


Python Libraries for Data Analysis and Machine Learning

Overview

- Environment Preparation for Python
 - Python Libraries for Data Scientists
 - Data Processing & Visualization Using Python
 - Python for Basic Machine Learning Models
- 
- A solid orange horizontal bar spanning the width of the slide, located at the bottom.

Environment Preparation for Python

We introduce

- **Anaconda** (<https://www.anaconda.com/>)
- **Jupyter Notebook** (<https://jupyter.org/>)

for Python environment.

Other alternatives:

- Text Editor + Command line
- IDE (Integrated Development Environment): PyCharm, Vscode, ...

What is Anaconda?

- The open-source Anaconda is the easiest way to perform Python/R data science and machine learning on Linux, Windows, and Mac OS X. With over 19 million users worldwide.
- It is the industry standard for developing, testing, and training on a single machine, enabling *individual data scientists* to:
 - Quickly download 7,500+ Python/R data science packages
 - Analyze data with scalability and performance with Dask, **NumPy**, **pandas**, and Numba
 - Visualize results with **Matplotlib**, Bokeh, Datashader, and Holoviews
 - Develop and train machine learning and deep learning models with **scikit-learn**, TensorFlow, and Theano

Anaconda Installation

Please follow the instruction here to install the Anaconda (for Python 3.7)

<https://www.anaconda.com/distribution/#download-section>

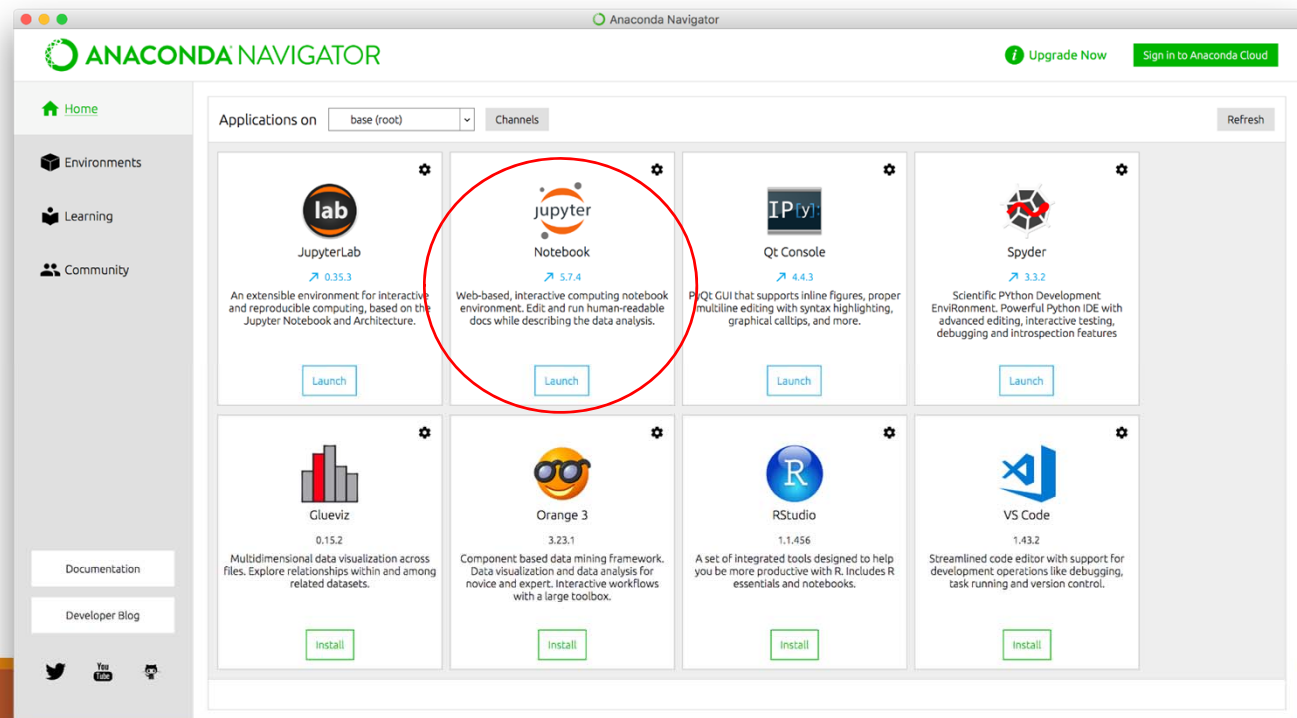
- It provides different versions to suit different OS. Please select the one you are using.
- Just install according to the default setting, and the environment variables will be automatically configured after installation.

What is Jupyter Notebook?

- The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text.
- It includes: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.
- Jupyter Notebook is included in the Anaconda.


Basic Operation on Jupyter Notebook

After installing the Anaconda, open Anaconda-Navigator as below, and you can find the Jupyter Notebook on the Anaconda. Then click Launch.



Basic Operation on Jupyter Notebook

Jupyter Notebook is presented as a website. Select the path, then under the button “New”, choose “Python 3” to open a new python file.




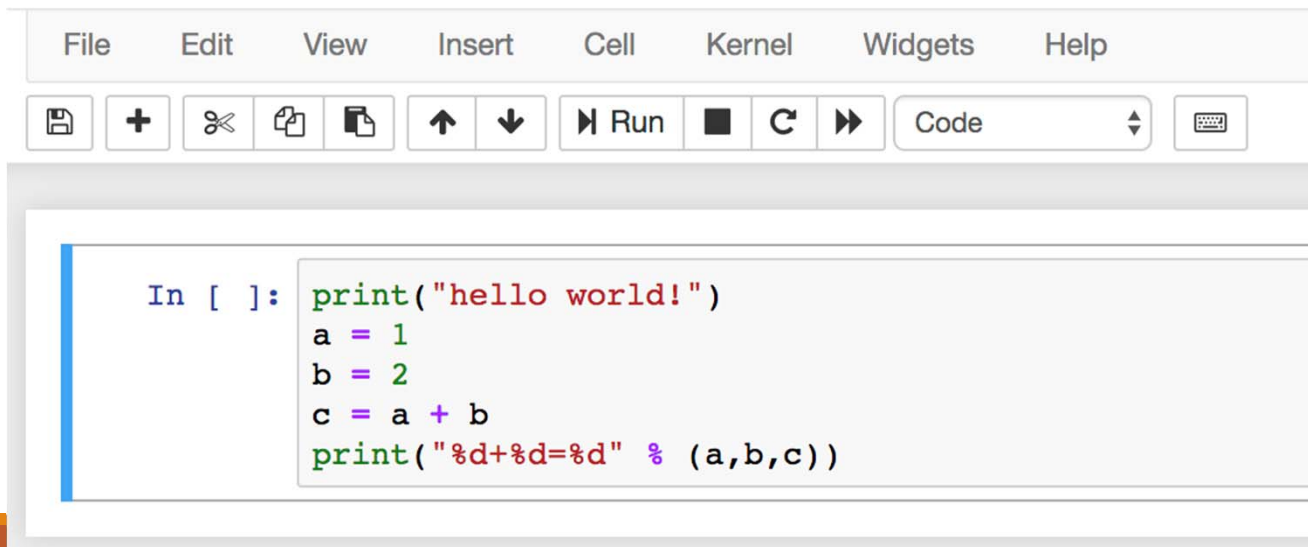
The screenshot displays the Jupyter Notebook web interface in a browser. The address bar shows the URL `localhost:8888/tree/Desktop/ecit5810`. The Jupyter logo is visible on the left, and 'Quit' and 'Logout' buttons are on the right. Below the logo, there are tabs for 'Files', 'Running', and 'Clusters'. A message 'Select items to perform actions on them.' is shown. The file browser shows the current directory as `/ Desktop / ecit5810`. A list of files is displayed, including `Untitled.ipynb`, `assignment-1-answer.arff`, `bank-additional-test.arff`, and `bank-additional.csv`. The 'New' button is clicked, opening a dropdown menu. The 'Notebook:' section is expanded, and 'Python 3' is selected and circled in red. Other options in the menu include 'Text File', 'Folder', and 'Terminal'. The 'Upload' button is also visible.

Basic Operation on Jupyter Notebook

Type the code into the input box on Jupyter.

Get started learning Python: <https://www.learnpython.org/>

 jupyter **Untitled1** Last Checkpoint: 11 minutes ago (autosaved)



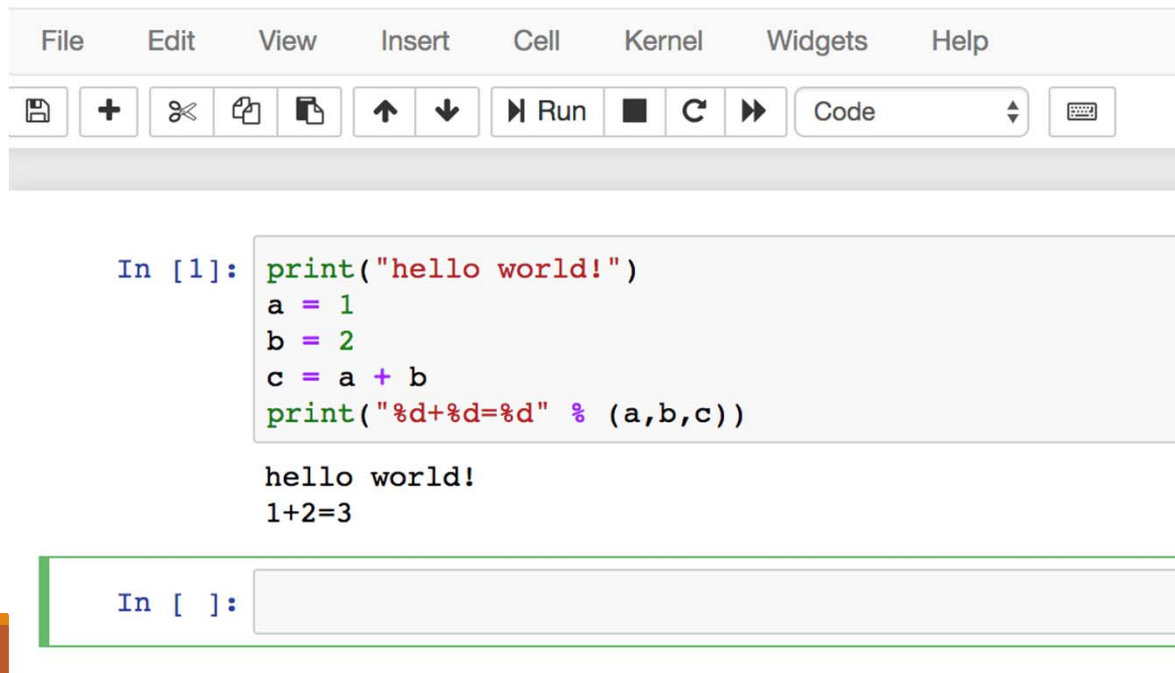
The screenshot shows the Jupyter Notebook interface. At the top, there is a menu bar with options: File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. Below the menu bar is a toolbar with icons for saving, adding, deleting, copying, pasting, undo, redo, and running code. The main area contains a code cell with the following Python code:

```
In [ ]: print("hello world!")
a = 1
b = 2
c = a + b
print("%d+%d=%d" % (a,b,c))
```

Basic Operation on Jupyter Notebook

Click “Run”.

The output will be shown in the blank area right below the input box.



Basic Operation on Jupyter Notebook

Jupyter Notebook will help you save your code automatically in “.ipynb” format.

If you want to save the code as “.py” format.

Here, we just use “.ipynb” format.

```
In [1]: print("hello world!")
a = 1
b = 2
c = a + b
print("%d+%d=%d" % (a,b,c))
```

```
hello world!
1+2=3
```

```
In [2]: %%writefile example.py
print("hello world!")
a = 1
b = 2
c = a + b
print("%d+%d=%d" % (a,b,c))
```

```
Writing example.py
```

```
In [ ]: |
```

Python Libraries for Data Scientists

Python toolboxes/libraries for data processing:

- NumPy
- SciPy
- Pandas

Visualization libraries

- matplotlib
- Seaborn

Machine learning & deep learning

- Scikit-learn
- Tensorflow/Pytorch/Theano

and many more ...

Python Libraries for Data Scientists

NumPy:

- introduces objects for multidimensional arrays and matrices, as well as functions that allow to easily perform advanced mathematical and statistical operations on those objects
- provides vectorization of mathematical operations on arrays and matrices which significantly improves the performance
- many other python libraries are built on NumPy

Link: <http://www.numpy.org/>



Python Libraries for Data Scientists

SciPy:

- collection of algorithms for linear algebra, differential equations, numerical integration, optimization, statistics and more
- part of SciPy Stack
- built on NumPy

Link: <https://www.scipy.org/scipylib/>



Python Libraries for Data Scientists

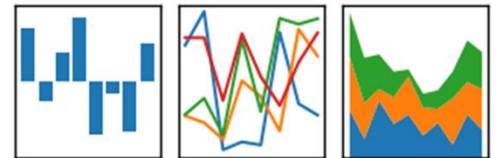
Pandas:

- adds data structures and tools designed to work with table-like data (similar to Series and Data Frames in R)
- provides tools for data manipulation: reshaping, merging, sorting, slicing, aggregation etc.
- allows handling missing data

Link: <http://pandas.pydata.org/>

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Python Libraries for Data Scientists

matplotlib:

- python 2D plotting library which produces publication quality figures in a variety of hardcopy formats
- a set of functionalities similar to those of MATLAB
- line plots, scatter plots, barcharts, histograms, pie charts etc.
- relatively low-level; some effort needed to create advanced visualization

Link: <https://matplotlib.org/>



Python Libraries for Data Scientists

Seaborn:

- based on matplotlib
- provides high level interface for drawing attractive statistical graphics
- Similar (in style) to the popular ggplot2 library in R

Link: <https://seaborn.pydata.org/>

Python Libraries for Data Scientists

SciKit-Learn:

- provides machine learning algorithms: classification, regression, clustering, model validation etc.
- built on NumPy, SciPy and matplotlib

Link: <http://scikit-learn.org/>



Loading Python Libraries

```
In [1]: #Import Python Libraries  
import numpy as np  
import scipy as sp  
import pandas as pd  
import matplotlib.pyplot as plt  
import seaborn as sns
```

Press Shift+Enter to execute the *jupyter* cell, or just click “Run”.

Reading data using pandas

```
In [4]: df = pd.read_csv("http://www1.se.cuhk.edu.hk/~eclt5810/lecture/weka_tutorial/bank.csv")
```

There is a number of pandas commands to read other data formats:

```
pd.read_excel('myfile.xlsx', sheet_name='Sheet1', index_col=None, na_values=['NA'])  
pd.read_stata('myfile.dta')  
pd.read_sas('myfile.sas7bdat')  
pd.read_hdf('myfile.h5', 'df')
```

Exploring data frames

```
In [5]: #List first 5 records  
df.head()
```

Out[5]:

| | age | job | marital | education | default | balance | housing | loan | contact | day | month | duration | campaign | pdays | previous | poutcome | y |
|---|-----|-------------|---------|-----------|---------|---------|---------|------|----------|-----|-------|----------|----------|-------|----------|----------|----|
| 0 | 30 | unemployed | married | primary | no | 1787 | no | no | cellular | 19 | oct | 79 | 1 | -1 | 0 | unknown | no |
| 1 | 33 | services | married | secondary | no | 4789 | yes | yes | cellular | 11 | may | 220 | 1 | 339 | 4 | failure | no |
| 2 | 35 | management | single | tertiary | no | 1350 | yes | no | cellular | 16 | apr | 185 | 1 | 330 | 1 | failure | no |
| 3 | 30 | management | married | tertiary | no | 1476 | yes | yes | unknown | 3 | jun | 199 | 4 | -1 | 0 | unknown | no |
| 4 | 59 | blue-collar | married | secondary | no | 0 | yes | no | unknown | 5 | may | 226 | 1 | -1 | 0 | unknown | no |

- ✓ Try to read the first 10, 20, 50 records
- ✓ Try to view the last few records

Data Frame data types

| Pandas Type | Native Python Type | Description |
|---------------------------|--|--|
| object | string | The most general dtype. Will be assigned to your column if column has mixed types (numbers and strings). |
| int64 | int | Numeric characters. 64 refers to the memory allocated to hold this character. |
| float64 | float | Numeric characters with decimals. If a column contains numbers and NaNs, pandas will default to float64, in case your missing value has a decimal. |
| datetime64, timedelta[ns] | N/A (but see the datetime module in Python's standard library) | Values meant to hold time data. Look into these for time series experiments. |

Data Frame data types

```
In [7]: #Check a particular column type  
df['age'].dtype
```

```
Out[7]: dtype('int64')
```

```
In [8]: #Check types for all the columns  
df.dtypes
```

```
Out[8]: age           int64  
        job           object  
        marital       object  
        education     object  
        default       object  
        balance       int64  
        housing       object  
        loan          object  
        contact       object  
        day           int64  
        month         object  
        duration      int64  
        campaign      int64  
        pdays         int64  
        previous      int64  
        poutcome      object  
        y             object  
        dtype: object
```

Data Frames attributes

Python objects have *attributes* and *methods*.

| df.attribute | description |
|--------------|--|
| dtypes | list the types of the columns |
| columns | list the column names |
| axes | list the row labels and column names |
| ndim | number of dimensions |
| size | number of elements |
| shape | return a tuple representing the dimensionality |
| values | numpy representation of the data |

Data Frames attributes

```
In [11]: #Find how many records this data frame has  
df.shape
```

```
Out[11]: (4521, 17)
```

```
In [12]: #How many elements are there?  
df.size
```

```
Out[12]: 76857
```

```
In [13]: #What are the column names?  
df.columns
```

```
Out[13]: Index(['age', 'job', 'marital', 'education', 'default', 'balance', 'housing',  
               'loan', 'contact', 'day', 'month', 'duration', 'campaign', 'pdays',  
               'previous', 'poutcome', 'y'],  
              dtype='object')
```

Data Frames methods

Unlike attributes, python methods have *parenthesis*.

All attributes and methods can be listed with a *dir()* function: **dir(df)**

| df.method() | description |
|--------------------------|--|
| head([n]), tail([n]) | first/last n rows |
| describe() | generate descriptive statistics (for numeric columns only) |
| max(), min() | return max/min values for all numeric columns |
| mean(), median() | return mean/median values for all numeric columns |
| std() | standard deviation |
| sample([n]) | returns a random sample of the data frame |
| dropna() | drop all the records with missing values |

Data Frames methods

```
In [14]: #Give the summary for the numeric columns in the dataset  
df.describe()
```

Out[14]:

| | age | balance | day | duration | campaign | pdays | previous |
|--------------|-------------|--------------|-------------|-------------|-------------|-------------|-------------|
| count | 4521.000000 | 4521.000000 | 4521.000000 | 4521.000000 | 4521.000000 | 4521.000000 | 4521.000000 |
| mean | 41.170095 | 1422.657819 | 15.915284 | 263.961292 | 2.793630 | 39.766645 | 0.542579 |
| std | 10.576211 | 3009.638142 | 8.247667 | 259.856633 | 3.109807 | 100.121124 | 1.693562 |
| min | 19.000000 | -3313.000000 | 1.000000 | 4.000000 | 1.000000 | -1.000000 | 0.000000 |
| 25% | 33.000000 | 69.000000 | 9.000000 | 104.000000 | 1.000000 | -1.000000 | 0.000000 |
| 50% | 39.000000 | 444.000000 | 16.000000 | 185.000000 | 2.000000 | -1.000000 | 0.000000 |
| 75% | 49.000000 | 1480.000000 | 21.000000 | 329.000000 | 3.000000 | -1.000000 | 0.000000 |
| max | 87.000000 | 71188.000000 | 31.000000 | 3025.000000 | 50.000000 | 871.000000 | 25.000000 |

Data Frames methods

```
In [15]: #Calculate standard deviation for all numeric columns  
df.std()
```

```
Out[15]: age          10.576211  
balance    3009.638142  
day         8.247667  
duration   259.856633  
campaign    3.109807  
pdays     100.121124  
previous    1.693562  
dtype: float64
```

```
In [17]: #What are the mean values for all numeric columns?  
df.mean()
```

```
Out[17]: age          41.170095  
balance    1422.657819  
day        15.915284  
duration   263.961292  
campaign    2.793630  
pdays     39.766645  
previous    0.542579  
dtype: float64
```

Selecting a column in a Data Frame

```
In [22]: #Subset the data frame using column name  
df['job'][:5]
```

```
Out[22]: 0    unemployed  
        1      services  
        2    management  
        3    management  
        4  blue-collar  
        Name: job, dtype: object
```

```
In [23]: #Use the column name as an attribute  
df.job[:5]
```

```
Out[23]: 0    unemployed  
        1      services  
        2    management  
        3    management  
        4  blue-collar  
        Name: job, dtype: object
```

Note: If we want to select a column with a name as the attribute in DataFrames we should use method 1.

E.G., Since there is an attribute – *rank* in DataFrame, if we want to select the column ‘rank’, we should use `df['rank']`, and cannot use method 2, i.e., `df.rank`, which will return the attribute *rank* of the data frame instead of the column “rank”.

Selecting a column in a Data Frame

```
In [24]: #Calculate the basic statistics for the age column  
df.age.describe()
```

```
Out[24]: count      4521.000000  
         mean        41.170095  
         std         10.576211  
         min         19.000000  
         25%         33.000000  
         50%         39.000000  
         75%         49.000000  
         max         87.000000  
         Name: age, dtype: float64
```

```
In [26]: #Calculate the average age  
df.age.mean()
```

```
Out[26]: 41.17009511170095
```

```
In [25]: #Find how many values in the age column (use count method)  
df.age.count()
```

```
Out[25]: 4521
```

Data Frames *groupby* method

Using "group by" method we can:

- Split the data into groups based on some criteria
- Calculate statistics (or apply a function) to each group

```
In [27]: #Group data using rank  
df_job = df.groupby(['job'])
```

```
In [28]: #Calculate mean value for each numeric column per each group  
df_job.mean()
```

Out[28]:

| | age | balance | day | duration | campaign | pdays | previous |
|--------------|-----------|-------------|-----------|------------|----------|-----------|----------|
| job | | | | | | | |
| admin. | 39.682008 | 1226.736402 | 16.324268 | 234.669456 | 2.631799 | 49.993724 | 0.644351 |
| blue-collar | 40.156448 | 1085.161734 | 15.482030 | 278.161734 | 2.846723 | 41.590909 | 0.493658 |
| entrepreneur | 42.011905 | 1645.125000 | 15.255952 | 285.476190 | 2.589286 | 32.273810 | 0.428571 |
| housemaid | 47.339286 | 2083.803571 | 15.294643 | 292.633929 | 2.500000 | 26.401786 | 0.357143 |

Data Frames *groupby* method

Once *groupby* object is create we can calculate various statistics for each group:

```
In [29]: #Calculate mean age for each job:  
df.groupby('job')[['age']].mean()
```

Out[29]:

| age | |
|--------------|-----------|
| job | |
| admin. | 39.682008 |
| blue-collar | 40.156448 |
| entrepreneur | 42.011905 |
| housemaid | 47.339286 |
| management | 40.540764 |
| retired | 61.869565 |

Note: If single brackets are used to specify the column (e.g. age), then the output is Pandas Series object. When double brackets are used the output is a Data Frame (e.g. age & balance)

Data Frames *groupby* method

groupby performance notes:

- no grouping/splitting occurs until it's needed. Creating the *groupby* object only verifies that you have passed a valid mapping
- by default the group keys are sorted during the *groupby* operation. You may want to pass `sort=False` for potential speedup:

```
In [30]: #Calculate mean age for each job:  
df.groupby(['job'], sort=False)[['age']].mean()
```

Data Frame: filtering

To subset the data we can apply Boolean indexing. This indexing is commonly known as a filter. For example if we want to subset the rows in which the age value is greater than 50:

```
In [31]: #Subset the rows in which the age value is greater than 50  
df_sub = df[ df['age'] > 50 ]
```

Any Boolean operator can be used to subset the data:

> greater; >= greater or equal;
< less; <= less or equal;
== equal; != not equal;

```
In [32]: #Select only those rows whose education level is primary  
df_primary = df[ df['education'] == 'primary' ]
```

Data Frames: Slicing

There are a number of ways to subset the Data Frame:

- one or more columns
- one or more rows
- a subset of rows and columns

Rows and columns can be selected by their position or label

Data Frames: Slicing

When selecting one column, it is possible to use single set of brackets, but the resulting object will be a Series (not a DataFrame):

```
In [ ]: #Select column age:  
df['age']
```

When we need to select more than one column and/or make the output to be a DataFrame, we should use double brackets:

```
In [ ]: #Select column age and job:  
df[['age', 'job']]
```

Data Frames: Selecting rows

If we need to select a range of rows, we can specify the range using ":"

```
In [ ]: #Select rows by their position:  
df[10:20]
```

Notice that the first row has a position 0, and the last value in the range is omitted:
So for 0:10 range the first 10 rows are returned with the positions starting with 0
and ending with 9

Data Frames: method loc

If we need to select a range of rows, using their labels we can use method loc:

```
In [35]: #Select rows by their labels:  
df_sub.loc[10:20,['age','job','education']]
```

Out[35]:

| | age | job | education |
|----|-----|------------|-----------|
| 16 | 56 | technician | secondary |

Recall that In [31]: *#Subset the rows in which the age value is greater than 50*
df_sub = df[df['age'] > 50]

Data Frames: method iloc

If we need to select a range of rows and/or columns, using their positions we can use method iloc:

```
In [47]: #Select rows by their labels:  
df_sub.iloc[10:20,[0, 1, 3]]
```

Out[47]:

| | age | job | education |
|-----------|-----|-------------|-----------|
| 46 | 55 | blue-collar | primary |
| 49 | 61 | admin. | unknown |
| 54 | 53 | blue-collar | secondary |
| 56 | 57 | management | secondary |
| 59 | 54 | technician | secondary |
| 61 | 63 | retired | secondary |

Data Frames: method iloc (summary)

```
df.iloc[0]    # First row of a data frame  
df.iloc[i]    #(i+1)th row  
df.iloc[-1]   # Last row
```

```
df.iloc[:, 0] # First column  
df.iloc[:, -1] # Last column
```

```
df.iloc[0:7]      #First 7 rows  
df.iloc[:, 0:2]    #First 2 columns  
df.iloc[1:3, 0:2]  #Second through third rows and first 2 columns  
df.iloc[[0,5], [1,3]] #1st and 6th rows and 2nd and 4th columns
```


Data Frames: Sorting

We can sort the data by a value in the column. By default the sorting will occur in ascending order and a new data frame is return.

```
In [50]: # Create a new data frame from the original sorted by the column Age
df_sorted = df.sort_values( by ='age')
df_sorted.head()
```

Out[50]:

| | age | job | marital | education | default | balance | housing | loan | contact | day | month | duration | campaign | pdays | previous | poutcome | y |
|------|-----|---------|---------|-----------|---------|---------|---------|------|-----------|-----|-------|----------|----------|-------|----------|----------|-----|
| 503 | 19 | student | single | primary | no | 103 | no | no | cellular | 10 | jul | 104 | 2 | -1 | 0 | unknown | yes |
| 1900 | 19 | student | single | unknown | no | 0 | no | no | cellular | 11 | feb | 123 | 3 | -1 | 0 | unknown | no |
| 2780 | 19 | student | single | secondary | no | 302 | no | no | cellular | 16 | jul | 205 | 1 | -1 | 0 | unknown | yes |
| 3233 | 19 | student | single | unknown | no | 1169 | no | no | cellular | 6 | feb | 463 | 18 | -1 | 0 | unknown | no |
| 999 | 20 | student | single | secondary | no | 291 | no | no | telephone | 11 | may | 172 | 5 | 371 | 5 | failure | no |

Data Frames: Sorting

We can sort the data using 2 or more columns:

```
In [51]: df_sorted = df.sort_values( by=['age', 'balance'], ascending=[True, False])
df_sorted.head(10)
```

Out[51]:

| | age | job | marital | education | default | balance | housing | loan | contact | day | month | duration | campaign | pdays | previous | poutcome | y |
|------|-----|----------|---------|-----------|---------|---------|---------|------|-----------|-----|-------|----------|----------|-------|----------|----------|-----|
| 3233 | 19 | student | single | unknown | no | 1169 | no | no | cellular | 6 | feb | 463 | 18 | -1 | 0 | unknown | no |
| 2780 | 19 | student | single | secondary | no | 302 | no | no | cellular | 16 | jul | 205 | 1 | -1 | 0 | unknown | yes |
| 503 | 19 | student | single | primary | no | 103 | no | no | cellular | 10 | jul | 104 | 2 | -1 | 0 | unknown | yes |
| 1900 | 19 | student | single | unknown | no | 0 | no | no | cellular | 11 | feb | 123 | 3 | -1 | 0 | unknown | no |
| 1725 | 20 | student | single | secondary | no | 1191 | no | no | cellular | 12 | feb | 274 | 1 | -1 | 0 | unknown | no |
| 13 | 20 | student | single | secondary | no | 502 | no | no | cellular | 30 | apr | 261 | 1 | -1 | 0 | unknown | yes |
| 999 | 20 | student | single | secondary | no | 291 | no | no | telephone | 11 | may | 172 | 5 | 371 | 5 | failure | no |
| 4152 | 21 | student | single | secondary | no | 6844 | no | no | cellular | 14 | aug | 126 | 3 | 127 | 7 | other | no |
| 110 | 21 | student | single | secondary | no | 2488 | no | no | cellular | 30 | jun | 258 | 6 | 169 | 3 | success | yes |
| 2046 | 21 | services | single | secondary | no | 1903 | yes | no | unknown | 29 | may | 107 | 2 | -1 | 0 | unknown | no |

Aggregation Functions in Pandas

Aggregation - computing a summary statistic about each group, i.e.

- compute group sums or means
- compute group sizes/counts

Common aggregation functions:

min, max

count, sum, prod

mean, median, mode, mad

std, var

Aggregation Functions in Pandas

agg() method are useful when multiple statistics are computed per column:

```
In [28]: df[['age', 'balance']].agg(['min', 'mean', 'max'])
```

Out[28]:

| | age | balance |
|------|-----------|--------------|
| min | 19.000000 | -3313.000000 |
| mean | 41.170095 | 1422.657819 |
| max | 87.000000 | 71188.000000 |

Basic Descriptive Statistics

| df.method() | description |
|--------------------|--|
| describe | Basic statistics (count, mean, std, min, quantiles, max) |
| min, max | Minimum and maximum values |
| mean, median, mode | Arithmetic average, median and mode |
| var, std | Variance and standard deviation |
| sem | Standard error of mean |
| skew | Sample skewness |
| kurt | kurtosis |

Graphics to explore the data

Seaborn package is built on matplotlib but provides high level interface for drawing attractive statistical graphics, similar to ggplot2 library in R. It specifically targets statistical data visualization

To show graphs within Python notebook include inline directive:

```
In [ ]: %matplotlib inline
```

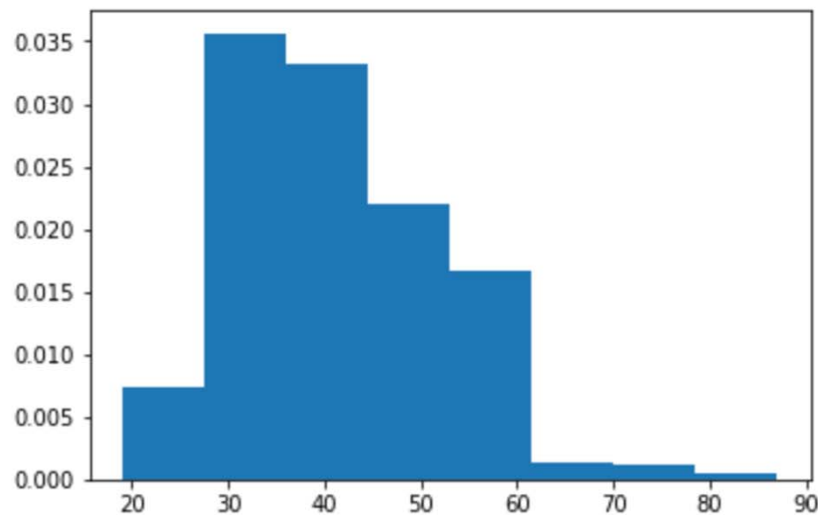
Graphics

| | description |
|------------|--|
| histplot | histogram |
| barplot | estimate of central tendency for a numeric variable |
| violinplot | similar to boxplot, also shows the probability density of the data |
| jointplot | Scatterplot |
| regplot | Regression plot |
| pairplot | Pairplot |
| boxplot | boxplot |
| swarmplot | categorical scatterplot |
| factorplot | General categorical plot |

Draw Histogram Using Matplotlib

```
In [31]: #Use matplotlib to draw a histogram of a age data  
plt.hist(df['age'],bins=8, density=1)
```

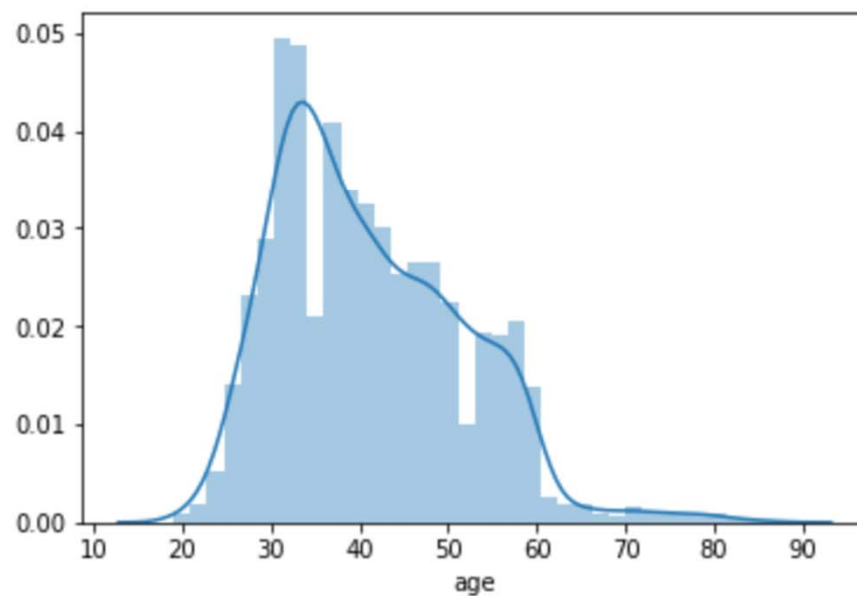
```
Out[31]: (array([0.0073383 , 0.03565062, 0.03320452, 0.02193684, 0.01662828,  
                0.00130112, 0.00111896, 0.0004684 ]),  
         array([19. , 27.5, 36. , 44.5, 53. , 61.5, 70. , 78.5, 87. ]),  
         <a list of 8 Patch objects>)
```



Draw Histogram Using Seaborn

```
In [33]: #Use seaborn package to draw a histogram  
sns.distplot(df[('age')])
```

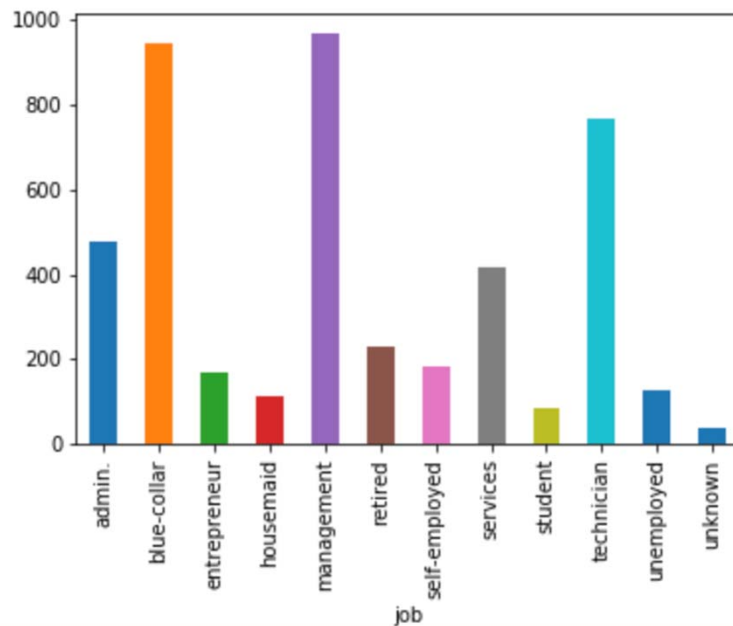
```
Out[33]: <matplotlib.axes._subplots.AxesSubplot at 0x1a21b0cb00>
```



Draw Barplot Using Matplotlib

```
In [41]: # Use regular matplotlib function to display a barplot  
df.groupby(['job'])['age'].count().plot(kind='bar')
```

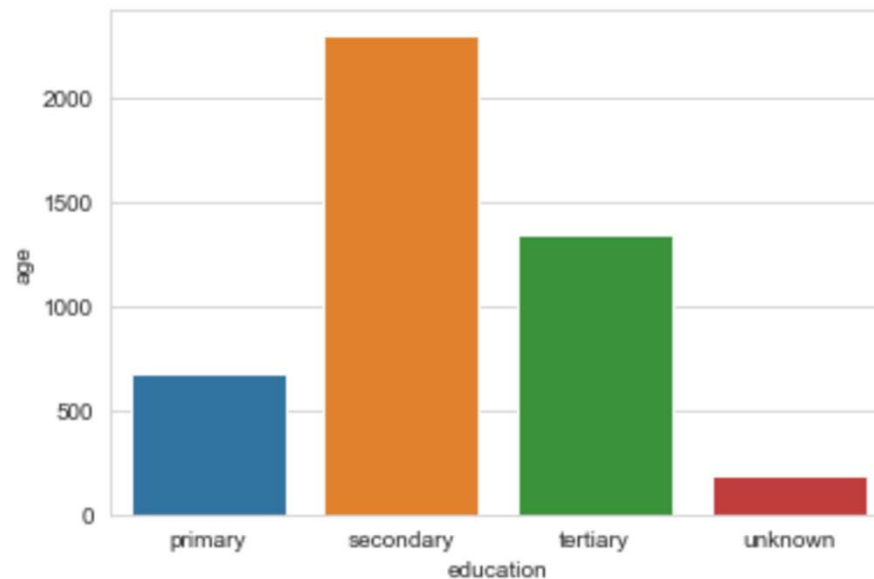
```
Out[41]: <matplotlib.axes._subplots.AxesSubplot at 0x1a227f29e8>
```



Draw Barplot Using Seaborn

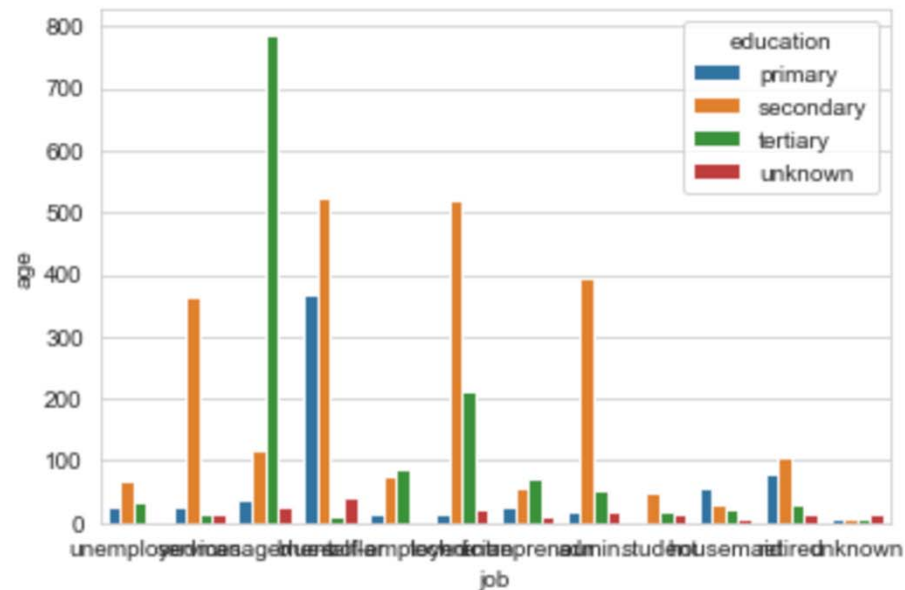
```
In [43]: # Use seaborn package to display a barplot
sns.set_style("whitegrid")

ax = sns.barplot(x='education', y='age', data=df, estimator=len)
```



Draw Barplot Using Seaborn

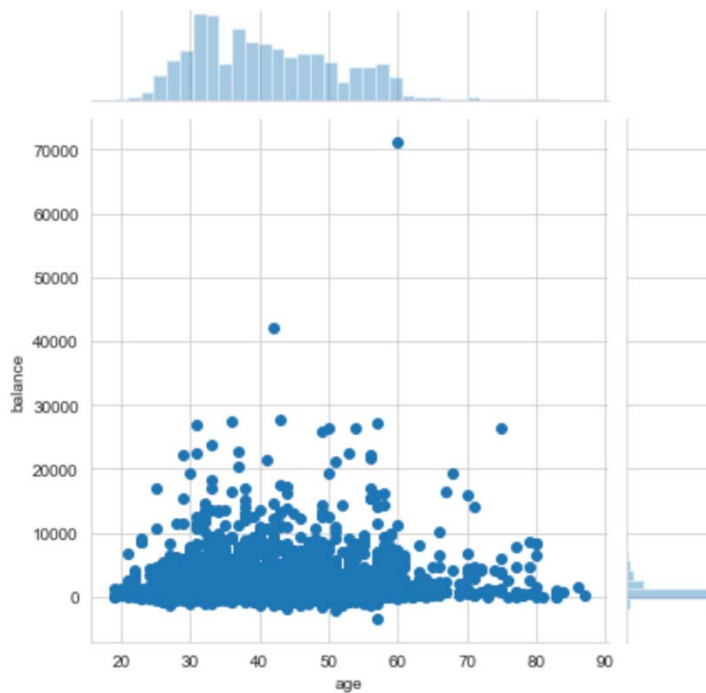
```
In [54]: # Split into groups based on education:  
ax = sns.barplot(x='job', y='age', hue='education', data=df, estimator=len)
```



Draw Scatterplot Using Seaborn

```
In [58]: #Scatterplot in seaborn  
sns.jointplot(x='age', y='balance', data=df)
```

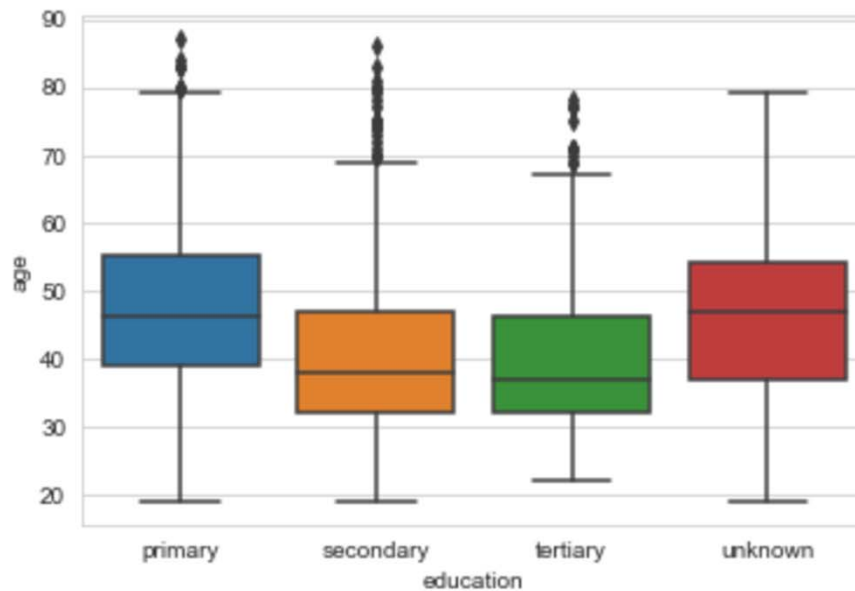
```
Out[58]: <seaborn.axisgrid.JointGrid at 0x1a22d9aeb8>
```



Draw Boxplot Using Seaborn

```
In [60]: # box plot  
sns.boxplot(x='education', y='age', data=df)
```

```
Out[60]: <matplotlib.axes._subplots.AxesSubplot at 0x1a22f2cac8>
```



Python for Machine Learning

Machine learning: the problem setting:

In general, a learning problem considers a set of n samples of data and then tries to predict properties of unknown data. If each sample is more than a single number and, for instance, a multi-dimensional entry (aka multivariate data), it is said to have several attributes or features.

We can separate learning problems in a few large categories:

- Supervised Learning (https://sklearn.org/supervised_learning.html#supervised-learning)
 - Classification
 - Regression
- Unsupervised Learning (https://sklearn.org/unsupervised_learning.html#unsupervised-learning)
 - Clustering

Python for Machine Learning

Training set and testing set:

Machine learning is about learning some properties of a data set and applying them to new data. This is why a common practice in machine learning to evaluate an algorithm is to split the data at hand into two sets, one that we call the **training set** on which we learn data properties and one that we call the **testing set** on which we test these properties.

scikit-learn comes with a few standard datasets, for instance the **iris** and **digits** datasets for **classification** and the **boston house prices** dataset for **regression**.

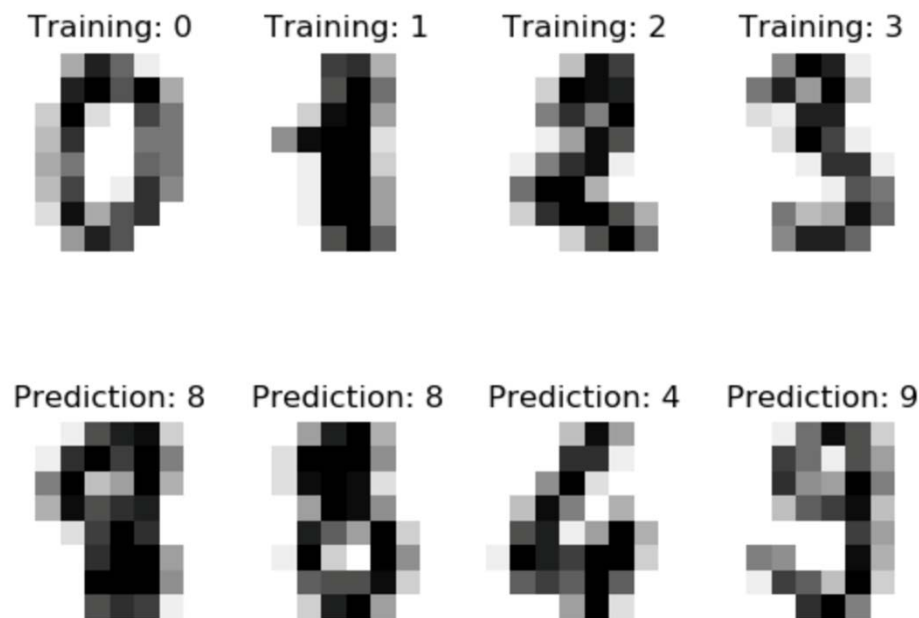
Loading an example dataset

```
In [61]: import sklearn  
         from sklearn import datasets  
         iris = datasets.load_iris()  
         digits = datasets.load_digits()
```

A dataset is a dictionary-like object that holds all the data and some metadata about the data. This data is stored in the *.data* member, which is a *(n_samples, n_features)* array. In the case of supervised problem, one or more response variables are stored in the *.target* member.

Loading an example dataset - *digits*

An example showing how the scikit-learn can be used to recognize images of hand-written digits.



Loading an example dataset - *digits*

For instance, in the case of the digits dataset, *digits.data* gives access to the features that can be used to classify the digits samples:

```
In [62]: print(digits.data)
[[ 0.  0.  5. ...  0.  0.  0.]
 [ 0.  0.  0. ... 10.  0.  0.]
 [ 0.  0.  0. ... 16.  9.  0.]
 ...
 [ 0.  0.  1. ...  6.  0.  0.]
 [ 0.  0.  2. ... 12.  0.  0.]
 [ 0.  0. 10. ... 12.  1.  0.]]
```

and *digits.target* gives the ground truth for the digit dataset, that is the number corresponding to each digit image that we are trying to learn:

```
In [63]: print(digits.target)
[0 1 2 ... 8 9 8]
```

Learning and predicting

In the case of the *digits* dataset, the task is to predict, given an image, which digit it represents. We are given samples of each of the 10 possible classes (the digits *zero* through *nine*) on which we *fit* a **classifier** to be able to *predict* the classes to which unseen samples belong.

In scikit-learn, a classifier for classification is a Python object that implements the methods `fit(X, y)` and `predict(T)`.

An example of a classifier is the class `sklearn.svm.SVC`, which implements **support vector classification**. The classifier's constructor takes as arguments the model's parameters.

Learning and predicting

For now, we will consider the classifier as a black box:

```
In [41]: from sklearn import svm  
clf = svm.SVC(gamma=0.001, C=100.)
```

Choosing the parameters of the model

In this example, we set the value of gamma manually. To find good values for these parameters, we can use tools such as [grid search](#) and [cross validation](#).

Learning and predicting

For the training set, we'll use all the images from our dataset, except for the last image, which we'll reserve for our predicting. We select the training set with the `[:-1]` Python syntax, which produces a new array that contains all but the last item from `digits.data`:

```
In [42]: clf.fit(digits.data[:-1], digits.target[:-1])
```

```
Out[42]: SVC(C=100.0, cache_size=200, class_weight=None, coef0=0.0,  
            decision_function_shape='ovr', degree=3, gamma=0.001, kernel='rbf',  
            max_iter=-1, probability=False, random_state=None, shrinking=True,  
            tol=0.001, verbose=False)
```

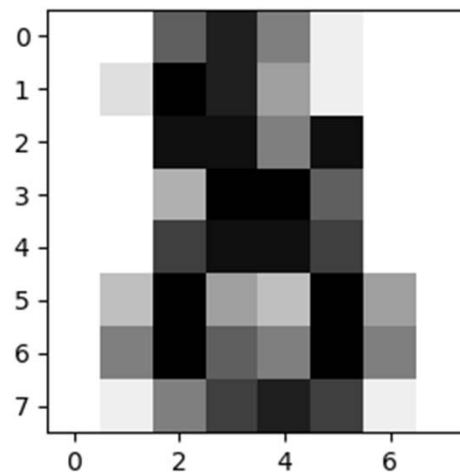
Learning and predicting

Now you can *predict* new values. In this case, you'll predict using the last image from `digits.data`. By predicting, you'll determine the image from the training set that best matches the last image.

```
In [43]: clf.predict(digits.data[-1:])
```

```
Out[43]: array([8])
```

The corresponding image is:



Model persistence

It is possible to save a model in scikit-learn by using [pickle](#):

```
In [45]: clf = svm.SVC(gamma='auto')
X, y = datasets.load_iris(return_X_y=True)
clf.fit(X, y)
```

```
Out[45]: SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
max_iter=-1, probability=False, random_state=None, shrinking=True,
tol=0.001, verbose=False)
```

```
In [46]: import pickle
s = pickle.dumps(clf)
clf2 = pickle.loads(s)
clf2.predict(X[0:1])
```

```
Out[46]: array([0])
```

```
In [47]: print(y[0])
```

0