Selecting Subsets of Data in Pandas: Part 1

medium.com/dunder-data/selecting-subsets-of-data-in-pandas-6fcd0170be9c

Ted Petrou

November 25, 2017

This article is available as a Jupyter Notebook complete with exercises at the bottom to practice and detailed solutions in another notebook.

100+ Hours of Free Video Tutorials

I'm just about to release 100+ hours of extremely detailed tutorials on Pandas, Matplotlib, Seaborn, and Scikit-Learn. Subscribe to the Dunder Data YouTube channel to stay updated.

Intro to Data Science Bootcamp

For a more personalized class, take my Data Science Bootcamp in:

- Houston, Dec 15–16, 22–23
- Houston, Dec 17–21

Part 1: Selection with `[ ]`, `.loc` and `.iloc`

This is the beginning of a seven-part series on how to select subsets of data from a pandas DataFrame or Series. Pandas offers a wide variety of options for subset selection which necessitates multiple articles. This series is broken down into the following 8 topics.

1. Selection with `[]`, `.loc` and `.iloc`
2. Boolean indexing
3. Assigning subsets of data
4. How NOT to select subsets of data
5. Selection with a MultiIndex
6. Selecting subsets of data with methods
7. Selections with other Index types
8. Internals, Miscellaneous, and Conclusion

Assumptions before we begin

These series of articles assume you have no knowledge of pandas, but that you understand the fundamentals of the Python programming language. It also assumes that you have installed pandas on your machine.

The easiest way to get pandas along with Python and the rest of the main scientific computing libraries is to install the Anaconda distribution.

If you have no knowledge of Python then I suggest completing the following two books cover to cover before even touching pandas. They are both free.

- Think Python by Allen B. Downey
- Automate the Boring Stuff by Al Sweigart

The importance of making subset selections

You might be wondering why there needs to be so many articles on selecting subsets of data. This topic is extremely important to pandas and it's unfortunate that it is fairly complicated because subset selection happens frequently during an actual analysis. Because you are frequently making subset selections, you need to master it in order to make your life with pandas easier.

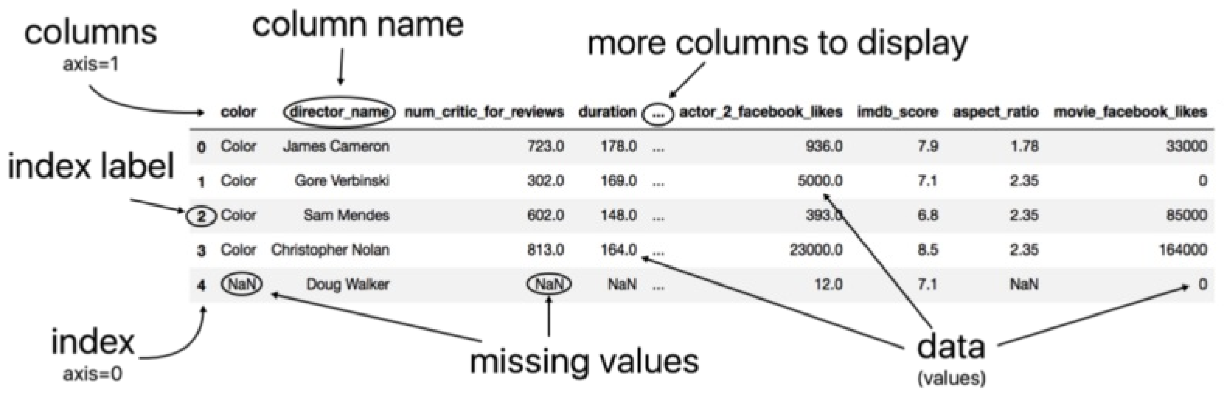
I will also be doing a follow-up series on index alignment which is another extremely important topic that requires you to understand subset selection.

Always reference the documentation

The material in this article is also covered in the official pandas documentation on Indexing and Selecting Data. I highly recommend that you read that part of the documentation along with this tutorial. In fact, the documentation is one of the primary means for mastering pandas. I wrote a step-by-step article, How to Learn Pandas, which gives suggestions on how to use the documentation as you master pandas.

The anatomy of a DataFrame and a Series

The pandas library has two primary containers of data, the DataFrame and the Series. You will spend nearly all your time working with both of the objects when you use pandas. The DataFrame is used more than the Series, so let's take a look at an image of it first.



	color	director_name	num_critic_for_reviews	duration	...	actor_2_facebook_likes	imdb_score	aspect_ratio	movie_facebook_likes
0	Color	James Cameron	723.0	178.0	...	936.0	7.9	1.78	33000
1	Color	Gore Verbinski	302.0	169.0	...	5000.0	7.1	2.35	0
2	Color	Sam Mendes	602.0	148.0	...	393.0	6.8	2.35	85000
3	Color	Christopher Nolan	813.0	164.0	...	23000.0	8.5	2.35	164000
4	NaN	Doug Walker	NaN	NaN	...	12.0	7.1	NaN	0

Anatomy of a DataFrame

This image comes with some added illustrations to highlight its components. At first glance, the DataFrame looks like any other two-dimensional table of data that you have seen. It has rows and it has columns. Technically, there are three main components of the DataFrame.

The three components of a DataFrame

A DataFrame is composed of three different components, the **index**, **columns**, and the **data**. The data is also known as the **values**.

The index represents the sequence of values on the far left-hand side of the DataFrame. All the values in the index are in **bold** font. Each individual value of the index is called a **label**. Sometimes the index is referred to as the **row labels**. In the example above, the row labels are not very interesting and are just the integers beginning from 0 up to n-1, where n is the number of rows in the table. Pandas defaults DataFrames with this simple index.

The columns are the sequence of values at the very top of the DataFrame. They are also in **bold** font. Each individual value of the columns is called a **column**, but can also be referred to as **column name** or **column label**.

Everything else not in bold font is the data or values. You will sometimes hear DataFrames referred to as **tabular** data. This is just another name for a rectangular table data with rows and columns.

Axis and axes

It is also common terminology to refer to the rows or columns as an **axis**. Collectively, we call them **axes**. So, a row is an axis and a column is another axis.

The word axis appears as a parameter in many DataFrame methods. Pandas allows you to choose the direction of how the method will work with this parameter. This has nothing to do with subset selection so you can just ignore it for now.

Each row has a label and each column has a label

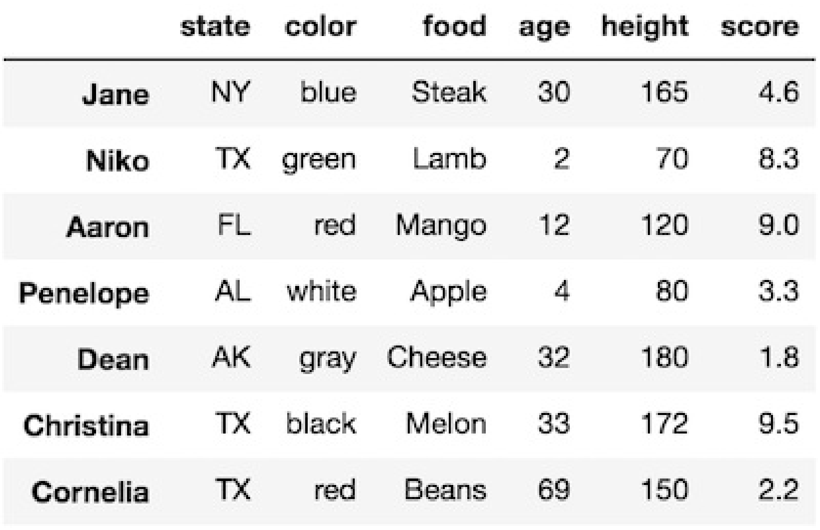
The main takeaway from the DataFrame anatomy is that each row has a label and each column has a label. These labels are used to refer to specific rows or columns in the DataFrame. It's the same as how humans use names to refer to specific people.

What is subset selection?

Before we start doing subset selection, it might be good to define what it is. Subset selection is simply selecting particular rows and columns of data from a DataFrame (or Series). This could mean selecting all the rows and some of the columns, some of the rows and all of the columns, or some of each of the rows and columns.

Example selecting some columns and all rows

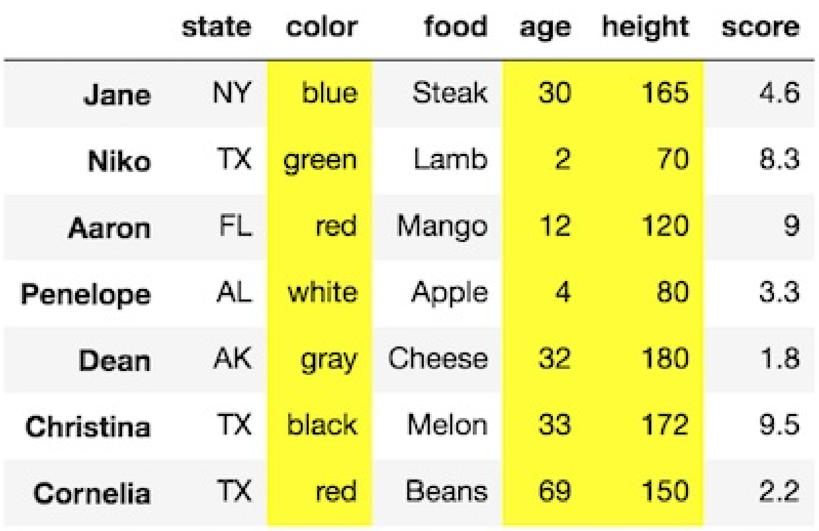
Let's see some images of subset selection. We will first look at a sample DataFrame with fake data.



	state	color	food	age	height	score
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

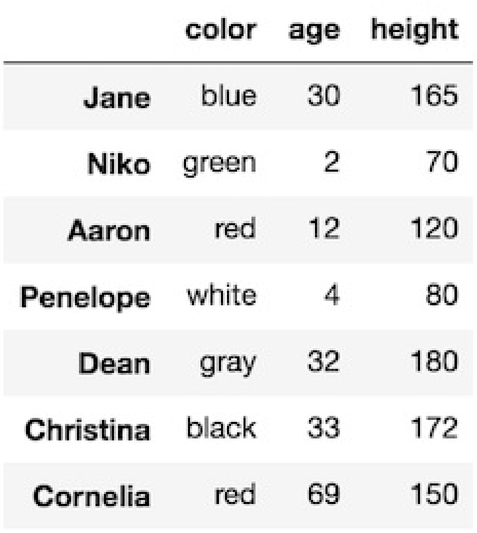
Sample DataFrame

Let's say we want to select just the columns `color`, `age`, and `height` but keep all the rows.



	state	color	food	age	height	score
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

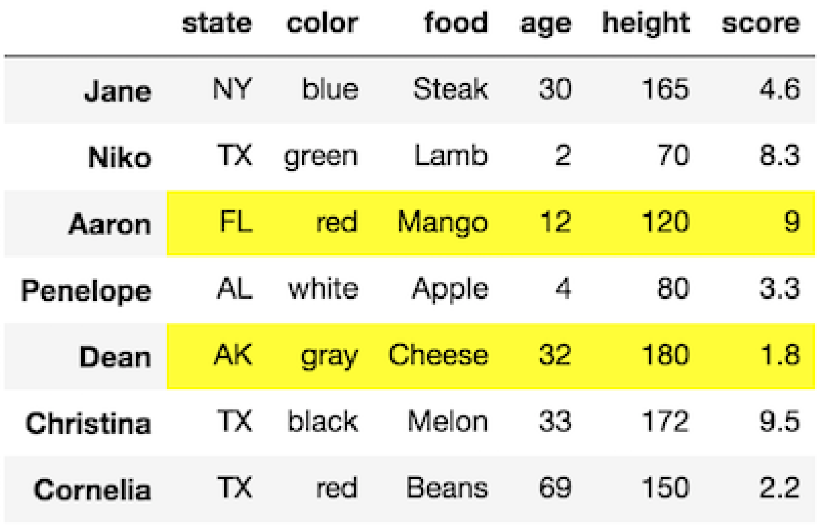
Our final DataFrame would look like this:



	color	age	height
Jane	blue	30	165
Niko	green	2	70
Aaron	red	12	120
Penelope	white	4	80
Dean	gray	32	180
Christina	black	33	172
Cornelia	red	69	150

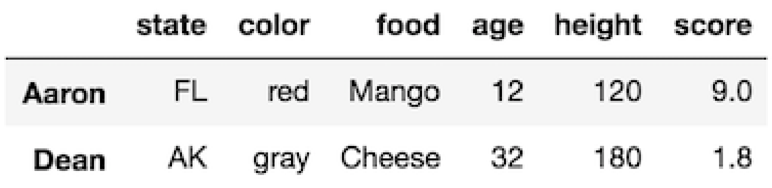
Example selecting some rows and all columns

We can also make selections that select just some of the rows. Let's select the rows with labels `Aaron` and `Dean` along with all of the columns:



	state	color	food	age	height	score
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

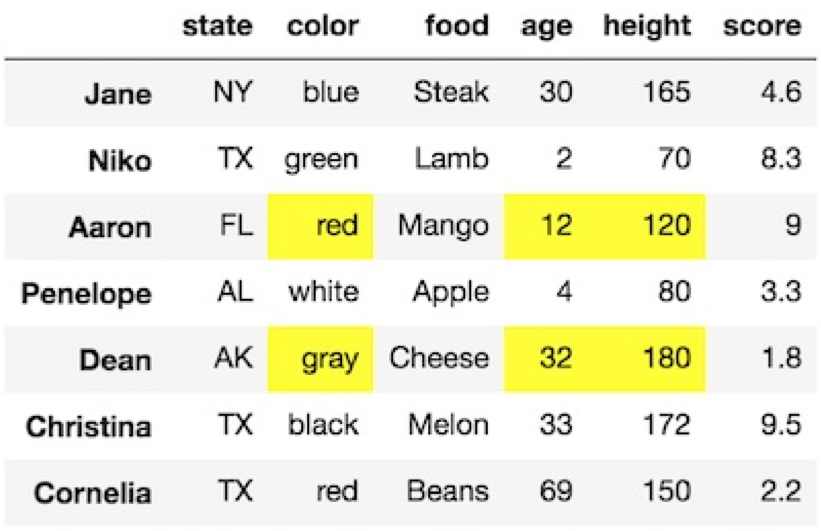
Our final DataFrame would like:



	state	color	food	age	height	score
Aaron	FL	red	Mango	12	120	9.0
Dean	AK	gray	Cheese	32	180	1.8

Example selecting some rows and some columns

Let's combine the selections from above and select the columns `color`, `age`, and `height` for only the rows with labels `Aaron` and `Dean`.



	state	color	food	age	height	score
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

Our final DataFrame would look like this:

	color	age	height
Aaron	red	12	120
Dean	gray	32	180

Pandas dual references: by label and by integer location

We already mentioned that each row and each column have a specific label that can be used to reference them. This is displayed in bold font in the DataFrame.

But, what hasn't been mentioned, is that each row and column may be referenced by an integer as well. I call this **integer location**. The integer location begins at 0 and ends at n-1 for each row and column. Take a look above at our sample DataFrame one more time.

The rows with labels `Aaron` and `Dean` can also be referenced by their respective integer locations 2 and 4. Similarly, the columns `color`, `age` and `height` can be referenced by their integer locations 1, 3, and 4.

The documentation refers to integer location as **position**. I don't particularly like this terminology as its not as explicit as integer location. The key thing term here is INTEGER.

What's the difference between indexing and selecting subsets of data?

The documentation uses the term **indexing** frequently. This term is essentially just a one-word phrase to say 'subset selection'. I prefer the term subset selection as, again, it is more descriptive of what is actually happening. Indexing is also the term used in the official Python documentation.

Focusing only on `[]`, `.loc`, and `.iloc`

There are many ways to select subsets of data, but in this article we will only cover the usage of the square brackets (**`[]`**), **`.loc`** and **`.iloc`**. Collectively, they are called the **indexers**. These are by far the most common ways to select data. A different part of this Series will discuss a few methods that can be used to make subset selections.

If you have a DataFrame, `df`, your subset selection will look something like the following:

```
df[ ]
df.loc[ ]
df.iloc[ ]
```

A real subset selection will have something inside of the square brackets. All selections in this article will take place inside of those square brackets.

Notice that the square brackets also follow `.loc` and `.iloc`. All indexing in Python happens inside of these square brackets.

A term for just those square brackets

The term **indexing operator** is used to refer to the square brackets following an object. The `.loc` and `.iloc` indexers also use the indexing operator to make selections. I will use the term **just the indexing operator** to refer to `df[]`. This will distinguish it from `df.loc[]` and `df.iloc[]`.

Read in data into a DataFrame with `read_csv`

Let's begin using pandas to read in a DataFrame, and from there, use the indexing operator by itself to select subsets of data. All the data for these tutorials are in the **data** directory.

We will use the `read_csv` function to read in data into a DataFrame. We pass the path to the file as the first argument to the function. We will also use the `index_col` parameter to select the first column of data as the index (more on this later).

```
>>> import pandas as pd
>>> import numpy as np

>>> df = pd.read_csv('data/sample_data.csv', index_col=0)
>>> df
```

	state	color	food	age	height	score
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

Extracting the individual DataFrame components

Earlier, we mentioned the three components of the DataFrame. The index, columns and data (values). We can extract each of these components into their own variables. Let's do that and then inspect them:

```
>>> index = df.index
>>> columns = df.columns
>>> values = df.values

>>> index
Index(['Jane', 'Niko', 'Aaron', 'Penelope', 'Dean', 'Christina',
       'Cornelia'], dtype='object')

>>> columns
Index(['state', 'color', 'food', 'age', 'height', 'score'],
      dtype='object')
```

```
>>> values
array([['NY', 'blue', 'Steak', 30, 165, 4.6],
       ['TX', 'green', 'Lamb', 2, 70, 8.3],
       ['FL', 'red', 'Mango', 12, 120, 9.0],
       ['AL', 'white', 'Apple', 4, 80, 3.3],
       ['AK', 'gray', 'Cheese', 32, 180, 1.8],
       ['TX', 'black', 'Melon', 33, 172, 9.5],
       ['TX', 'red', 'Beans', 69, 150, 2.2]], dtype=object)
```

Data types of the components

Let's output the type of each component to understand exactly what kind of object they are.

```
>>> type(index)
pandas.core.indexes.base.Index

>>> type(columns)
pandas.core.indexes.base.Index

>>> type(values)
numpy.ndarray
```

Understanding these types

Interestingly, both the index and the columns are the same type. They are both a pandas `Index` object. This object is quite powerful in itself, but for now you can just think of it as a sequence of labels for either the rows or the columns.

The values are a NumPy `ndarray`, which stands for n-dimensional array, and is the primary container of data in the NumPy library. Pandas is built directly on top of NumPy and it's this array that is responsible for the bulk of the workload.

Beginning with just the indexing operator on DataFrames

We will begin our journey of selecting subsets by using just the indexing operator on a DataFrame. Its main purpose is to select a single column or multiple columns of data.

Selecting a single column as a Series

To select a single column of data, simply put the name of the column in-between the brackets. Let's select the food column:

```
>>> df['food']
Jane          Steak
Niko           Lamb
Aaron         Mango
Penelope      Apple
Dean         Cheese
Christina     Melon
Cornelia      Beans
Name: food, dtype: object
```

Anatomy of a Series

Selecting a single column of data returns the other pandas data container, the Series. A Series is a one-dimensional sequence of labeled data. There are two main components of a Series, the **index** and the **data**(or **values**). There are NO columns in a Series.

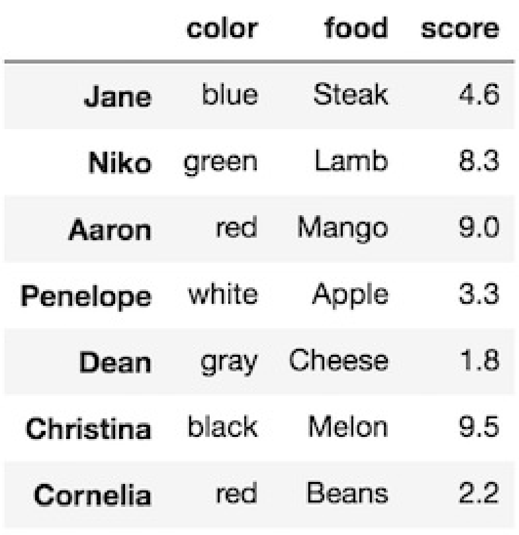
The visual display of a Series is just plain text, as opposed to the nicely styled table for DataFrames. The sequence of person names on the left is the index. The sequence of food items on the right is the values.

You will also notice two extra pieces of data on the bottom of the Series. The **name** of the Series becomes the old-column name. You will also see the data type or `dtype` of the Series. You can ignore both these items for now.

Selecting multiple columns with just the indexing operator

It's possible to select multiple columns with just the indexing operator by passing it a list of column names. Let's select `color`, `food`, and `score`:

```
>>> df[['color', 'food', 'score']]
```



	color	food	score
Jane	blue	Steak	4.6
Niko	green	Lamb	8.3
Aaron	red	Mango	9.0
Penelope	white	Apple	3.3
Dean	gray	Cheese	1.8
Christina	black	Melon	9.5
Cornelia	red	Beans	2.2

Selecting multiple columns returns a DataFrame

Selecting multiple columns returns a DataFrame. You can actually select a single column as a DataFrame with a one-item list:

```
df[['food']]
```

Although, this resembles the Series from above, it is technically a DataFrame, a different object.

	food
Jane	Steak
Niko	Lamb
Aaron	Mango
Penelope	Apple
Dean	Cheese
Christina	Melon
Cornelia	Beans

Column order doesn’t matter

When selecting multiple columns, you can select them in any order that you choose. It doesn’t have to be the same order as the original DataFrame. For instance, let’s select `height` and `color`.

```
df[['height', 'color']]
```

	height	color
Jane	165	blue
Niko	70	green
Aaron	120	red
Penelope	80	white
Dean	180	gray
Christina	172	black
Cornelia	150	red

Exceptions

There are a couple common exceptions that arise when doing selections with just the indexing operator.

- If you misspell a word, you will get a `KeyError`
- If you forgot to use a list to contain multiple columns you will also get a `KeyError`

```
>>> df['hight']
KeyError: 'hight'

>>> df['color', 'age'] # should be:  df[['color', 'age']]
KeyError: ('color', 'age')
```

Summary of just the indexing operator

- Its primary purpose is to select columns by the column names
- Select a single column as a Series by passing the column name directly to it: `df['col_name']`
- Select multiple columns as a DataFrame by passing a **list** to it: `df[['col_name1', 'col_name2']]`
- You actually can select rows with it, but this will not be shown here as it is confusing and not used often.

Getting started with `.loc`

The `.loc` indexer selects data in a different way than just the indexing operator. It can select subsets of rows or columns. It can also simultaneously select subsets of rows and columns. Most importantly, it only selects data by the **LABEL** of the rows and columns.

Select a single row as a Series with `.loc`

The `.loc` indexer will return a single row as a Series when given a single row label. Let's select the row for `Niko`.

```
>>> df.loc['Niko']
state         TX
color      green
food        Lamb
age            2
height        70
score        8.3
Name: Niko, dtype: object
```

We now have a Series, where the old column names are now the index labels. The `name` of the Series has become the old index label, `Niko` in this case.

Select multiple rows as a DataFrame with `.loc`

To select multiple rows, put all the row labels you want to select in a list and pass that to `.loc`. Let's select `Niko` and `Penelope`.

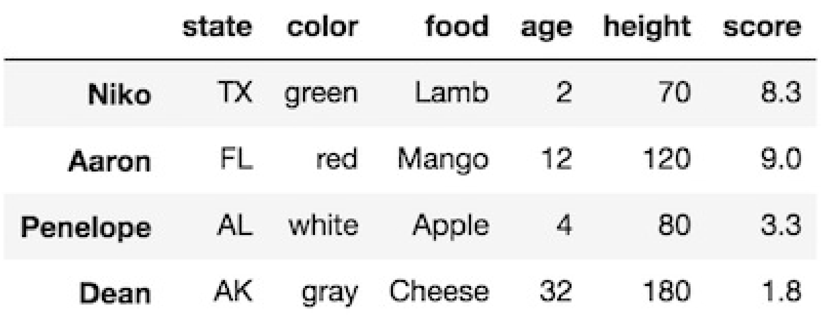
```
>>> df.loc[['Niko', 'Penelope']]
```

	state	color	food	age	height	score
Niko	TX	green	Lamb	2	70	8.3
Penelope	AL	white	Apple	4	80	3.3

Use slice notation to select a range of rows with `.loc`

It is possible to 'slice' the rows of a DataFrame with `.loc` by using **slice notation**. Slice notation uses a colon to separate **start**, **stop** and **step** values. For instance we can select all the rows from `Niko` through `Dean` like this:

```
>>> df.loc['Niko':'Dean']
```



	state	color	food	age	height	score
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8

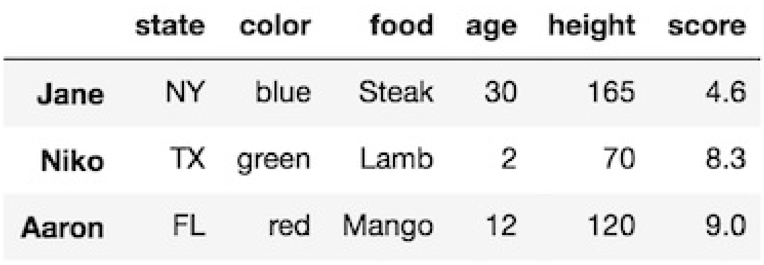
`.loc` includes the last value with slice notation

Notice that the row labeled with `Dean` was kept. In other data containers such as Python lists, the last value is excluded.

Other slices

You can use slice notation similarly to how you use it with lists. Let's slice from the beginning through `Aaron`:

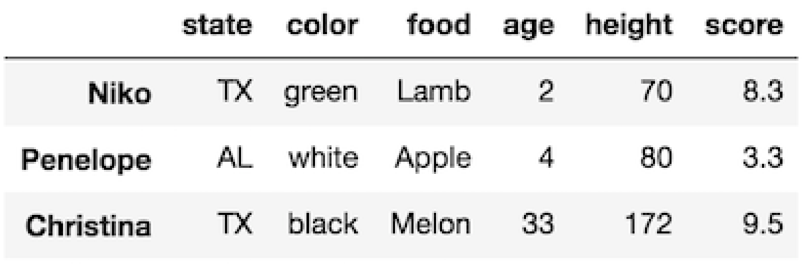
```
>>> df.loc[:'Aaron']
```



	state	color	food	age	height	score
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0

Slice from `Niko` to `Christina` stepping by 2:

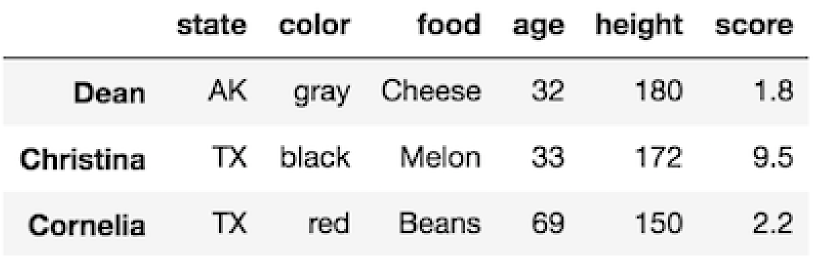
```
>>> df.loc['Niko':'Christina':2]
```



	state	color	food	age	height	score
Niko	TX	green	Lamb	2	70	8.3
Penelope	AL	white	Apple	4	80	3.3
Christina	TX	black	Melon	33	172	9.5

Slice from `Dean` to the end:

```
>>> df.loc['Dean':]
```



	state	color	food	age	height	score
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

Selecting rows and columns simultaneously with `.loc`

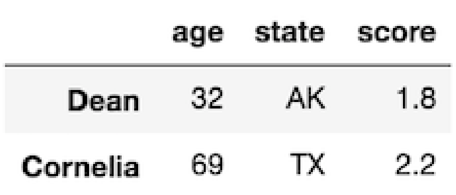
Unlike just the indexing operator, it is possible to select rows and columns simultaneously with `.loc`. You do it by separating your row and column selections by a **comma**. It will look something like this:

```
>>> df.loc[row_selection, column_selection]
```

Select two rows and three columns

For instance, if we wanted to select the rows `Dean` and `Cornelia` along with the columns `age`, `state` and `score` we would do this:

```
>>> df.loc[['Dean', 'Cornelia'], ['age', 'state', 'score']]
```



	age	state	score
Dean	32	AK	1.8
Cornelia	69	TX	2.2

Use any combination of selections for either row or columns for `.loc`

Row or column selections can be any of the following as we have already seen:

- A single label
- A list of labels
- A slice with labels

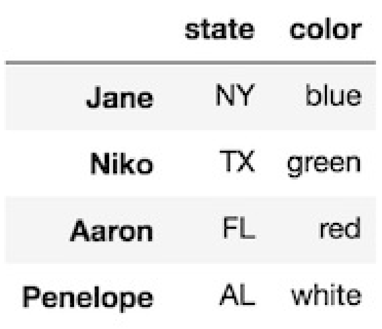
We can use any of these three for either row or column selections with `.loc`. Let's see some examples.

Let’s select two rows and a single column:

```
>>> df.loc[['Dean', 'Aaron'], 'food']
Dean      Cheese
Aaron      Mango
Name: food, dtype: object
```

Select a slice of rows and a list of columns:

```
>>> df.loc['Jane':'Penelope', ['state', 'color']]
```



	state	color
Jane	NY	blue
Niko	TX	green
Aaron	FL	red
Penelope	AL	white

Select a single row and a single column. This returns a scalar value.

```
>>> df.loc['Jane', 'age']
30
```

Select a slice of rows and columns

```
>>> df.loc[:'Dean', 'height':]
```

	height	score
Jane	165	4.6
Niko	70	8.3
Aaron	120	9.0
Penelope	80	3.3
Dean	180	1.8

Selecting all of the rows and some columns

It is possible to select all of the rows by using a single colon. You can then select columns as normal:

```
>>> df.loc[:, ['food', 'color']]
```

	food	color
Jane	Steak	blue
Niko	Lamb	green
Aaron	Mango	red
Penelope	Apple	white
Dean	Cheese	gray
Christina	Melon	black
Cornelia	Beans	red

You can also use this notation to select all of the columns:

```
>>> df.loc[['Penelope','Cornelia'], :]
```

	state	color	food	age	height	score
Penelope	AL	white	Apple	4	80	3.3
Cornelia	TX	red	Beans	69	150	2.2

But, it isn’t necessary as we have seen, so you can leave out that last colon:

```
>>> df.loc[['Penelope','Cornelia']]
```

	state	color	food	age	height	score
Penelope	AL	white	Apple	4	80	3.3
Cornelia	TX	red	Beans	69	150	2.2

Assign row and column selections to variables

It might be easier to assign row and column selections to variables before you use `.loc`. This is useful if you are selecting many rows or columns:

```
>>> rows = ['Jane', 'Niko', 'Dean', 'Penelope', 'Christina']
>>> cols = ['state', 'age', 'height', 'score']
>>> df.loc[rows, cols]
```

	state	age	height	score
Jane	NY	30	165	4.6
Niko	TX	2	70	8.3
Dean	AK	32	180	1.8
Penelope	AL	4	80	3.3
Christina	TX	33	172	9.5

Summary of `.loc`

- Only uses labels
- Can select rows and columns simultaneously
- Selection can be a single label, a list of labels or a slice of labels
- Put a comma between row and column selections

Getting started with `.iloc`

The `.iloc` indexer is very similar to `.loc` but only uses integer locations to make its selections. The word `.iloc` itself stands for integer location so that should help with remember what it does.

Selecting a single row with `.iloc`

By passing a single integer to `.iloc`, it will select one row as a Series:

```
>>> df.iloc[3]
state          AL
color       white
food        Apple
age             4
height         80
score         3.3
Name: Penelope, dtype: object
```

Selecting multiple rows with `.iloc`

Use a list of integers to select multiple rows:

```
>>> df.iloc[[5, 2, 4]]       # remember, don't do df.iloc[5, 2, 4]
```

	state	color	food	age	height	score
Christina	TX	black	Melon	33	172	9.5
Aaron	FL	red	Mango	12	120	9.0
Dean	AK	gray	Cheese	32	180	1.8

Use slice notation to select a range of rows with `.iloc`

Slice notation works just like a list in this instance and is exclusive of the last element

```
>>> df.iloc[3:5]
```

	state	color	food	age	height	score
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8

Select 3rd position until end:

```
>>> df.iloc[3:]
```

	state	color	food	age	height	score
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

Select 3rd position to end by 2:

```
>>> df.iloc[3::2]
```

	state	color	food	age	height	score
Penelope	AL	white	Apple	4	80	3.3
Christina	TX	black	Melon	33	172	9.5

Selecting rows and columns simultaneously with `.iloc`

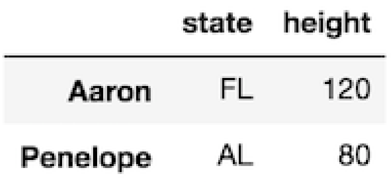
Just like with `.iloc` any combination of a single integer, lists of integers or slices can be used to select rows and columns simultaneously. Just remember to separate the selections with a **comma**.

Select two rows and two columns:

```
>>> df.iloc[[2,3], [0, 4]]
```

Select a slice of the rows and two columns:

```
>>> df.iloc[3:6, [1, 4]]
```



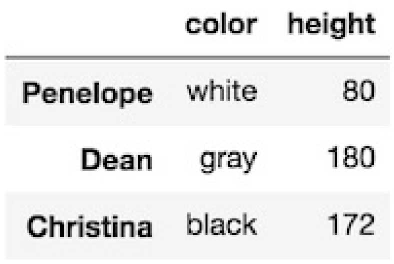
	state	height
Aaron	FL	120
Penelope	AL	80

Select slices for both

```
>>> df.iloc[2:5, 2:5]
```

Select a single row and column

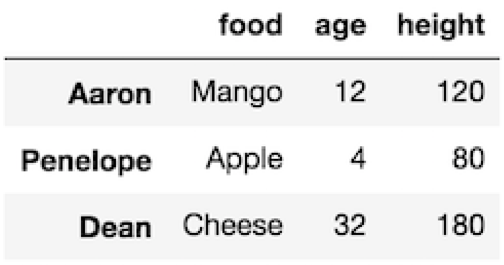
```
>>> df.iloc[0, 2]
'Steak'
```



	color	height
Penelope	white	80
Dean	gray	180
Christina	black	172

Select all the rows and a single column

```
>>> df.iloc[:, 5]
Jane         4.6
Niko         8.3
Aaron        9.0
Penelope     3.3
Dean         1.8
Christina    9.5
Cornelia     2.2
Name: score, dtype: float64
```



	food	age	height
Aaron	Mango	12	120
Penelope	Apple	4	80
Dean	Cheese	32	180

Deprecation of `.ix`

Early in the development of pandas, there existed another indexer, `ix`. This indexer was capable of selecting both by label and by integer location. While it was versatile, it caused lots of confusion because it's not explicit. Sometimes integers can also be labels for rows or columns. Thus there were instances where it was ambiguous.

You can still call `.ix`, but it has been deprecated, so please **never use it**.

Selecting subsets of Series

We can also, of course, do subset selection with a Series. Earlier I recommended using just the indexing operator for column selection on a DataFrame. Since Series do not have columns, I suggest using only `.loc` and `.iloc`. You can use just the indexing operator, but its ambiguous as it can take both labels and integers. I will come back to this at the end of the tutorial.

Typically, you will create a Series by selecting a single column from a DataFrame. Let’s select the `food` column:

```
>>> food = df['food']
>>> food
Jane          Steak
Niko           Lamb
Aaron         Mango
Penelope      Apple
Dean         Cheese
Christina     Melon
Cornelia      Beans
Name: food, dtype: object
```

Series selection with `.loc`

Series selection with `.loc` is quite simple, since we are only dealing with a single dimension. You can again use a single row label, a list of row labels or a slice of row labels to make your selection. Let's see several examples.

Let’s select a single value:

```
>>> food.loc['Aaron']
'Mango'
```

Select three different values. This returns a Series:

```
>>> food.loc[['Dean', 'Niko', 'Cornelia']]
Dean        Cheese
Niko          Lamb
Cornelia     Beans
Name: food, dtype: object
```

Slice from `Niko` to `Christina` - is inclusive of last index

```
>>> food.loc['Niko':'Christina']
Niko           Lamb
Aaron         Mango
Penelope      Apple
Dean         Cheese
Christina     Melon
Name: food, dtype: object
```

Slice from `Penelope` to the end:

```
>>> food.loc['Penelope':]
Penelope      Apple
Dean         Cheese
Christina     Melon
Cornelia      Beans
Name: food, dtype: object
```

Select a single value in a list which returns a Series

```
>>> food.loc[['Aaron']]
Aaron    Mango
Name: food, dtype: object
```

Series selection with `.iloc`

Series subset selection with `.iloc` happens similarly to `.loc` except it uses integer location. You can use a single integer, a list of integers or a slice of integers. Let's see some examples.

Select a single value:

```
>>> food.iloc[0]
'Steak'
```

Use a list of integers to select multiple values:

```
>>> food.iloc[[4, 1, 3]]
Dean        Cheese
Niko          Lamb
Penelope     Apple
Name: food, dtype: object
```

Use a slice—is exclusive of last integer

```
>>> food.iloc[4:6]
Dean         Cheese
Christina     Melon
Name: food, dtype: object
```

Comparison to Python lists and dictionaries

It may be helpful to compare pandas ability to make selections by label and integer location to that of Python lists and dictionaries.

Python lists allow for selection of data only through integer location. You can use a single integer or slice notation to make the selection but NOT a list of integers.

Let's see examples of subset selection of lists using integers:

```
>>> some_list = ['a', 'two', 10, 4, 0, 'asdf', 'mgmt', 434, 99]

>>> some_list[5]
'asdf'

>>> some_list[-1]
99

>>> some_list[:4]
['a', 'two', 10, 4]

>>> some_list[3:]
[4, 0, 'asdf', 'mgmt', 434, 99]

>>> some_list[2:6:3]
[10, 'asdf']
```

Selection by label with Python dictionaries

All values in each dictionary are labeled by a **key**. We use this key to make single selections. Dictionaries only allow selection with a single label. Slices and lists of labels are not allowed.

```
>>> d = {'a':1, 'b':2, 't':20, 'z':26, 'A':27}
>>> d['a']
1

>>> d['A']
27
```

Pandas has power of lists and dictionaries

DataFrames and Series are able to make selections with integers like a list and with labels like a dictionary.

Extra Topics

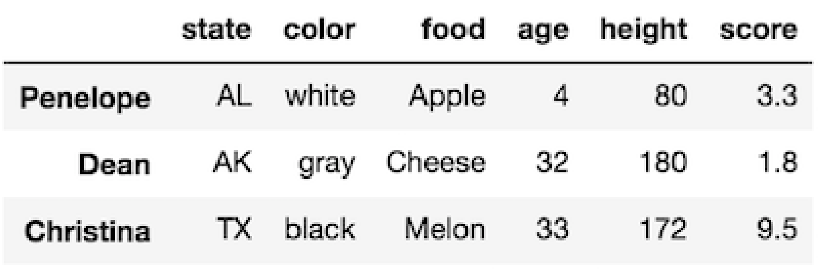
There are a few more items that are important and belong in this tutorial and will be mentioned now.

Using just the indexing operator to select rows from a DataFrame — Confusing!

Above, I used just the indexing operator to select a column or columns from a DataFrame. But, it can also be used to select rows using a **slice**. This behavior is very confusing in my opinion. The entire operation changes completely when a slice is passed.

Let's use an integer slice as our first example:

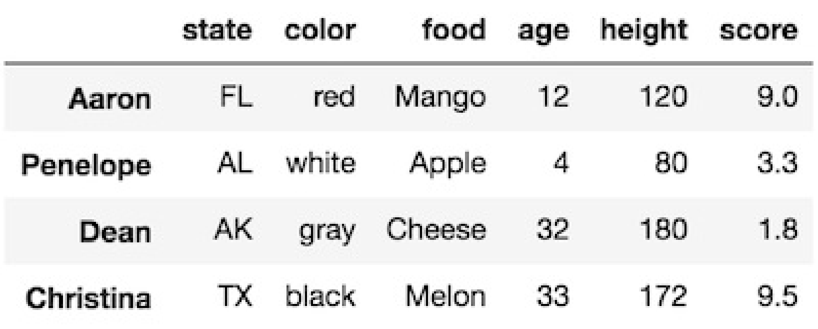
```
>>> df[3:6]
```



	state	color	food	age	height	score
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5

To add to this confusion, you can slice by labels as well.

```
>>> df['Aaron':'Christina']
```



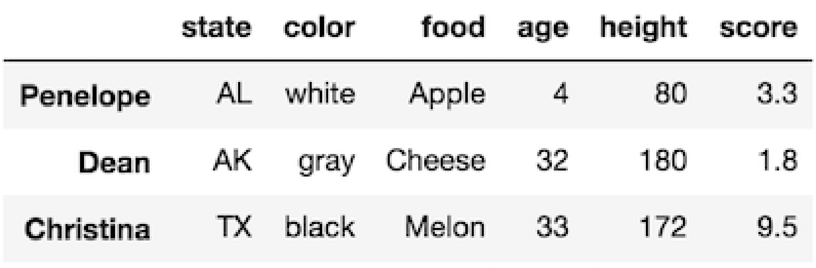
	state	color	food	age	height	score
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5

I recommend not doing this!

This feature is not deprecated and completely up to you whether you wish to use it. But, I highly prefer not to select rows in this manner as can be ambiguous, especially if you have integers in your index.

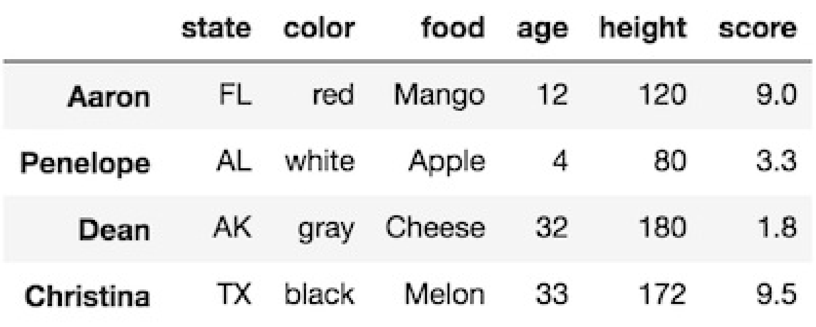
Using `.iloc` and `.loc` is explicit and clearly tells the person reading the code what is going to happen. Let's rewrite the above using `.iloc` and `.loc`.

```
>>> df.iloc[3:6]      # More explicit that df[3:6]
```



	state	color	food	age	height	score
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5

```
>>> df.loc['Aaron':'Christina']
```



	state	color	food	age	height	score
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5

Cannot simultaneously select rows and columns with `[]`

An exception will be raised if you try and select rows and columns simultaneously with just the indexing operator. You must use `.loc` or `.iloc` to do so.

```
>>> df[3:6, 'Aaron':'Christina']
TypeError: unhashable type: 'slice'
```

Using just the indexing operator to select rows from a Series—Confusing!

You can also use just the indexing operator with a Series. Again, this is confusing because it can accept integers or labels. Let's see some examples

```
>>> food
Jane          Steak
Niko           Lamb
Aaron         Mango
Penelope      Apple
Dean         Cheese
Christina     Melon
Cornelia      Beans
Name: food, dtype: object

>>> food[2:4]
Aaron       Mango
Penelope    Apple
Name: food, dtype: object
```

```
>>> food['Niko':'Dean']
Niko          Lamb
Aaron        Mango
Penelope     Apple
Dean        Cheese
Name: food, dtype: object
```

Since Series don't have columns you can use a single label and list of labels to make selections as well

```
>>> food['Dean']
'Cheese'

>>> food[['Dean', 'Christina', 'Aaron']]
Dean         Cheese
Christina     Melon
Aaron         Mango
Name: food, dtype: object
```

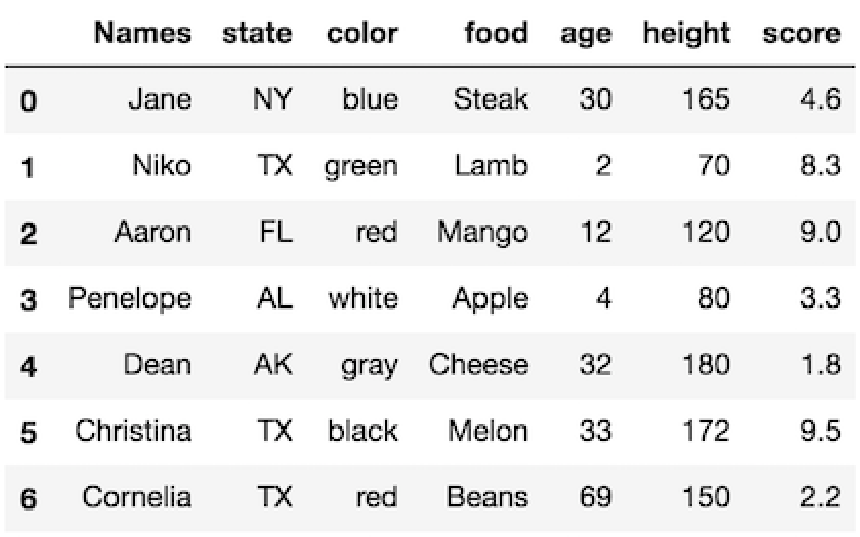
Again, I recommend against doing this and always use `.iloc` or `.loc`

Importing data without choosing an index column

We imported data by choosing the first column to be the index with the `index_col` parameter of the `read_csv` function. This is not typically how most DataFrames are read into pandas.

Usually, all the columns in the csv file become DataFrame columns. Pandas will use the integers 0 to n-1 as the labels. See the example data below with a slightly different dataset:

```
>>> df2 = pd.read_csv('data/sample_data2.csv')
>>> df2
```



	Names	state	color	food	age	height	score
0	Jane	NY	blue	Steak	30	165	4.6
1	Niko	TX	green	Lamb	2	70	8.3
2	Aaron	FL	red	Mango	12	120	9.0
3	Penelope	AL	white	Apple	4	80	3.3
4	Dean	AK	gray	Cheese	32	180	1.8
5	Christina	TX	black	Melon	33	172	9.5
6	Cornelia	TX	red	Beans	69	150	2.2

The default `RangeIndex`

If you don't specify a column to be the index when first reading in the data, pandas will use the integers 0 to n-1 as the index. This technically creates a `RangeIndex` object. Let's take a look at it.

```
>>> df2.index
RangeIndex(start=0, stop=7, step=1)
```

This object is similar to Python `range` objects. Let's create one:

```
>>> range(7)
range(0, 7)
```

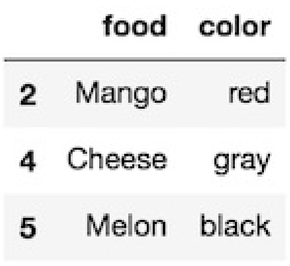
Converting both of these objects to a list produces the exact same thing:

```
>>> list(df2.index)
[0, 1, 2, 3, 4, 5, 6]
```

```
>>> list(range(7))
[0, 1, 2, 3, 4, 5, 6]
```

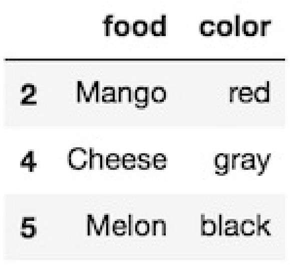
For now, it's not at all important that you have a `RangeIndex`. Selections from it happen just the same with `.loc` and `.iloc`. Let's look at some examples.

```
>>> df2.loc[[2, 4, 5], ['food', 'color']]
```



	food	color
2	Mango	red
4	Cheese	gray
5	Melon	black

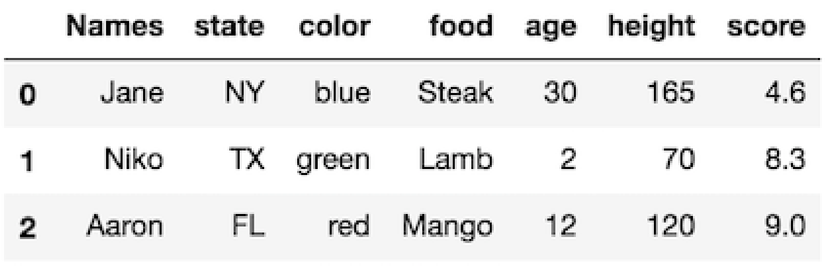
```
>>> df2.iloc[[2, 4, 5], [3,2]]
```



	food	color
2	Mango	red
4	Cheese	gray
5	Melon	black

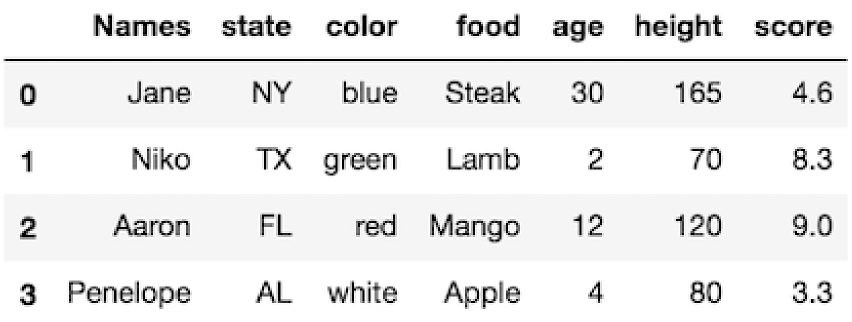
There is a subtle difference when using a slice. `.iloc` excludes the last value, while `.loc` includes it:

```
>>> df2.iloc[:3]
```



	Names	state	color	food	age	height	score
0	Jane	NY	blue	Steak	30	165	4.6
1	Niko	TX	green	Lamb	2	70	8.3
2	Aaron	FL	red	Mango	12	120	9.0

```
>>> df2.loc[:3]
```

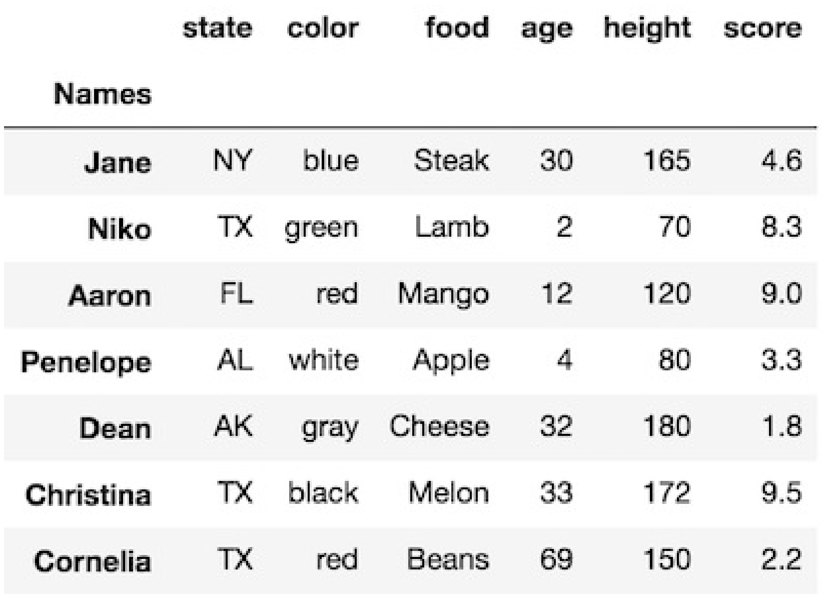


	Names	state	color	food	age	height	score
0	Jane	NY	blue	Steak	30	165	4.6
1	Niko	TX	green	Lamb	2	70	8.3
2	Aaron	FL	red	Mango	12	120	9.0
3	Penelope	AL	white	Apple	4	80	3.3

Setting an index from a column after reading in data

It is common to see pandas code that reads in a DataFrame with a RangeIndex and then sets the index to be one of the columns. This is typically done with the `set_index` method:

```
>>> df2_idx = df2.set_index('Names')
>>> df2_idx
```



	state	color	food	age	height	score
Names						
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

The index has a name

Notice that this DataFrame does not look exactly like our first one from the very top of this tutorial. Directly above the index is the bold-faced word `Names`. This is technically the **name** of the index. Our original DataFrame had no name for its index. You can ignore this small detail for now. Subset selections will happen in the same fashion.

DataFrame column selection with dot notation

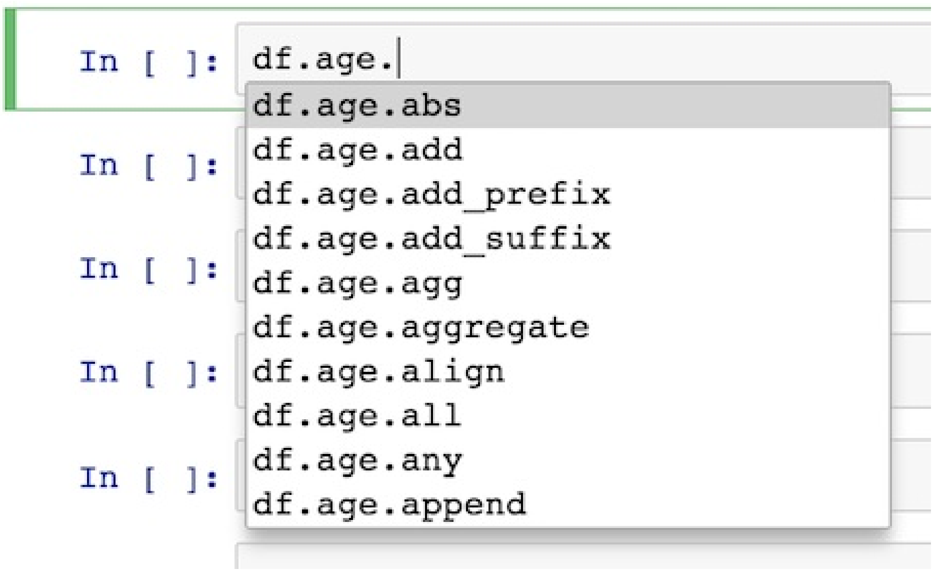
Pandas allows you to select a single column as a Series by using **dot notation**. This is also referred to as **attribute access**. You simply place the name of the column without quotes following a dot and the DataFrame like this:

```
>>> df.state
Jane          NY
Niko          TX
Aaron         FL
Penelope      AL
Dean          AK
Christina     TX
Cornelia      TX
Name: state, dtype: object

>>> df.age
Jane          30
Niko           2
Aaron         12
Penelope       4
Dean          32
Christina     33
Cornelia      69
Name: age, dtype: int64
```

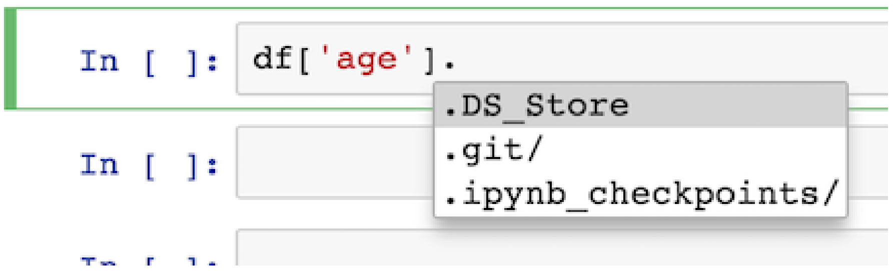
Pros and cons when selecting columns by attribute access

The best benefit of selecting columns like this is that you get help when chaining methods after selection. For instance, if you place another dot after the column name and press **tab**, a list of all the Series methods will appear in a pop-up menu. It will look like this:



```
In [ ]: df.age.
        df.age.abs
        df.age.add
        df.age.add_prefix
        df.age.add_suffix
        df.age.agg
        df.age.aggregate
        df.age.align
        df.age.all
        df.age.any
        df.age.append
In [ ]:
In [ ]:
In [ ]:
In [ ]:
```

This help disappears when you use just the indexing operator:



```
In [ ]: df['age'].
                   .DS_Store
                   .git/
                   .ipynb_checkpoints/
In [ ]:
```

The biggest drawback is that you cannot select columns that have spaces or other characters that are not valid as Python identifiers (variable names).

Selecting the same column twice?

This is rather peculiar, but you can actually select the same column more than once:

```
df[['age', 'age', 'age']]
```

	age	age	age
Jane	30	30	30
Niko	2	2	2
Aaron	12	12	12
Penelope	4	4	4
Dean	32	32	32
Christina	33	33	33
Cornelia	69	69	69

Summary of Part 1

We covered an incredible amount of ground. Let's summarize all the main points:

- Before learning pandas, ensure you have the fundamentals of Python
- Always refer to the documentation when learning new pandas operations
- The DataFrame and the Series are the containers of data
- A DataFrame is two-dimensional, tabular data
- A Series is a single dimension of data
- The three components of a DataFrame are the **index**, the **columns** and the **data** (or **values**)
- Each row and column of the DataFrame is referenced by both a **label** and an **integer location**
- There are three primary ways to select subsets from a DataFrame — `[]`, `.loc` and `.iloc`
- I use the term **just the indexing operator** to refer to `[]` immediately following a DataFrame/Series
- Just the indexing operator's primary purpose is to select a column or columns from a DataFrame
- Using a single column name to just the indexing operator returns a single column of data as a Series
- Passing multiple columns in a list to just the indexing operator returns a DataFrame
- A Series has two components, the **index** and the **data** (**values**). It has no columns
- `.loc` makes selections **only by label**
- `.loc` can simultaneously select rows and columns
- `.loc` can make selections with either a single label, a list of labels, or a slice of labels
- `.loc` makes row selections first followed by column selections: `df.loc[row_selection, col_selection]`

- `.iloc` is analogous to `.loc` but uses only **integer location** to refer to rows or columns.
- `.ix` is deprecated and should never be used
- `.loc` and `.iloc` work the same for Series except they only select based on the index as their are no columns
- Pandas combines the power of python lists (selection via integer location) and dictionaries (selection by label)
- You can use just the indexing operator to select rows from a DataFrame, but I recommend against this and instead sticking with the explicit `.loc` and `.iloc`
- Normally data is imported without setting an index. Use the `set_index` method to use a column as an index.
- You can select a single column as a Series from a DataFrame with dot notation

Way more to the story

This is only part 1 of the series, so there is much more to cover on how to select subsets of data in pandas. Some of the explanations in this part will be expanded to include other possibilities.

Exercises

This best way to learn pandas is to practice on your own. All these exercises will use the the Chicago food inspections dataset found here at data.world.

- Download the Jupyter Notebook to get started on the exercises.
- Make sure to review the detailed solutions as well after you attempt the exercises