```
df_students.iloc[0,[1,2]]
```

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:

```
df_students.loc[0,'Grade']
```

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:

```
df_students.loc[df_students['Name']=='Aisha']
```

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

```
df_students[df_students['Name']=='Aisha']
```

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

```
df_students.query('Name=="Aisha"')
```

The three previous examples underline a 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']
```

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.

```
!wget https://raw.githubusercontent.com/MicrosoftDocs/mslearn-introduction-to-machine-learning
df_students = pd.read_csv('grades.csv',delimiter=',',header='infer')
df_students.head()
```

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.)

```
df_students.isnull()
```

DataFrame contains missing values? You can use the **isnull** method to identify which individual values are null, like this:

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:

```
df_students.isnull().sum()
```

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.

```
df_students[df_students.isnull().any(axis=1)]
```

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

```
    df_students.StudyHours = df_students.StudyHours.fillna(df_students.StudyHours.mean())
    df_students
```

use the **fillna** method like this:

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

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

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 point!)
mean_grade = df_students.Grade.mean()

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

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

```
# Get students who studied for the mean or more hours
df_students[df_students.StudyHours > mean_study]
```

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.

```
# What was their mean grade?
df_students[df_students.StudyHours > mean_study].Grade.mean()
```

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 that indicates 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.

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

    df_students
```

DataFrames are designed for tabular data, and you can use them to perform many of the same kinds of data analytics operations 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 to count the number of names in each group. In other words, you can determine how many students passed and failed.

```
    print(df_students.groupby(df_students.Pass).Name.count())
```

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.

```
    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 by 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
```

Summary

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.

Next unit: Visualize data

Continue >