Selecting rows and columns

Now that we have learned to build a data frame, we are going to learn how to access information from them. There are multiple ways to select and index rows and columns from Pandas DataFrames. There are three main options to achieve the selection and indexing activities in Pandas.

Different Approaches for Selecting rows and columns

One dimensional object is obtained as output when we extract a single row or single column, which is called the Pandas Series data structure.

Here, we specify the name of columns and rows for which we want to retrieve the data.

Throughout this notebook, we take a dataframe as a reference. Let's make it then.

Creating a dataframe

```
import pandas as pd

A = ['a','b','c','d']
B = ['e','f','g','h']
C = ['i', 'j', 'k', 'l']

#create dataframe
df = pd.DataFrame(data=[A, B, C], columns=['one', 'two', 'three', 'four'])

#display dataframe
df
```

	one	two	three	four
0	a	b	c	d
1	e	f	g	h
2	i	j	k	l

Hopefully, everyone understands what the above code is doing. It creates three lists and uses them as rows. It names the columns as well. If you feel confused, please refer to the previous chapter.

Now, we will use **`df['column_name']`** approach to extract the data from specified columns.

```
df['one']
```

```
0    a
1    e
2    i
Name: one, dtype: object
```

We can also extract the specific rows as specified in `df[row_name]`

```
df[1:2] #only start index is inclusive so only first row is extracted
```

	one	two	three	four
1	e	f	g	h

```
df[1:3]
```

	one	two	three	four
1	e	f	g	h
2	i	j	k	l

Using `loc, iloc, ix`

Label based Selection: `loc`

`loc` method uses the label of the rows and columns to select the required data. Imagine that you have a table with people and their personal information. This table might contain their name, age, sex, etc. You need to find information about a person, "Peter." To do this, you look at the column marked "name," and then scroll until you find the particular person's name. Then you want to get all the information in his corresponding row.

In the dataframe we have created, let us try to simulate this by supposing that we want the information about the entry, which has a value of "f" in the column marked "two."

To do this, first, we tell Pandas about our column of interest. We use the `df.set_index()` command to do it.

Then we use the `.loc` method to find the row that contains the information we want.

```
df.set_index("two", inplace=True)
df.loc['f']  # extracts single row corresponding to the label specified.
```

```
one      e
three    g
four     h
Name: f, dtype: object
```

The first line in the above code tells Pandas that our data is in the column named "two." We changed the index for the dataframe from 0, 1, 2, 3, to the items in this column. Then we checked it for where the value was "f" using `df.loc['f']`.

One thing that needs more explanation is the `inplace=True` parameter. When we set the index using `df.set_index()` we modify the dataframe. In our example, if we did not put `inplace=True`, Pandas would first make a copy of the dataframe and then make modifications to the copy. In this case, it would change the index of the copy, and then return this duplicate dataframe with the modifications.

```
df
```

	one	two	three	four
0	a	b	c	d
1	z	f	g	h
2	i	j	k	l

```
# For multiple rows extraction
df.loc[['f', 'b']]
```

	one	three	four
two			
f	e	g	h
b	a	c	d

```
# Selecting both rows and columns
df.loc[['f', 'b'],['one', 'four']]
```

	one	four
two		
f	e	h
b	a	d

```
import pandas as pd

name_list = ['Ford', 'Ferrari', 'Lamborghini', 'Toyota']
shift_list = [1,1,1,0]
color_list = ['red', 'blue', 'white', 'white']
door_list = [4,2,2,4]

#create dataframe
```

```
df = pd.DataFrame(data={'Company':name_list,
                        'Automatic shift':shift_list,
                        'Color':color_list,
                        'Number of doors':door_list})
```

```
df.iloc[1]["Color"]
```

```
'blue'
```

Index based Selection - `iloc`

The `iloc` command for Pandas Dataframe stands for **integer-location(iloc)**. It selects the row and column based on the position.

```
df.iloc[<row selection>, <column selection>]
```

`iloc` in Pandas selects rows and columns by number, in the order that they appear in the data frame. You can imagine that each row has a row number from 0 to the total rows, and `iloc[]` allows selections based on these numbers. The same applies to columns.

For example, in the dataframe above, suppose we want to select the letter 'g'. It is in the second row and third column. However, remember that in programming, we start counting from zero. So, this means that it is on row `1` and column `2`.

We would use the command `df.iloc[1,2]` to access the letter.

```
# retrieving rows by iloc method
df.iloc[2]
```

```
'h'
```

```
# retrieving a specific element with both row and column
df.iloc[1,2]
```

```
'h'
```

```
# selecting multiple rows by iloc method
df.iloc[[1,2]]
```

two	one	three	four
f	e	g	h
j	i	k	l

As you can see, the command outputs the letter 'g'. Now, let's suppose that we want to select the whole row. If we do not plug in any information regarding the column, the command selects the whole row!

```
df.iloc[1] # retrieves the second row which is indexed as 1
```

```
one      e
three    g
four     h
Name: f, dtype: object
```

Notice that the output is correct but vertical. This happens because the output is a Pandas Series. Any operation applied to a pandas Series can be used on this.

You have probably seen the splicing operator for lists. This operator can select elements from the rows or columns of the list. In the same way, you can use it to slice through dataframes.

```
df.iloc[:,1] #retrieve all the rows of second column
```

```
two
b    c
f    g
j    k
Name: three, dtype: object
```

```
# retrieving two rows and two columns by iloc method
df.iloc [[0, 1], [1, 2]]
```

two	three	four
b	c	d
f	g	h

```
# retrieving all rows and some columns by iloc method
df.iloc [:, [1, 2]]
```

two	three	four
b	c	d
f	g	h
j	k	l

As you can see, we can use a semicolon instead of columns or rows number. The semicolon tells pandas to select all the rows. We have given the number 1 to the place where we are to write the column numbers. So it returns all the rows of column 1. Again, pay attention that we start counting from zero in programming.

Selecting data using `ix`

The `ix` operator is deprecated in the recent version of Pandas. `ix` is a hybrid form of `loc` and `iloc`. `ix` is a label-based operator and acts just as the `.loc` indexer. `ix` also supports integer type selections (as .iloc) where passed an integer.

This only works where the index of the DataFrame is not integer-based. Any form of input, i.e., `.loc` and `.iloc` is accepted by `ix`.

```
df.ix[1:2, 'three'] # outputs elements of row 1 and row 2 which fall under the 'three' column
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:1: DeprecationWarning:
.ix is deprecated. Please use
.loc for label based indexing or
.iloc for positional indexing

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#ix-indexer-is-deprecated
  """Entry point for launching an IPython kernel.
1    g
2    k
Name: three, dtype: object
```

Use of `ix` is deprecated and not preferred due to it's ambigious nature.

Selecting data using `at`

`at` command access the single value for row/column pair. It retrieves scalar values. It's a very fast `loc`.

```
df.at[1,'one']
```

```
'e'
```

What is the difference between `iloc` and `loc`?

If you want row labeled output then you use `loc[label]` and if you want a row with specific index output, then you use as `iloc[position]`. `iloc` is used to index a dataframe using 0 to (length-1) be it either row or column indices.

Many times, it becomes a necessity to select the data from a specific condition. This selection is quite natural to do with `loc`. We can pass an array to the `loc` method.

What is the difference between `loc` and `at` operation?

Both `loc` and `at` operations are label based. We use `at` if we only need to get or set a single value in a DataFrame or Series. If we want to replace data at some specific position, then we can use `at`.

```
df.at[1, 'one']='z' # 'e' is replaced by 'z'
df
```

	one	two	three	four
0	a	b	c	d
1	z	f	g	h
2	i	j	k	l

Boolean indices for the selection of the row

Let us suppose you have a database of people and information about them. You might want to find people that are above the age of 50, for example. To do this, we can imagine writing code that looks like something like `"age">50`. But how do we make this work in pandas?

The answer is, by using the `.loc` method. When we use operators to check conditions like less than, greater than, equal to, etc., they return either true or false. If the age is indeed greater, we get True. The `.loc` method then selects these rows that have returned "True" (successfully passed the condition).

Let us do an example. Let us say we want to select all the rows where the column "two" equals "f". In a practical situation, this query is analogous to finding all the people whose addresses equals "China.". In our case, we of finding "f," we use the following command.

```
df.loc[df['two']=='f']
```

	one	two	three	four
1	z	f	g	h

We only got one row here, because we have the letter 'f' in only one row of this column. In a real example, this function returns hundred of rows as the data might contain hundreds of people that live in China. You could also use a command similar to `df['age']>50` to select people above the age of 50.

The code does need more explanation. First, look at the section inside the square brackets `df['two']=='f'`. This code selects the indexes where the condition is true. First, we select the column "two" using `df['two']`. Then it returns a series from which we check equality with the letter "f." Then we use this information and use `df.loc[]` to find the respective rows where the condition is True. It is a three-step process.

Let us go one step further and find specific information. You might not need all the information from the rows where the condition is true. You might want to display the address column of all the people that are above 50, for example.

Let us suppose you want to find the information in column "three" for all entries that have the letter "f" in column two. Here, the condition is that we need the letter "f" on column two. If we meet this condition, we extract the rows. Finally, from these rows, we only pick out the column "three" because that is the only information we want.

We can do this using the following code:

```
df.loc[df['two'] == 'f' , ['three']]
```

	three
1	g

Notice how we have selected the third column. In real life, datasets are massive. So commands that select only specific parts of the data are handy.

With our previous example, you might want to find all the people that have their age above 50 and extract their address and phone number. If you wanted to see what they have in two or more columns, you could do so with the following command.

```
import pandas as pd

A = ['a','b','c','d']
B = ['e','f','g','h']
```

```
C = ['i', 'f', 'k', 'l']

#create dataframe
df = pd.DataFrame(data=[A, B, C], columns=['one', 'two', 'three', 'four'])

#display dataframe
df
```

	one	two	three	four
0	a	b	c	d
1	e	f	g	h
2	i	f	k	l

```
df.loc[df["four"] == "h"]
```

	one	two	three	four
1	e	f	g	h

```
import numpy as np

a = np.array([2,3,4])
b = np.array([True, False, False])

a[b]
```

```
array([2])
```

Start coding or generate with AI.

```
# Retrieve all the rows that has "h" as its value for the column "four"


df.loc[df['two'] == 'f']
```

Start coding or generate with AI.

```
df.iloc[]

df.loc[]

df["name_of_the_column"]


df['two'] == 'f'
```

```
0    False
1     True
2     True
Name: two, dtype: bool
```

```
df.loc[df['two'] == 'f']
```

	one	two	three	four
1	e	f	g	h
2	i	f	k	l

Start coding or generate with AI.

Start coding or generate with AI.

```
df.loc[df['two'] == 'f' , ['three','four']]
```

	three	four
1	g	h

As you can see, we simply put in the columns we want inside the square brackets.

Multiple boolean indices for the selection of the row

```
import pandas as pd

A = ['a','b','c','d']
B = ['e','f','g','h']
C = ['i', 'f', 'k', 'l']
D = ['a', 'u', 'j', 'l']

#create dataframe
df = pd.DataFrame(data=[A, B, C, D], columns=['one', 'two', 'three', 'four'])

#display dataframe
df
```

	one	two	three	four
0	a	b	c	d
1	e	f	g	h
2	i	f	k	l
3	a	u	j	l

```
"f" = > "two"
"l" => "four"


df.loc[(df["two"]=="f") & (df["four"]=="l")]
```

	one	two	three	four
2	i	f	k	l

```
df.loc[(df['one']=='a') & (df['two']== 'b')]
```

	one	two	three	four
0	a	b	c	d

```
df.loc[(df["one"]=="a") & (df["two"]=="b")]
```

	one	two	three	four
0	a	b	c	d

```
"a" => "one"
"b" => "two"


(df["one"]== "a") & (df["two"]=="b") & (df["three"]=="c")
```

	one	two	three	four
0	a	b	c	d

```
df.loc[(df["one"]== "a") & (df["two"]=="b")]
```

	one	two	three	four
0	a	b	c	d

```
df.loc[(df["two"] =='f') & (df["three"] == 'g')]
```

	one	two	three	four
1	e	f	g	h

References

Articles:

- Data Selection
- Filter
- Sort
- Groupby

Pandas cheat sheet (Datacamp):

- https://assets.datacamp.com/blog_assets/PandasPythonForDataScience.pdf