buy:', groceries['eggs'])
print()

# we can access multiple index labels
print('Do we need milk and bread:\n', groceries[['milk', 'bread']])
print()

# we use loc to access multiple index labels
print('How many eggs and apples do we need to buy:\n', groceries.loc[['eggs', 'apples']])
print()

# We access elements in Groceries using numerical indices:

# we use multiple numerical indices
print('How many eggs and apples do we need to buy:\n',  groceries[[0, 1]])
print()

# We use a negative numerical index
print('Do we need bread:\n', groceries[[-1]])
```

```
print()

# We use a single numerical index
print('How many eggs do we need to buy:', groceries[0])
print()
# we use iloc to access multiple numerical indices
print('Do we need milk and bread:\n', groceries.iloc[[2, 3]])
```

```
How many eggs do we need to buy: 30

Do we need milk and bread:
 milk     Yes
bread     No
dtype: object

How many eggs and apples do we need to buy:
 eggs      30
apples     6
dtype: object

How many eggs and apples do we need to buy:
 eggs      30
apples     6
dtype: object

Do we need bread:
 bread    No
dtype: object

How many eggs do we need to buy: 30

Do we need milk and bread:
 milk     Yes
bread     No
dtype: object
```

In [7]:
```
# Example 2. Mutate elements using index labels

# We display the original grocery list
print('Original Grocery List:\n', groceries)

# We change the number of eggs to 2
groceries['eggs'] = 2

# We display the changed grocery list
print()
print('Modified Grocery List:\n', groceries)
```

```
Original Grocery List:
 eggs       30
apples      6
milk      Yes
bread      No
dtype: object

Modified Grocery List:
 eggs        2
apples      6
milk      Yes
bread      No
dtype: object
```

We can also delete items from a Pandas Series by using the `.drop()` method. The `Series.drop(label)` method removes the given label from the given Series. We should note that the `Series.drop(label)` method drops elements from the Series out-of-place, meaning that it doesn't change the original Series being modified. Let's see how this works:

Example 3. Delete elements out-of-place using `drop()`

In [8]:
```
# We display the original grocery list
print('Original Grocery List:\n', groceries)

# We remove apples from our grocery list. The drop function removes elements out of place
print()
print('We remove apples (out of place):\n', groceries.drop('apples'))

# When we remove elements out of place the original Series remains intact. To see this
# we display our grocery list again
print()
print('Grocery List after removing apples out of place:\n', groceries)
```

```
Original Grocery List:
 eggs        2
apples       6
milk       Yes
bread       No
dtype: object

We remove apples (out of place):
 eggs       2
milk      Yes
bread      No
dtype: object

Grocery List after removing apples out of place:
 eggs        2
apples       6
milk       Yes
bread       No
dtype: object
```

Arithmetic Operations on Pandas Series

In [9]:
```
# We create a Pandas Series that stores a grocery list of just fruits
fruits= pd.Series(data = [10, 6, 3,], index = ['apples', 'oranges', 'bananas'])

# We display the fruits Pandas Series
fruits
```

Out[9]:
```
apples     10
oranges     6
bananas     3
dtype: int64
```

In [10]:
```
# Example 1. Element-wise basic arithmetic operations

# We print fruits for reference
print('Original grocery list of fruits:\n ', fruits)

# We perform basic element-wise operations using arithmetic symbols
print()
print('fruits + 2:\n', fruits + 2) # We add 2 to each item in fruits
print()
print('fruits - 2:\n', fruits - 2) # We subtract 2 to each item in fruits
print()
print('fruits * 2:\n', fruits * 2) # We multiply each item in fruits by 2
print()
print('fruits / 2:\n', fruits / 2) # We divide each item in fruits by 2
print()
```

```
Original grocery list of fruits:
  apples     10
oranges     6
bananas     3
dtype: int64

fruits + 2:
 apples     12
oranges     8
bananas     5
dtype: int64

fruits - 2:
 apples     8
oranges    4
bananas    1
dtype: int64

fruits * 2:
 apples     20
oranges    12
bananas     6
dtype: int64

fruits / 2:
 apples     5.0
oranges    3.0
bananas    1.5
dtype: float64
```

In [11]:
```
# Example 2. Use mathematical functions from NumPy to operate on Series

# We import NumPy as np to be able to use the mathematical functions
import numpy as np

# We print fruits for reference
print('Original grocery list of fruits:\n', fruits)
```

```
# We apply different mathematical functions to all elements of fruits
print()
print('EXP(X) = \n', np.exp(fruits))
print()
print('SQRT(X) =\n', np.sqrt(fruits))
print()
print('POW(X,2) =\n',np.power(fruits,2)) # We raise all elements of fruits to the power of 2
```

```
Original grocery list of fruits:
 apples     10
oranges     6
bananas     3
dtype: int64

EXP(X) =
 apples     22026.465795
oranges      403.428793
bananas       20.085537
dtype: float64

SQRT(X) =
 apples     3.162278
oranges    2.449490
bananas    1.732051
dtype: float64

POW(X,2) =
 apples     100
oranges     36
bananas      9
dtype: int64
```

In [12]:
```
# Example 3. Perform arithmetic operations on selected elements

# We print fruits for reference
print('Original grocery list of fruits:\n ', fruits)
print()

# We add 2 only to the bananas
print('Amount of bananas + 2 = ', fruits['bananas'] + 2)
print()

# We subtract 2 from apples
print('Amount of apples - 2 = ', fruits.iloc[0] - 2)
print()

# We multiply apples and oranges by 2
print('We double the amount of apples and oranges:\n', fruits[['apples', 'oranges']] * 2)
print()

# We divide apples and oranges by 2
print('We half the amount of apples and oranges:\n', fruits.loc[['apples', 'oranges']] / 2)
```

```
Original grocery list of fruits:
  apples     10
oranges     6
bananas     3
dtype: int64

Amount of bananas + 2 =  5

Amount of apples - 2 =  8

We double the amount of apples and oranges:
 apples     20
oranges    12
dtype: int64

We half the amount of apples and oranges:
 apples     5.0
oranges    3.0
dtype: float64
```

In [13]:
```
#Example 4. Perform multiplication on a Series having integer and string elements

# We multiply our grocery list by 2
groceries * 2
```

Out[13]:
```
eggs           4
apples        12
milk      YesYes
bread       NoNo
dtype: object
```

As we can see, in this case, since we multiplied by `2`, Pandas doubles the data of each item including the strings. Pandas can do this because the multiplication operation `*` is defined both for numbers and strings. If you were to apply an operation that was valid for numbers but not strings, say for instance, `/` you will get an error. So when you have mixed data types in your Pandas Series make sure

the arithmetic operations are valid on all the data types of your elements.

Creating Pandas DataFrames

Pandas DataFrames are two-dimensional data structures with labeled rows and columns, that can hold many data types. If you are familiar with Excel, you can think of Pandas DataFrames as being similar to a spreadsheet. We can create Pandas DataFrames manually or by loading data from a file. In this lesson, we will start by learning how to create Pandas DataFrames manually from dictionaries, and later we will see how we can load data into a DataFrame from a data file.

Create a DataFrame manually

We will start by creating a DataFrame manually from a dictionary of Pandas Series. It is a two-step process:

1. The first step is to create the dictionary of Pandas Series.
2. After the dictionary is created we can then pass the dictionary to the `pd.DataFrame()` function.

We will create a dictionary that contains items purchased by two people, Alice and Bob, on an online store. The Pandas Series will use the price of the items purchased as data, and the purchased items will be used as the index labels to the Pandas Series. Let's see how this done in code:

In [14]:
```
# We import Pandas as pd into Python
import pandas as pd

# We create a dictionary of Pandas Series
items = {'Bob' : pd.Series(data = [245, 25, 55], index = ['bike', 'pants', 'watch']),
         'Alice' : pd.Series(data = [40, 110, 500, 45], index = ['book', 'glasses', 'bike', 'pants'])}

# We print the type of items to see that it is a dictionary
print(type(items))
```

```
<class 'dict'>
```

Now that we have a dictionary, we are ready to create a DataFrame by passing it to the `pd.DataFrame()` function. We will create a DataFrame that could represent the shopping carts of various users, in this case we have only two users, Alice and Bob.

Example 1. Create a DataFrame using a dictionary of Series.

In [15]:
```
# We create a Pandas DataFrame by passing it a dictionary of Pandas Series
shopping_carts = pd.DataFrame(items)

# We display the DataFrame
shopping_carts
```

Out[15]:

	Bob	Alice
bike	245.0	500.0
book	NaN	40.0
glasses	NaN	110.0
pants	25.0	45.0
watch	55.0	NaN

1. We see that DataFrames are displayed in tabular form, much like an Excel spreadsheet, with the labels of rows and columns in bold.
2. Also, notice that the row labels of the DataFrame are built from the union of the index labels of the two Pandas Series we used to construct the dictionary. And the column labels of the DataFrame are taken from the keys of the dictionary.
3. Another thing to notice is that the columns are arranged alphabetically and not in the order given in the dictionary. We will see later that this won't happen when we load data into a DataFrame from a data file.
4. The last thing we want to point out is that we see some `NaN` values appear in the DataFrame. `NaN` stands for Not a Number, and is Pandas way of indicating that it doesn't have a value for that particular row and column index. For example, if we look at the column of Alice, we see that it has NaN in the watch index. You can see why this is the case by looking at the dictionary we created at the beginning. We clearly see that the dictionary has no item for Alice labeled watches. So whenever a DataFrame is created, if a particular column doesn't have values for a particular row index, Pandas will put a NaN value there.
    A. If we were to feed this data into a machine learning algorithm we will have to remove these `NaN` values first. In a later lesson, we will learn how to deal with `NaN` values and clean our data. For now, we will leave these values in our DataFrame.

In the example above, we created a Pandas DataFrame from a dictionary of Pandas Series that had clearly defined indexes. If we don't provide index labels to the Pandas Series, Pandas will use numerical row indexes when it creates the DataFrame. Let's see an example:

Example 2. DataFrame assigns the numerical row indexes by default.

In [16]:
```
# We create a dictionary of Pandas Series without indexes
```

```
data = {'Bob' : pd.Series([245, 25, 55]),
        'Alice' : pd.Series([40, 110, 500, 45])}

# We create a DataFrame
df = pd.DataFrame(data)

# We display the DataFrame
df
```

Out[16]:

	Bob	Alice
0	245.0	40
1	25.0	110
2	55.0	500
3	NaN	45

We can see that Pandas indexes the rows of the DataFrame starting from 0, just like NumPy indexes ndarrays.

Now, just like with Pandas Series we can also extract information from DataFrames using attributes. Let's print some information from our `shopping_carts` DataFrame

Example 3. Demonstrate a few attributes of DataFrame

In [17]:
```
# We print some information about shopping_carts
print('shopping_carts has shape:', shopping_carts.shape)
print('shopping_carts has dimension:', shopping_carts.ndim)
print('shopping_carts has a total of:', shopping_carts.size, 'elements')
print()
print('The data in shopping_carts is:\n', shopping_carts.values)
print()
print('The row index in shopping_carts is:', shopping_carts.index)
print()
print('The column index in shopping_carts is:', shopping_carts.columns)
```

```
shopping_carts has shape: (5, 2)
shopping_carts has dimension: 2
shopping_carts has a total of: 10 elements

The data in shopping_carts is:
 [[245. 500.]
 [ nan  40.]
 [ nan 110.]
 [ 25.  45.]
 [ 55.  nan]]

The row index in shopping_carts is: Index(['bike', 'book', 'glasses', 'pants', 'watch'], dtype='object')

The column index in shopping_carts is: Index(['Bob', 'Alice'], dtype='object')
```

When creating the `shopping_carts` DataFrame we passed the entire dictionary to the `pd.DataFrame()` function. However, there might be cases when you are only interested in a subset of the data. Pandas allows us to select which data we want to put into our DataFrame by means of the keywords `columns` and `index`. Let's see some examples:

In [18]:
```
# We Create a DataFrame that only has Bob's data
bob_shopping_cart = pd.DataFrame(items, columns=['Bob'])

# We display bob_shopping_cart
bob_shopping_cart
```

Out[18]:

	Bob
bike	245
pants	25
watch	55

In [19]:
```
# Example 4. Selecting specific rows of a DataFrame

# We Create a DataFrame that only has selected items for both Alice and Bob
sel_shopping_cart = pd.DataFrame(items, index = ['pants', 'book'])

# We display sel_shopping_cart
sel_shopping_cart
```

Out[19]:

	Bob	Alice
pants	25.0	45
book	NaN	40

In [20]:
```
#Example 5. Selecting specific columns of a DataFrame
```

```
# We Create a DataFrame that only has selected items for Alice
alice_sel_shopping_cart = pd.DataFrame(items, index = ['glasses', 'bike'], columns = ['Alice'])

# We display alice_sel_shopping_cart
alice_sel_shopping_cart
```

Out[20]:

	Alice
glasses	110
bike	500

You can also manually create DataFrames from a dictionary of lists (arrays). The procedure is the same as before, we start by creating the dictionary and then passing the dictionary to the `pd.DataFrame()` function. In this case, however, all the lists (arrays) in the dictionary must be of the same length. Let' see an example:

Example 6. Create a DataFrame using a dictionary of lists

In [21]:
```
# We create a dictionary of lists (arrays)
data = {'Integers' : [1,2,3],
        'Floats' : [4.5, 8.2, 9.6]}

# We create a DataFrame
df = pd.DataFrame(data)

# We display the DataFrame
df
```

Out[21]:

	Integers	Floats
0	1	4.5
1	2	8.2
2	3	9.6

In [22]:
```
#Example 7. Create a DataFrame using a dictionary of lists, and custom row-indexes (labels)

# We create a dictionary of lists (arrays)
data = {'Integers' : [1,2,3],
        'Floats' : [4.5, 8.2, 9.6]}

# We create a DataFrame and provide the row index
df = pd.DataFrame(data, index = ['label 1', 'label 2', 'label 3'])

# We display the DataFrame
df
```

Out[22]:

	Integers	Floats
label 1	1	4.5
label 2	2	8.2
label 3	3	9.6

In [23]:
```
#Example 8. Create a DataFrame using a of list of dictionaries

# We create a list of Python dictionaries
items2 = [{'bikes': 20, 'pants': 30, 'watches': 35},
          {'watches': 10, 'glasses': 50, 'bikes': 15, 'pants':5}]

# We create a DataFrame
store_items = pd.DataFrame(items2)

# We display the DataFrame
store_items
```

Out[23]:

	bikes	pants	watches	glasses
0	20	30	35	NaN
1	15	5	10	50.0

In [24]:
```
#Example 9. Create a DataFrame using a of list of dictionaries, and custom row-indexes (labels)

# We create a list of Python dictionaries
items2 = [{'bikes': 20, 'pants': 30, 'watches': 35},
          {'watches': 10, 'glasses': 50, 'bikes': 15, 'pants':5}]

# We create a DataFrame  and provide the row index
store_items = pd.DataFrame(items2, index = ['store 1', 'store 2'])

# We display the DataFrame
store_items
```

Out[24]:

	bikes	pants	watches	glasses
store 1	20	30	35	NaN
store 2	15	5	10	50.0

In [ ]: