100 XP

Exercise - Explore data with NumPy and Pandas

10 minutes

Sandbox activated! Time remaining: **46 min**

You have used 1 of 10 sandboxes for today. More sandboxes will be available tomorrow.

learn-notebooks-defb0d0d-e550-4ae0-ae3b-be9... Py38_default

We would love to hear your feedback on the notebooks experience! Please take a few minutes to complete our survey.

Exploring Data with Python

A significant part of a a data scientist's role is to explore, analyze, and visualize data. There's a wide range of tools and programming languages that they can use to do this; and of the most popular approaches is to use Jupyter notebooks (like this one) and Python.

Python is a flexible programming language that is used in a wide range of scenarios; from web applications to device programming. It's extremely popular in the data science and machine learning community because of the many packages it supports for data analysis and visualization.

In this notebook, we'll explore some of these packages, and apply basic techniques to analyze data. This is not intended to be a comprehensive Python programming exercise; or even a deep dive into data analysis. Rather, it's intended as a crash course in some of the common ways in which data scientists can use Python to work with data.

> **Note**: If you've never used the Jupyter Notebooks environment before, there are a few things you should be aware of:
>
> - Notebooks are made up of *cells*. Some cells (like this one) contain *markdown* text, while others (like the one beneath this one) contain code.
> - You can run each code cell by using the ► **Run** button. the ► **Run** button will show up when you hover over the cell.

- The output from each code cell will be displayed immediately below the cell.
- Even though the code cells can be run individually, some variables used in the code are global to the notebook. That means that you should run all of the code cells **in order**. There may be dependencies between code cells, so if you skip a cell, subsequent cells might not run correctly.

Exploring data arrays with NumPy

Lets start by looking at some simple data.

Suppose a college takes a sample of student grades for a data science class.

Run the code in the cell below by clicking the ► **Run** button to see the data.

[1]
```
data = [50,50,47,97,49,3,53,42,26,74,82,62,37,15,70,27,36,35,48,52,63,64]
print(data)
```
✓ <1 sec

```
[50, 50, 47, 97, 49, 3, 53, 42, 26, 74, 82, 62, 37, 15, 70, 27, 36, 35, 48,
52, 63, 64]
```

The data has been loaded into a Python **list** structure, which is a good data type for general data manipulation, but not optimized for numeric analysis. For that, we're going to use the **NumPy** package, which includes specific data types and functions for working with *Num*bers in *Py*thon.

Run the cell below to load the data into a NumPy **array**.

[2]
```
import numpy as np

grades = np.array(data)
print(grades)
```
✓ 1 sec

```
[50 50 47 97 49  3 53 42 26 74 82 62 37 15 70 27 36 35 48 52 63 64]
```

Just in case you're wondering about the differences between a **list** and a NumPy **array**, let's compare how these data types behave when we use them in an expression that multiplies them by 2.

```
print (type(data),'x 2:', data * 2)
print('---')
```

```
print (type(grades),'x 2:', grades * 2)
```

[3] ✓ <1 sec

```
<class 'list'> x 2: [50, 50, 47, 97, 49, 3, 53, 42, 26, 74, 82, 62, 37, 15,
70, 27, 36, 35, 48, 52, 63, 64, 50, 50, 47, 97, 49, 3, 53, 42, 26, 74, 82, 62,
37, 15, 70, 27, 36, 35, 48, 52, 63, 64]
---
<class 'numpy.ndarray'> x 2: [100 100  94 194  98   6 106  84  52 148 164 124
74  30 140  54  72  70
  96 104 126 128]
```

Note that multiplying a list by 2 creates a new list of twice the length with the original sequence of list elements repeated. Multiplying a NumPy array on the other hand performs an element-wise calculation in which the array behaves like a *vector*, so we end up with an array of the same size in which each element has been multiplied by 2.

The key takeaway from this is that NumPy arrays are specifically designed to support mathematical operations on numeric data - which makes them more useful for data analysis than a generic list.

You might have spotted that the class type for the numpy array above is a **numpy.ndarray**. The **nd** indicates that this is a structure that can consists of multiple *dimensions* (it can have *n* dimensions). Our specific instance has a single dimension of student grades.

Run the cell below to view the **shape** of the array.

```
grades.shape
```

[4] ✓ <1 sec

```
(22,)
```

The shape confirms that this array has only one dimension, which contains 22 elements (there are 22 grades in the original list). You can access the individual elements in the array by their zero-based ordinal position. Let's get the first element (the one in position 0).

```
grades[0]
```

[5] ✓ <1 sec

```
50
```

Alright, now you know your way around a NumPy array, it's time to perform some analysis of the grades data.

You can apply aggregations across the elements in the array, so let's find the simple average grade (in other words, the *mean* grade value).

[6]
```
grades.mean()
```
✓ <1 sec

```
49.18181818181818
```

So the mean grade is just around 50 - more or less in the middle of the possible range from 0 to 100.

Let's add a second set of data for the same students, this time recording the typical number of hours per week they devoted to studying.

[7]
```
# Define an array of study hours
study_hours = [10.0,11.5,9.0,16.0,9.25,1.0,11.5,9.0,8.5,14.5,15.5,
               13.75,9.0,8.0,15.5,8.0,9.0,6.0,10.0,12.0,12.5,12.0]

# Create a 2D array (an array of arrays)
student_data = np.array([study_hours, grades])

# display the array
student_data
```
✓ <1 sec

```
array([[10.  , 11.5 ,  9.  , 16.  ,  9.25,  1.  , 11.5 ,  9.  ,  8.5 ,
        14.5 , 15.5 , 13.75,  9.  ,  8.  , 15.5 ,  8.  ,  9.  ,  6.  ,
        10.  , 12.  , 12.5 , 12.  ],
       [50.  , 50.  , 47.  , 97.  , 49.  ,  3.  , 53.  , 42.  , 26.  ,
        74.  , 82.  , 62.  , 37.  , 15.  , 70.  , 27.  , 36.  , 35.  ,
        48.  , 52.  , 63.  , 64.  ]])
```

Now the data consists of a 2-dimensional array - an array of arrays. Let's look at its shape.

[8]
```
# Show shape of 2D array
student_data.shape
```
✓ <1 sec

```
(2, 22)
```

The **student_data** array contains two elements, each of which is an array containing 22 elements.

To navigate this structure, you need to specify the position of each element in the hierarchy. So to find the first value in the first array (which contains the study hours data), you can use the following code.

```
[9]  # Show the first element of the first element
     student_data[0][0]
```
✓ <1 sec

```
10.0
```

Now you have a multidimensional array containing both the student's study time and grade information, which you can use to compare data. For example, how does the mean study time compare to the mean grade?

```
[10] # Get the mean value of each sub-array
     avg_study = student_data[0].mean()
     avg_grade = student_data[1].mean()

     print('Average study hours: {:.2f}\nAverage grade: {:.2f}'.format(avg_study
```
✓ <1 sec

```
Average study hours: 10.52
Average grade: 49.18
```

Exploring tabular data with Pandas

While NumPy provides a lot of the functionality you need to work with numbers, and specifically arrays of numeric values; when you start to deal with two-dimensional tables of data, the **Pandas** package offers a more convenient structure to work with - the **DataFrame**.

Run the following cell to import the Pandas library and create a DataFrame with three columns. The first column is a list of student names, and the second and third columns are the NumPy arrays containing the study time and grade data.

```
import pandas as pd

df_students = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan
                                     'Rhonda', 'Giovanni', 'Francesca', 'Raja
```

```
                                    'Jakeem','Helena','Ismat','Anila','Skye'
                                'StudyHours':student_data[0],
                                'Grade':student_data[1]})

df_students
```

[11] ✓ 3 sec

Note that in addition to the columns you specified, the DataFrame includes an *index* to unique identify each row. We could have specified the index explicitly, and assigned any kind of appropriate value (for example, an email address); but because we didn't specify an index, one has been created with a unique integer value for each row.

Finding and filtering data in a DataFrame

You can use the DataFrame's **loc** method to retrieve data for a specific index value, like this.

[12]
```
# Get the data for index value 5
df_students.loc[5]
```
✓ <1 sec

```
Name          Vicky
StudyHours      1.0
Grade           3.0
Name: 5, dtype: object
```

You can also get the data at a range of index values, like this:

[13]
```
# Get the rows with index values from 0 to 5
df_students.loc[0:5]
```
✓ <1 sec

In addition to being able to use the **loc** method to find rows based on the index, you can use the **iloc** method to find rows based on their ordinal position in the DataFrame (regardless of the index):

[14]
```
# Get data in the first five rows
df_students.iloc[0:5]
```
✓ <1 sec

Look carefully at the `iloc[0:5]` results, and compare them to the `loc[0:5]` results you obtained previously. Can you spot the difference?

The **loc** method returned rows with index *label* in the list of values from *0* to *5* - which includes *0*, *1*, *2*, *3*, *4*, and *5* (six rows). However, the **iloc** method returns the rows in the *positions* included in the range 0 to 5, and since integer ranges don't include the upper-bound value, this includes positions *0*, *1*, *2*, *3*, and *4* (five rows).

iloc identifies data values in a DataFrame by *position*, which extends beyond rows to columns. So for example, you can use it to find the values for the columns in positions 1 and 2 in row 0, like this:

[15]
```
df_students.iloc[0,[1,2]]
```
✓ <1 sec

```
StudyHours    10.0
Grade         50.0
Name: 0, dtype: object
```

Let's return to the **loc** method, and see how it works with columns. Remember that **loc** is used to locate data items based on index values rather than positions. In the absence of an explicit index column, the rows in our dataframe are indexed as integer values, but the columns are identified by name:

[16]
```
df_students.loc[0,'Grade']
```
✓ <1 sec

```
50.0
```

Here's another useful trick. You can use the **loc** method to find indexed rows based on a filtering expression that references named columns other than the index, like this:

[17]
```
df_students.loc[df_students['Name']=='Aisha']
```
✓ <1 sec

Actually, you don't need to explicitly use the **loc** method to do this - you can simply apply a DataFrame filtering expression, like this:

[18]
```
df_students[df_students['Name']=='Aisha']
```
✓ <1 sec

And for good measure, you can achieve the same results by using the DataFrame's **query** method, like this:

[19]
```
df_students.query('Name=="Aisha"')
```
✓ <1 sec

The three previous examples underline an occassionally confusing truth about working with Pandas. Often, there are multiple ways to achieve the same results. Another example of this is the way you refer to a DataFrame column name. You can specify the column name as a named index value (as in the `df_students['Name']` examples we've seen so far), or you can use the column as a property of the DataFrame, like this:

```
[ ]  df_students[df_students.Name == 'Aisha']
     Press shift + enter to run
```

Loading a DataFrame from a file

We constructed the DataFrame from some existing arrays. However, in many real-world scenarios, data is loaded from sources such as files. Let's replace the student grades DataFrame with the contents of a text file.

```
[20]  !wget https://raw.githubusercontent.com/MicrosoftDocs/mslearn-introduction-to
      df_students = pd.read_csv('grades.csv',delimiter=',',header='infer')
      df_students.head()
      ✓ 3 sec
```

```
--2022-03-12 22:58:04--
https://raw.githubusercontent.com/MicrosoftDocs/mslearn-introduction-to-
machine-learning/main/Data/ml-basics/grades.csv
Resolving raw.githubusercontent.com... 185.199.111.133, 185.199.108.133,
185.199.109.133, ...
Connecting to raw.githubusercontent.com|185.199.111.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 322 [text/plain]
Saving to: ‘grades.csv’

grades.csv          100%[===================>]     322  --.-KB/s    in 0s
```

```
2022-03-12 22:58:04 (8.12 MB/s) - ‘grades.csv’ saved [322/322]
```

The DataFrame's **read_csv** method is used to load data from text files. As you can see in the example code, you can specify options such as the column delimiter and which row (if any) contains column headers (in this case, the delimiter is a comma and the first row contains the column names - these are the default settings, so the parameters could have been omitted).

Handling missing values

One of the most common issues data scientists need to deal with is incomplete or missing data. So how would we know that the DataFrame contains missing values? You can use the **isnull** method to identify which individual values are null, like this:

[21]
```
df_students.isnull()
```
✓ <1 sec

Of course, with a larger DataFrame, it would be inefficient to review all of the rows and columns individually; so we can get the sum of missing values for each column, like this:

[22]
```
df_students.isnull().sum()
```
✓ <1 sec

```
Name          0
StudyHours    1
Grade         2
dtype: int64
```

So now we know that there's one missing **StudyHours** value, and two missing **Grade** values.

To see them in context, we can filter the dataframe to include only rows where any of the columns (axis 1 of the DataFrame) are null.

[23]
```
df_students[df_students.isnull().any(axis=1)]
```
✓ <1 sec

When the DataFrame is retrieved, the missing numeric values show up as **NaN** (*not a number*).

So now that we've found the null values, what can we do about them?

One common approach is to *impute* replacement values. For example, if the number of study hours is missing, we could just assume that the student studied for an average amount of time and replace the missing value with the mean study hours. To do this, we can use the **fillna** method, like this:

[24]
```
df_students.StudyHours = df_students.StudyHours.fillna(df_students.StudyHou
df_students
```
✓ <1 sec

Alternatively, it might be important to ensure that you only use data you know to be absolutely correct; so you can drop rows or columns that contains null values by using the **dropna** method. In this case, we'll remove rows (axis 0 of the DataFrame) where any of the columns contain null values.

[25]

```
df_students = df_students.dropna(axis=0, how='any')
df_students
```

✓ <1 sec

Explore data in the DataFrame

Now that we've cleaned up the missing values, we're ready to explore the data in the DataFrame. Let's start by comparing the mean study hours and grades.

```
    # Get the mean study hours using to column name as an index
    mean_study = df_students['StudyHours'].mean()

    # Get the mean grade using the column name as a property (just to make the
    mean_grade = df_students.Grade.mean()

    # Print the mean study hours and mean grade
    print('Average weekly study hours: {:.2f}\nAverage grade: {:.2f}'.format(me
```

[26]
✓ <1 sec

```
Average weekly study hours: 10.52
Average grade: 49.18
```

OK, let's filter the DataFrame to find only the students who studied for more than the average amount of time.

[27]
```
# Get students who studied for the mean or more hours
df_students[df_students.StudyHours > mean_study]
```
✓ <1 sec

Note that the filtered result is itself a DataFrame, so you can work with its columns just like any other DataFrame.

For example, let's find the average grade for students who undertook more than the average amount of study time.

[28]
```
# What was their mean grade?
df_students[df_students.StudyHours > mean_study].Grade.mean()
```
✓ <1 sec

```
66.7
```

Let's assume that the passing grade for the course is 60.

We can use that information to add a new column to the DataFrame, indicating whether or not each student passed.

First, we'll create a Pandas **Series** containing the pass/fail indicator (True or False), and then we'll concatenate that series as a new column (axis 1) in the DataFrame.

[29]

```
passes  = pd.Series(df_students['Grade'] >= 60)
df_students = pd.concat([df_students, passes.rename("Pass")], axis=1)

df_students
```

✓ <1 sec

DataFrames are designed for tabular data, and you can use them to perform many of the kinds of data analytics operation you can do in a relational database; such as grouping and aggregating tables of data.

For example, you can use the **groupby** method to group the student data into groups based on the **Pass** column you added previously, and count the number of names in each group - in other words, you can determine how many students passed and failed.

[30]
```
print(df_students.groupby(df_students.Pass).Name.count())
```
✓ <1 sec

```
Pass
False    15
True      7
Name: Name, dtype: int64
```

You can aggregate multiple fields in a group using any available aggregation function. For example, you can find the mean study time and grade for the groups of students who passed and failed the course.

[31]
```
print(df_students.groupby(df_students.Pass)['StudyHours', 'Grade'].mean())
```
✓ <1 sec

```
       StudyHours      Grade
Pass
False    8.783333  38.000000
True    14.250000  73.142857
<ipython-input-31-8b15539b8d19>:1: FutureWarning: Indexing with multiple keys
(implicitly converted to a tuple of keys) will be deprecated, use a list
instead.
  print(df_students.groupby(df_students.Pass)['StudyHours', 'Grade'].mean())
```

DataFrames are amazingly versatile, and make it easy to manipulate data. Many DataFrame operations return a new copy of the DataFrame; so if you want to modify a DataFrame but keep the existing variable, you need to assign the result of the operation to the existing variable. For example, the following code sorts the student data into descending order of Grade, and assigns the resulting sorted DataFrame to the original **df_students** variable.

```
# Create a DataFrame with the data sorted by Grade (descending)
df_students = df_students.sort_values('Grade', ascending=False)

# Show the DataFrame
df_students
```

[32] ✓ <1 sec

	Name	StudyHours	Grade	Pass
3	Rosie	16.00	97.0	True
10	Francesca	15.50	82.0	True
9	Giovanni	14.50	74.0	True
14	Jenny	15.50	70.0	True
21	Aisha	12.00	64.0	True
20	Daniel	12.50	63.0	True
11	Rajab	13.75	62.0	True
6	Frederic	11.50	53.0	False
19	Skye	12.00	52.0	False
1	Joann	11.50	50.0	False
0	Dan	10.00	50.0	False
4	Ethan	9.25	49.0	False
18	Anila	10.00	48.0	False
2	Pedro	9.00	47.0	False
7	Jimmie	9.00	42.0	False
12	Naiyana	9.00	37.0	False
16	Helena	9.00	36.0	False
17	Ismat	6.00	35.0	False
15	Jakeem	8.00	27.0	False
8	Rhonda	8.50	26.0	False
13	Kian	8.00	15.0	False
5	Vicky	1.00	3.0	False

Summary

That's it for now!

Numpy and DataFrames are the workhorses of data science in Python. They provide us ways to load, explore, and analyze tabular data. As we will see in subsequent modules, even advanced analysis methods typically rely on Numpy and Pandas for these important roles.

In our next workbook, we'll take a look at how create graphs and explore your data in more interesting ways.

+ Code + Markdown

Next unit: Visualize data

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆