ISE-529 Predictive Analytics

Module 1: Introduction

Module Overview/Agenda

- Introductions and Course Objectives
- Introduction to Predictive Analytics
- Course Approach and Grading
- Getting Started with Python and Jupyter Notebook

Introductions and Course Objectives

Course Objectives

- Develop an advanced level or proficiency with the primary classes of predictive modeling used by data scientists.
- Develop skills in using the Python programming environment and the primary packages and tools currently used by data scientists.
- Understand key concepts for measuring the performance of analytical models and key techniques for enhancing their performance.

Further Objectives

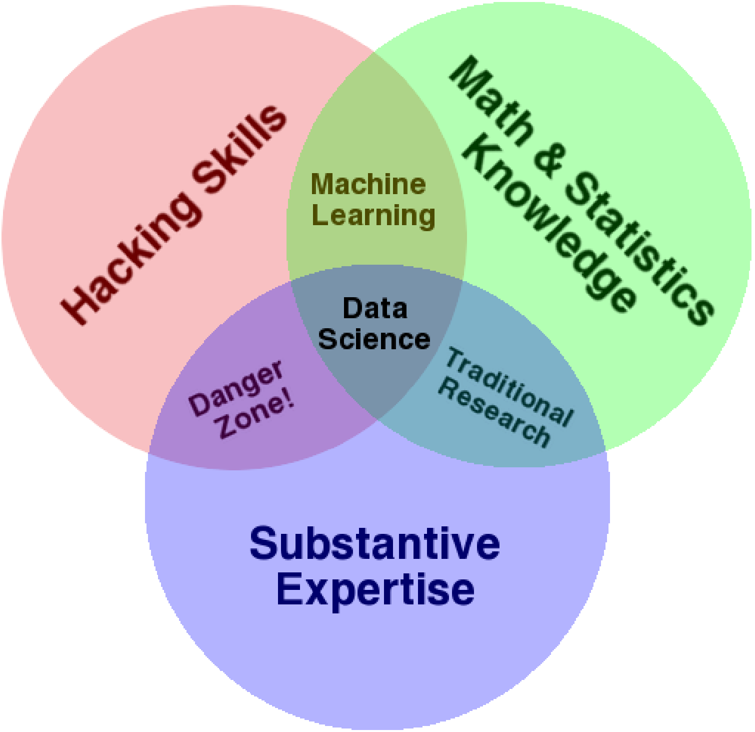
- In addition to the formal course objectives, another goal is to prepare students for success in their careers in the analytics field. This includes:
  - Preparing for the job search that many will be undertaking as the program draws towards completion
  - Learning about the tools that are commonly used in corporate settings
  - Getting experience in preparing and presenting reports in a manner that is transferrable to corporate settings
  - Developing a literacy in the classic papers and concepts in the field

Introduction to Predictive Analytics

What is Predictive Analytics?

- Predictive analytics is the process of using known results to create, process, and validate a model that can be used to forecast future outcomes.
- Generally, we are trying to develop models for the relationships between one or more "inputs" ("independent" or "predictor" or "explanatory" variables) to one "output" attribute ("dependent" or "response" variable) in the data

Drew Conway's Data Science Venn Diagram



Terminology

"It seems we have more terms than concepts here"

Data Science

Analytics

Business Analytics

Predictive Modeling

Data Mining

Big Data

Artificial Intelligence

Supervised Analytics

Deep Learning

Online Analytical Processing (OLAP)

Business Intelligence

Prescriptive Analytics

Machine Learning

Predictive Analytics

Unsupervised Analytics

Statistical Learning

Descriptive Analytics

Terminology

Data Science vs Analytics

Both are “umbrella terms” that are largely synonymous in general usage, but …

Analytics

- Focus on inference (getting insights from data)
- Emphasis on statistical techniques
- Primary tools: R, SQL, SAS, Tableau

Data Science

- Focus on prediction (predicting the future, or characteristics of previously unseen observations)
- Emphasis on computational techniques
- Primary tools: Python, Java

Terminology

Descriptive/Predictive/Prescriptive Analytics

Different objectives for the analytics work:

Descriptive Analytics

- Focused on understanding what happened or what is happening
- Uses both statistical and computational techniques
- Often involves visualization

Predictive Analytics

- Focused on predicting what will happen.
- Model assessment and validation are important topics
  - Model bias/variance tradeoff
  - Underfitting/overfitting tradeoff

Prescriptive Analytics

- Focused on recommending (or making) the business decisions based on data
- Primary techniques include optimization and simulation

Terminology

Supervised vs Unsupervised Analytics

One fundamental division of analytical techniques:

Supervised Learning

- The training data includes the response variable (the “answer”) that we are trying to model
- Predictive modeling falls into this category

Unsupervised Learning

- The training data does not include a response variable
- Unsupervised learning is focused on understanding patterns in the data
- Key unsupervised techniques include:
  - Clustering
  - Association Rule Mining/Collaborative Filtering
  - Outlier detection

Terminology

Statistical Learning vs Machine Learning

- Machine Learning arose as a subfield of artificial intelligence
- Statistical Learning arose as a subfield of statistics
- There is now significant overlap in the terms
  - Machine learning emphasizes large scale applications and prediction accuracy
  - Statistical learning emphasizes models, their interpretability, and precision and uncertainty

Terminology

Regression vs Classification

Predictive modeling divides into two basic types:

Regression

- Response variable (that we are trying to model and predict) is continuous
- Regression models are assessed and selected based on some metric of the average "error" – difference between the value predicted by the model and the actual value (usually on separate "training data")

Classification

- Response variable (that we are trying to model and predict) is a category (discrete)
  - Binary classification (two possible classes) is the most common form
- Classification models are assessed and selected based on some "misclassification rate" metric – percentage of time the value predicted by the model is wrong (usually on separate "training data")

Terminology

Regression vs Classification

Note: the term "regression" is overloaded (has two different meanings)

- Definition on previous slide (continuous response variable) is now standard
- "Regression models" also have a historical definition tied to the linear regression equation ($Y = \beta_0 + \beta_1 X$)
  - So, we have the situation where a "logistic regression" model is NOT a regression model – it is a classification model.

Terminology

Other Confusing Terms

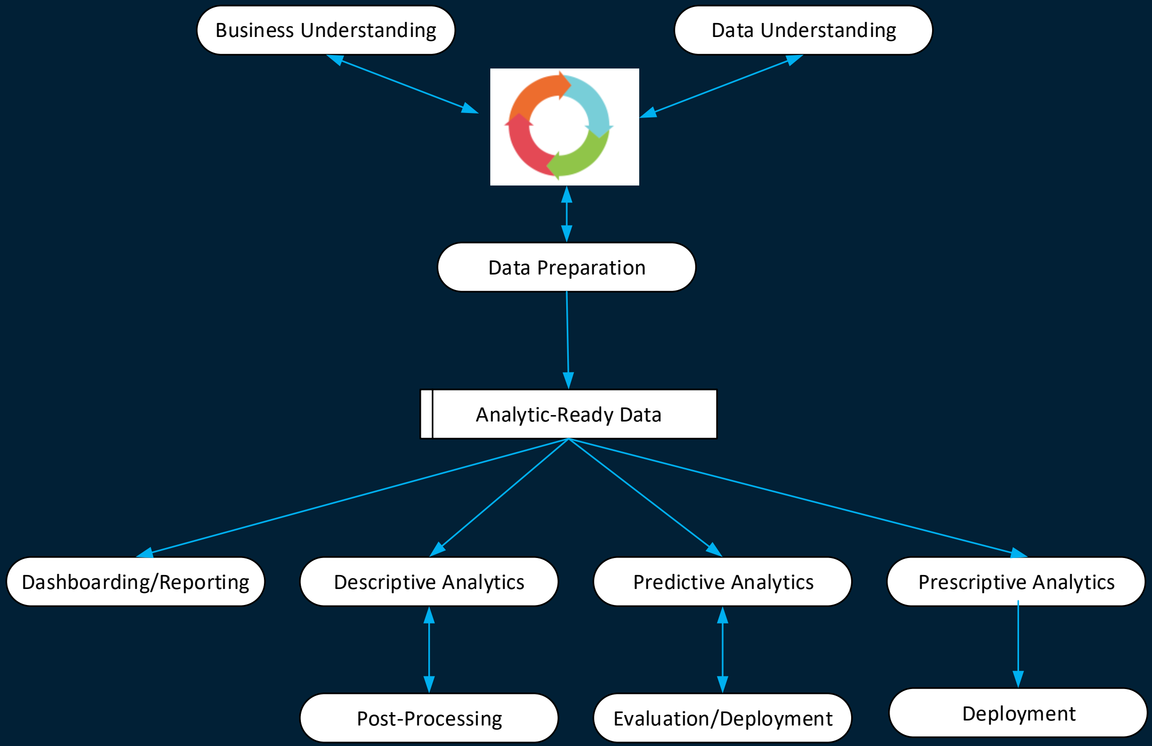
Artificial Intelligence – analytics which attempts to carry out tasks that are traditionally performed by humans

- Chatbots
- Healthcare diagnoses

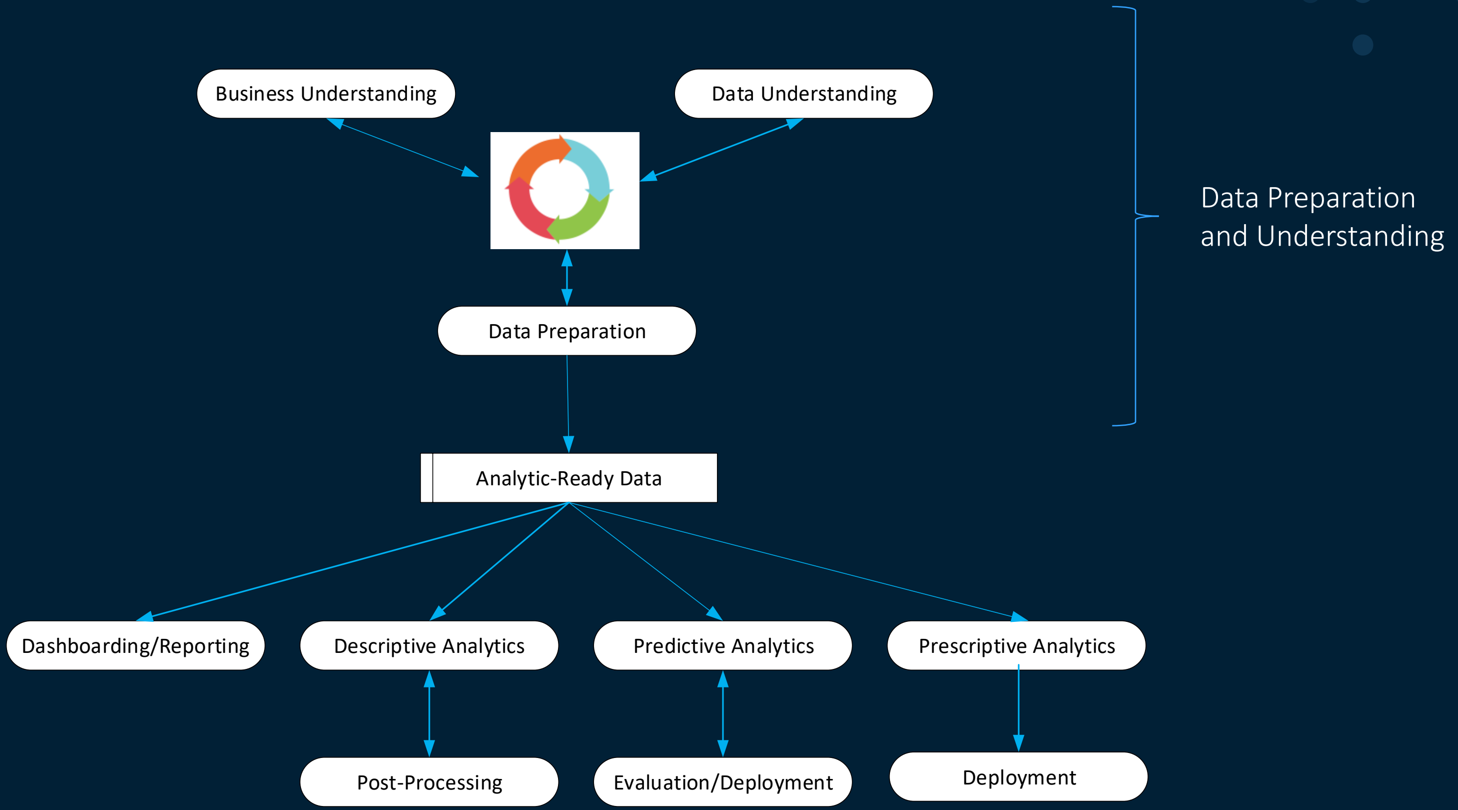
Neural Networks – A specific machine learning algorithm that attempts to replicate the structure of human cognitive processes

“Deep Learning” – the new name for neural networks

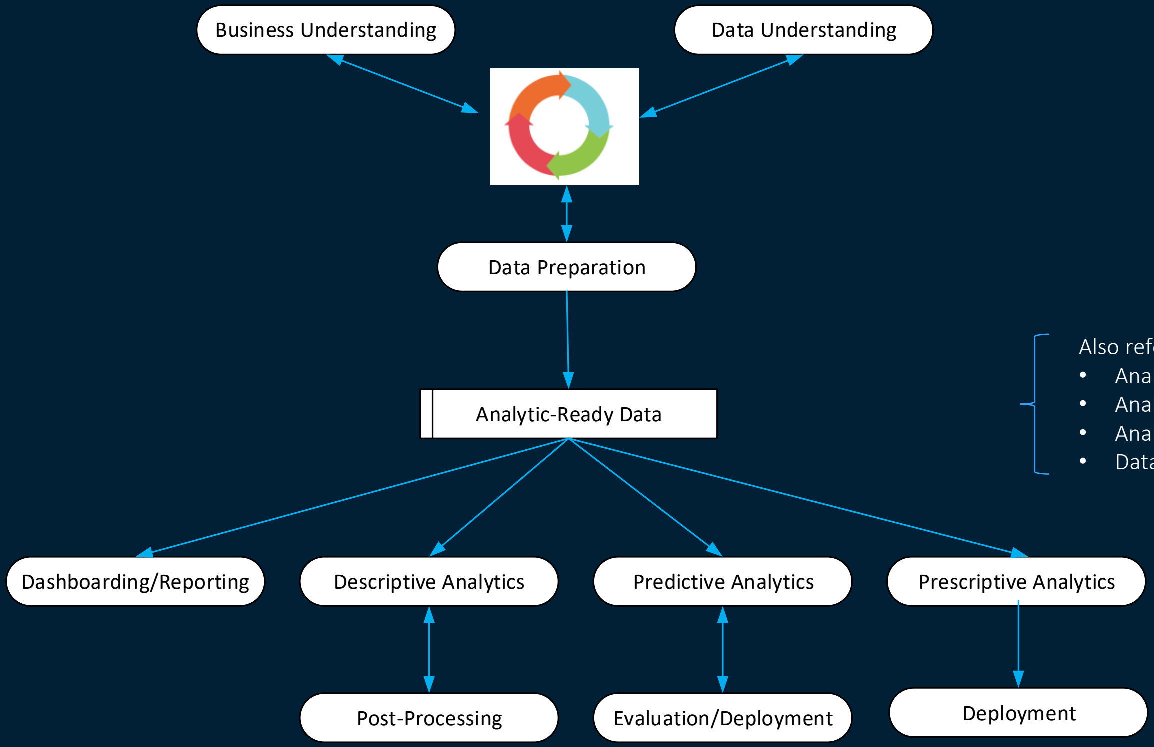
A Data Science Taxonomy



A Data Science Taxonomy



A Data Science Taxonomy

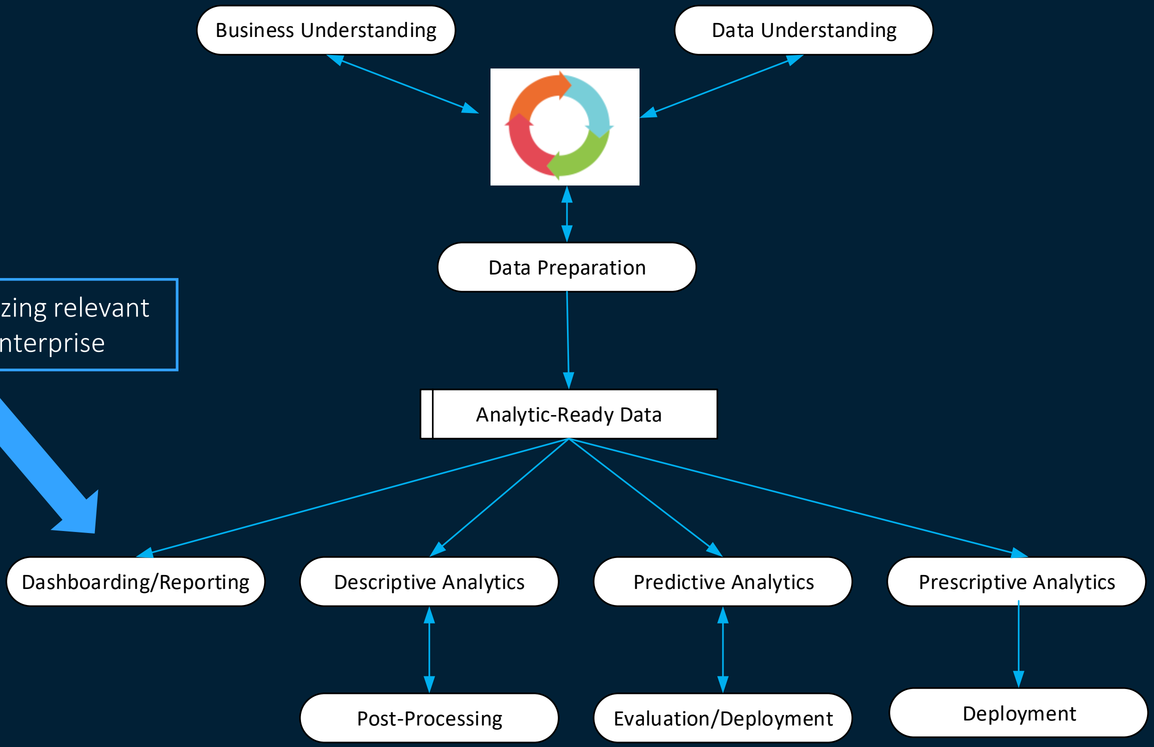


Also referred to as:

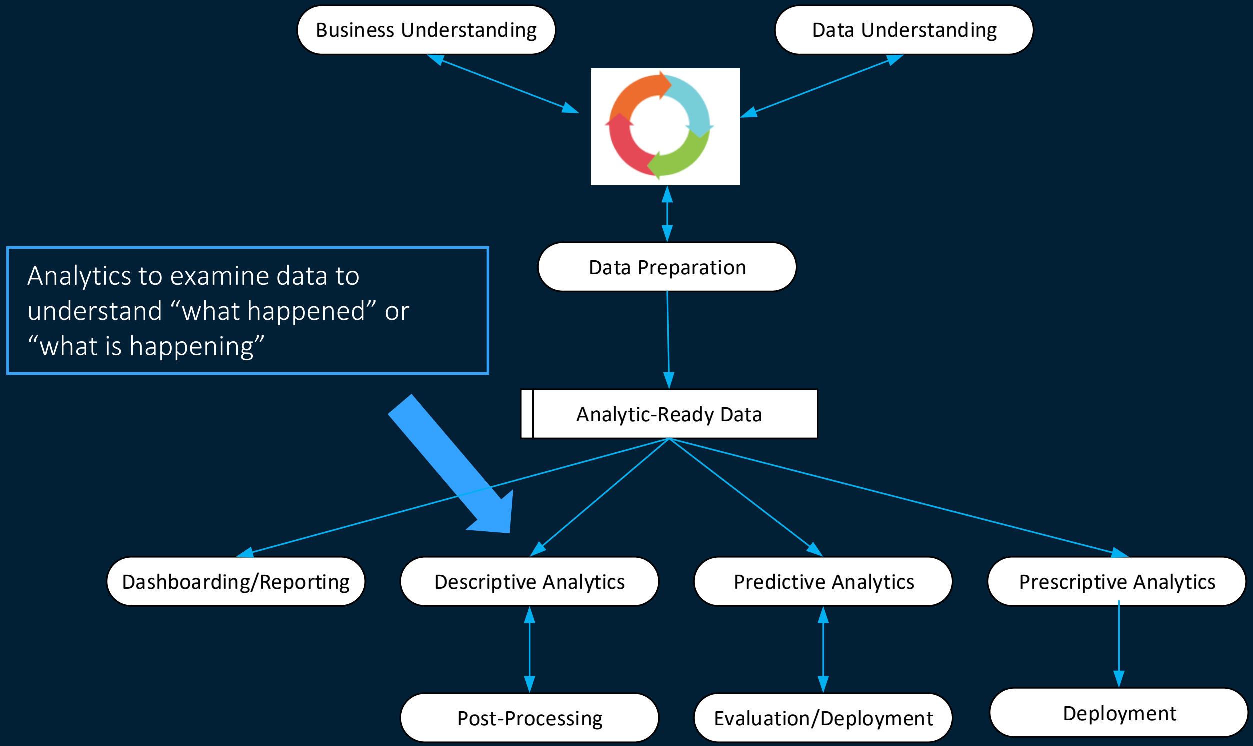
- Analytic Data Mart
- Analytic Base Tables
- Analysis Table
- Data Matrix

A Data Science Taxonomy

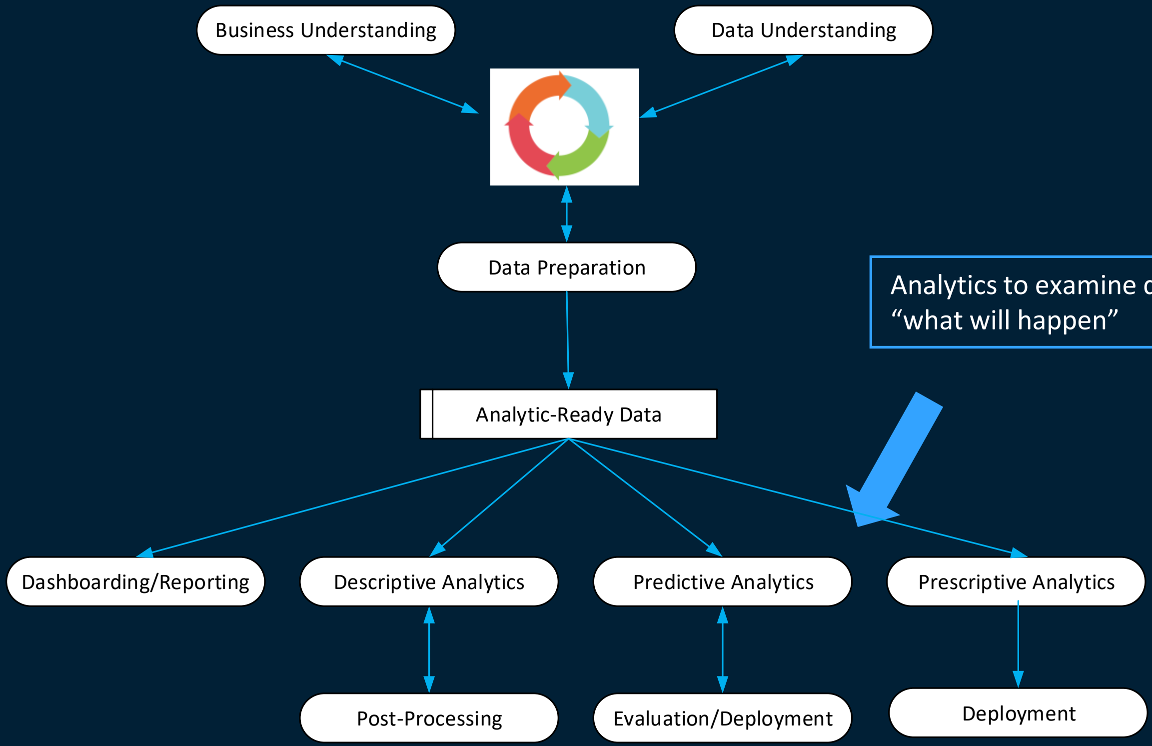
Visualizing and summarizing relevant information about the enterprise



A Data Science Taxonomy

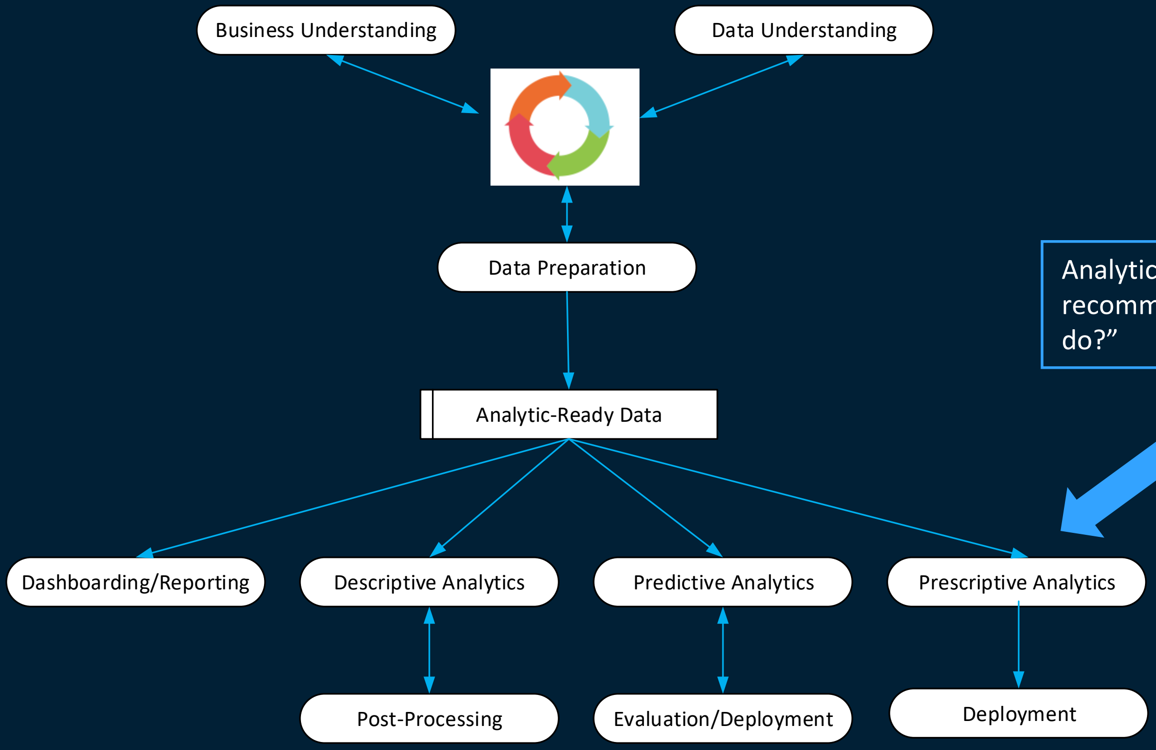


A Data Science Taxonomy



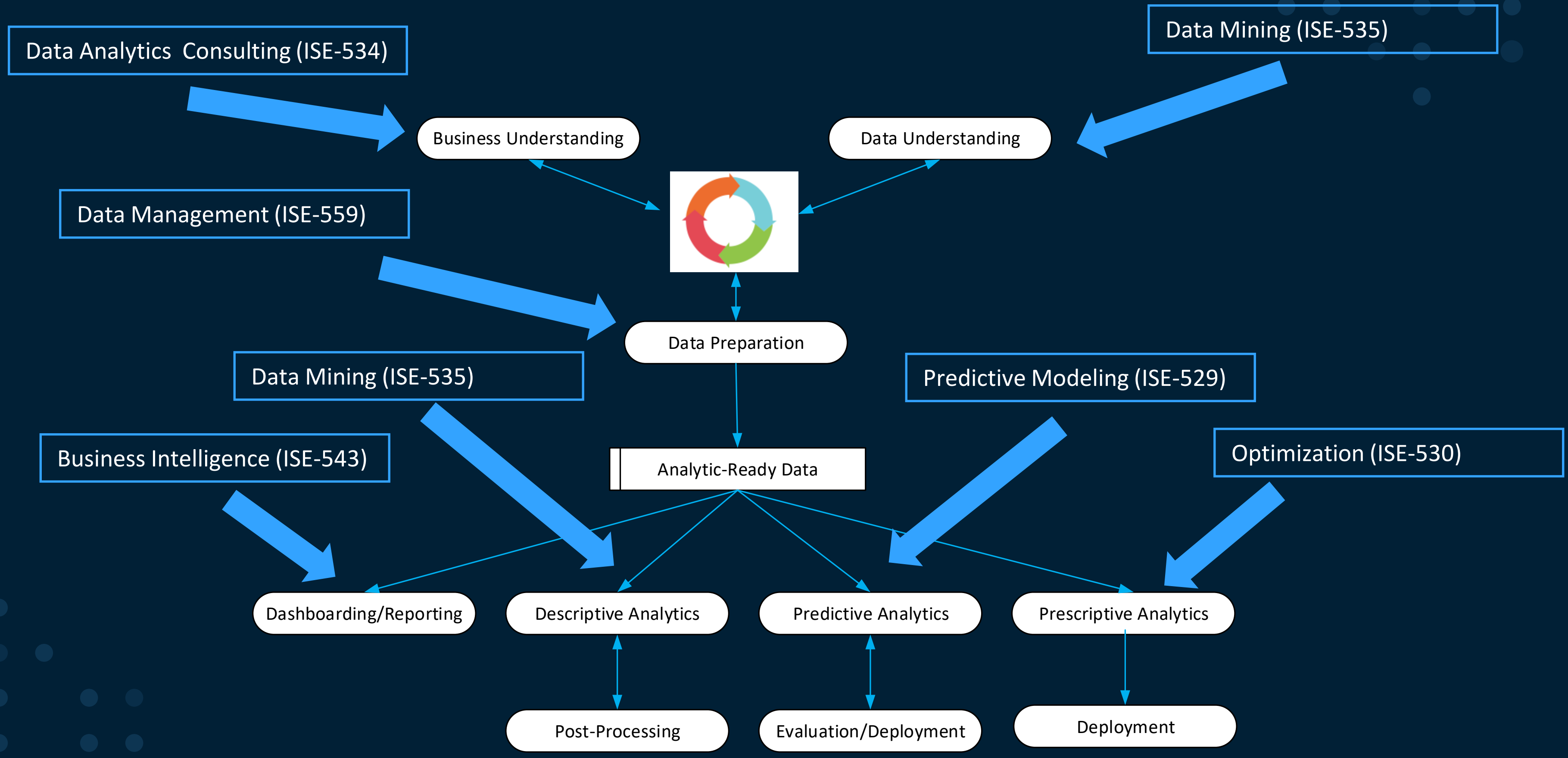
Analytics to examine data to predict "what will happen"

A Data Science Taxonomy



Analytics to examine data to make recommendations on “what should I do?”

A Data Science Taxonomy



Course Approach and Grading

Course Approach

Structure

- The course will be organized into eleven primary modules. Each module will be covered in 1-2 weeks and will include in-class exercises and one graded homework assignment.
  - For each module, I will identify textbook(s) chapters that I have used as primary resources. Students are encouraged to review these chapters prior to the corresponding lectures.

Course Approach

Modules

First half – "the linear model"

1. Introduction to Predictive Analytics and Python
2. Data Preparation and Modeling Introduction
3. Linear Regression Model Definition and Assessment
4. Linear Model Diagnostics and Validation
5. Linear Model Selection and Regularization

Second half – "extensions and ML models"

6. Classification Models
7. Generalized Linear Models and Poisson Regression
8. Moving Beyond Linearity
9. Tree-Based Models and Ensemble Models
10. Support Vector Machines
11. Introduction to Neural Networks

Course Logistics

Homework

- All materials will be uploaded to Blackboard. Assignments will be posted on Blackboard but will be submitted using GradeScope.
- Submission process is straightforward:
  - Only PDF files can be submitted
  - To prepare your solution for submission:
    - In Jupyter notebook, File -> Download as -> HTML
    - Open HTML in browser and “print to PDF”
  - After uploading your solutions PDF file to GradeScope, you will be prompted to map each rubric grading item to a page or pages in your file
  - A file with instructions is uploaded to Blackboard under this module

Course Logistics

Homework

- The homework due date is generally the day of the next class after it is assigned.
  - See GradeScope for the official due date
- I will set GradeScope to accept late submissions for two days after the due date
  - After the late due date, submissions will not be accepted
  - Students are allowed one "free" late submission. After that late submissions will be penalized 10 points (out of 100) unless approved by me *<u>in advance</u>*
  - No submissions will be accepted after the late submission due date
- The lowest homework grade will be dropped

Course Logistics

Communications

- We will use Piazza as our primary communications channel

  https://piazza.com/class/l4yk8f5xgy66v5?cid=4#

- Please post any questions you have there instead of sending me an email
  - If your message is only for me, send it as a private message on Piazza
- Students are strongly encouraged to answer each other's questions and to help clarify existing questions
  - This is one way to earn "class engagement" extra credit

Grading

- Grading will be based on the following components
  - Homework assignments that primarily consist of Python programming assignments (50%)
  - In-class mid-term (20%) and final exams (30%) on theory

A	95-100	B-	80-82	D+	67-69
A-	90-94	C+	77-79	D	63-66
B+	87-89	C	73-76	D-	60-62
B	83-86	C-	70-72	F	59 and below

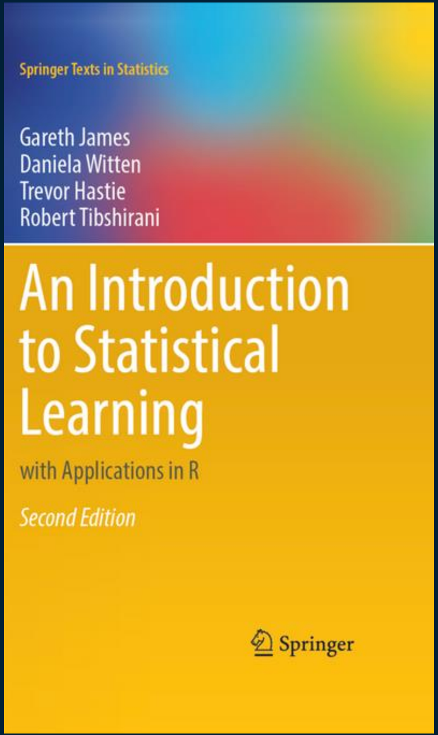
- In addition, up to 2 points may be added to the overall grade based on “class engagement”

Grading

- Class engagement extra credit will be awarded at the discretion of the instructor based on:
  - Active participation during the lectures (can be done in class or online)
  - Active online participation in discussions and answering questions on Piazza
- The mid-term and final exam will be during class time (see detailed class schedule in the syllabus)
  - The exams are open book and may be taken remotely, but you may not collaborate with other students
- As noted earlier, the lowest homework grade will be dropped

Texts

The “Core” Text



- We will cover Chapter 1-10 of this text
- The authors have made the book available for free on their website: https://www.statlearning.com/
- The book uses R for some exercises. We will be doing similar exercises but using Python.
- Additional texts are listed in the syllabus and will be referenced when I use materials from them

Office Hours

Instructor

- In person: Monday/Thursday 3:00PM – 4:00PM (OHE 310u)
  - If you would like to join by Zoom, please let me know in advance. I will do my best to connect with you and answer your questions, but I will give priority to students who are in the office
  - I am open to scheduling Zoom office hour times by appointment

TAs

- To be announced

Please check Piazza for any changes to these times!

Detailed Class Schedule

Class	Date	Topics/Daily Activities	Assignments	References
1	6/30	**Module 1: Introduction to Predictive Analytics and Python/Pandas** Introduction to Python, Jupyter Notebook *Tools: NumPy, Pandas*	Module 1 HW Assigned	Course Notes
2	7/7	**Module 2: Modeling Introduction.** Statistical learning, modeling types, model assessment and selection	Module 1 HW Due Module 2 HW Assigned	ISLR, Chapters 1-2
3	7/11	**Module 3: Linear Regression, Part 1** Model definition and model assessment Tools: scikit-learn, statsmodels	Module 2 HW Due Module 3 HW Assigned	ISLR, Chapter 3
4	7/14	**Module 4: Linear Regression, Part 2** Model diagnosis and validation Resampling methods and model variance	Module 3 HW Due Module 4 HW Assigned	ISLR 3.3.3 & 5.1
5	7/18	**Module 5: Linear Model Selection and Regularization** Subset selection, shrinkage methods, dimension reduction methods, high-dimensional data	Module 4 HW Due Module 5 HW Assigned	ISLR, Chapter 6 & 5.2
6	7/21	**Linear Models Review** **Mid-Term (90 Minutes)**	Module 5 HW due	
7	7/25	**Module 6: Classification** Logistic regression, linear discriminant analysis, and generalized linear models	Module 6 HW Assigned	ISLR, Chapter 4.1-4.5
8	7/28	**Module 7: Generalized Linear Models and Poisson Regression**	Module 6 HW Due Module 7 HW Assigned	ISLR, Chapter 4.6
9	8/1	**Module 8: Moving Beyond Linearity** Basis functions, splines, and generalized additive models	Module 7 HW Due Module 8 HW Assigned	ISLR, Chapter 7
10	8/4	**Module 9: Tree-Based Methods and Ensemble Models** Decision trees, forests, gradient boosting	Module 8 HW Due Module 9 HW Assigned	ISLR, Chapter 8
11	8/8	**Module 10: Support Vector Machines** **Module 11: Introduction to Neural Networks**	Module 9 HW Due Module 10/11 HW Assigned	ISLR Chapter 9 & 10
12	8/11	**Course Review** **Final Exam (120 minutes)**	Module 10/11 HW Due	

Notes:

- This schedule will almost certainly have to get adjusted as we move through the course. A current version of it will always be posted on Piazza
- The schedule in the syllabus will not be updated throughout the semester
- The assignment due dates here are not official. The official due dates will always be viewable on GradeScope

Introduction to Python

Core Tools of a Data Scientist

Mandatory:
- SQL (ISE-559)
- Python/Pandas (ISE-529)
- R/Tidyverse (ISE-535)
- Data modeling (ISE-559)
- Excel

BI/Dashboarding
- Tableau
- PowerBI
- SAS Visual Analytics (ISE-543)

Other analytical tools:
- SAS
- Matlab

Integrated data science platforms
- SAS Viya (ISE-543)
- IBM Watson Studio
- Databricks
- Tibco
- Dataiku

Introduction to Python

Outline

- Introduction to Python and Jupyter Notebook
- Python basics
  - Control flow
  - Data structures and sequences
  - Functions
  - Libraries
- NumPy
- Pandas
- Reading data from files
- Plotting with Matplolib and Seaborn

Python Tutorials

Optional Learning Material

First, sign up for your free account:

- Datacamp signup link

Once you have done that, I would recommend that you consider the following tutorials (depending on your skill levels):

- Introduction to Python
- Introduction to Data Science in Python - Parts 1 and 2
- Data Manipulation with Pandas

Introduction to Python and Jupyter Notebook

Outline

- Overview of Python, Jupyter Notebook, and the core data science libraries
- Language Basics
  - Python
  - NumPy
  - Pandas
  - MatplotLib
- Loading Data

Python

- Open-source programming language developed by Guido van Rossum in the early 1990s
- Named after Monty Python
- Interpreted language (as opposed to compiled)

R or Python?

- Both languages are used extensively by data scientists
- Both include a large number of libraries for data analytics, data visualization, machine learning, web scraping, text analytics, and deep learning
- R is focused heavily on statistical learning/data analysis.
- Python is a more general-purpose language

We will focus on using Python as data science tool

The Python Ecosystem

A collection of

- Python language (currently version 3.9)
- Integrated development environments (IDEs - we will use Jupyter Notebook)
- Libraries

The Anaconda distribution bundle contains everything needed for typical data science uses.

Why Jupyter Notebook?

Data Science IDEs vs Developer IDEs

Data Science IDE

- Data-centric
- Interactivity, visualizations, variable explorer
- Less code complexity, scripts
- Integration with data sources
- Models and narratives/storytelling

Developer IDE

- Code-centric
- Classes, debugging, profiling
- More complex code, programs
- Integration with git, build tools, compilers
- Tools and libraries/functionality

Major IDEs

Data Science IDEs

- Jupyter
- Spyder
- RStudio

Developer IDEs

- PyCharm
- Pydev
- Wing IDE
- Sublime text
- Visual studio

Installing Python and Jupyter Notebook Using Anaconda

- Go to www.anaconda.com
  - “Get Started”
  - “Download Anaconda Installers”
  - Select the appropriate operating system for your use

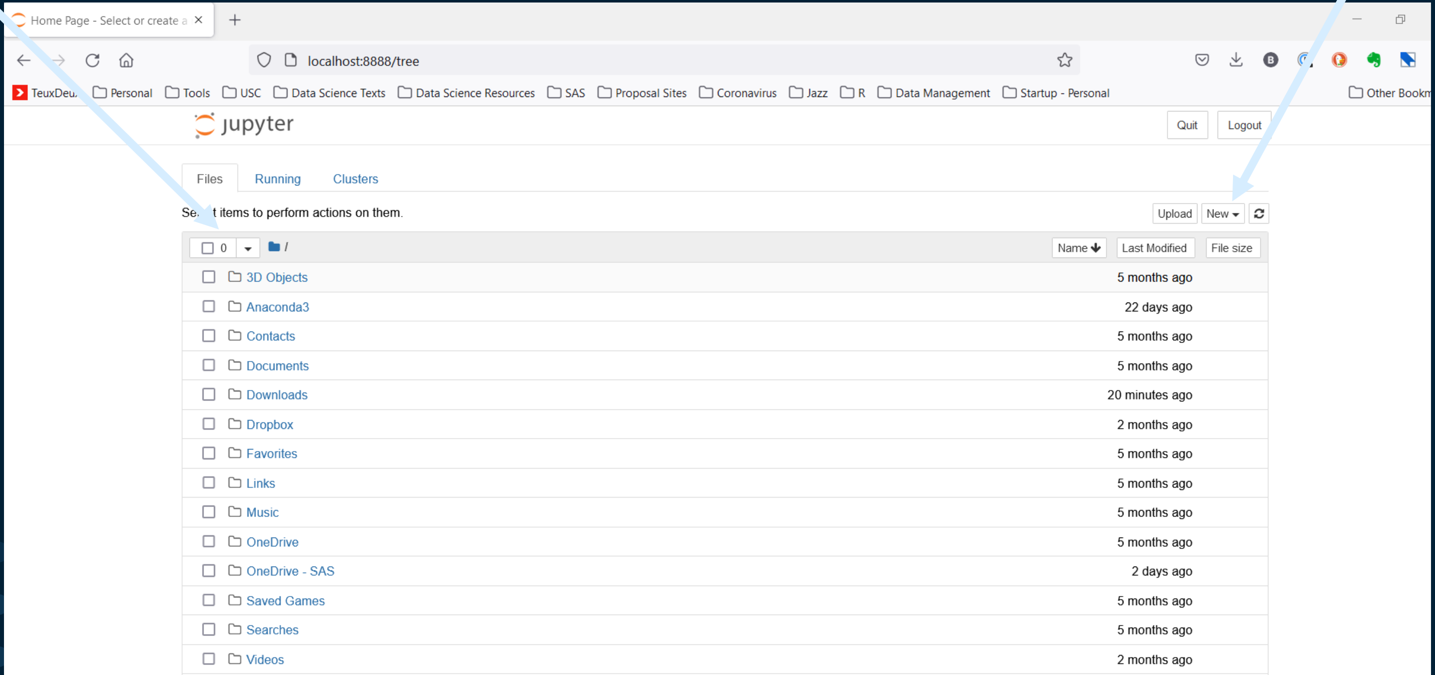
- Executing the Jupyter Notebook program opens the IDE in a browser

Getting Started With Jupyter Notebook

Dashboard

Navigate to the folder you want as your working area

Create a new notebook

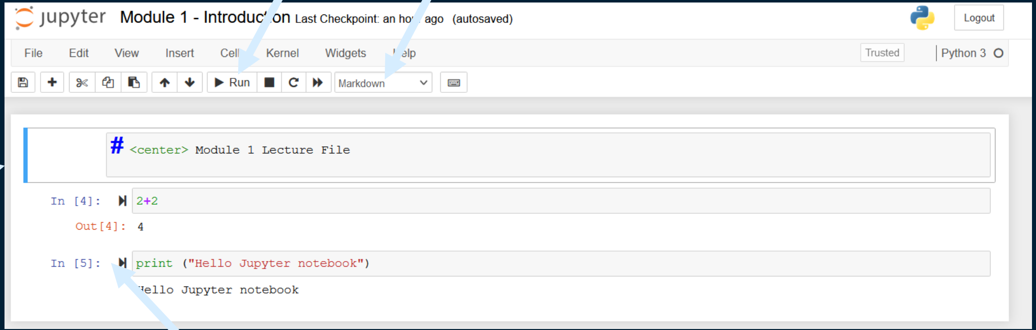


Getting Started With Jupyter Notebook

A notebook consists of a collection of cells which are either code cells or Markdown cells

Run all the cells in the notebook

Specify the type of cell



Keyboard input modes:

- Edit mode (green ribbon)
- Command mode (blue ribbon)

Execute the code in the cell (also can be done by hitting control-Enter)

Jupyter Notebook Markdown Guides:

https://medium.com/analytics-vidhya/the-ultimate-markdown-guide-for-jupyter-notebook-d5e5abf728fd

https://www.ibm.com/docs/en/watson-studio-local/1.2.3?topic=notebooks-markdown-jupyter-cheatsheet

Python Basics

Python is an "interpreted" language, meaning that you can type a Python command (or string of commands) and get an immediate result:

```
In [1]:  2+2
Out[1]:  4

In [2]:  print ("Hello Jupyter notebook")
         Hello Jupyter notebook
```

Basic Datatypes

- Integers (default for numbers)
- Floats
- Strings
  - Specified with “” or ‘’: “abc” = ‘abc’
  - Unmatched can occur within string: “matt’s”

Python Basics

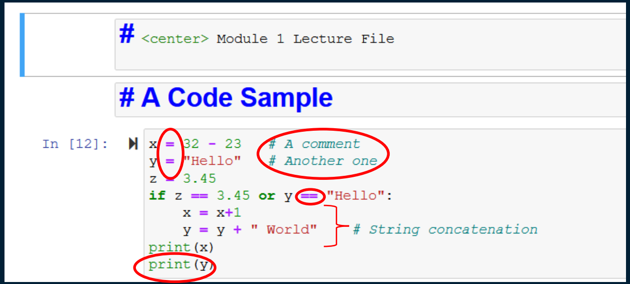
- Whitespace is meaningful in Python!
  - Especially indentations and placement of newlines
- Use a newline to end a line of code
  - Use \ when you want to continue a Python commandment onto a new line
- No braces {} to mark blocks of code in Python
  - Use consistent indentation instead
  - First line with more indentation starts a nested block
  - First line with less indentation is outside of the block
- Often a colon appears at the start of a new block (e.g., for function definitions)

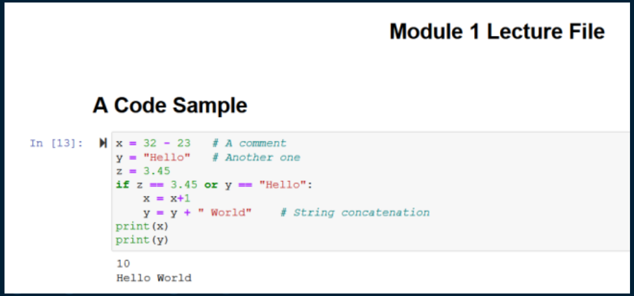
Comments

- Start comments with # - the rest of the line is ignored
- Can optionally include a “documentation string” as the first line of any new function you define (recommended):

```
def my_function (x,y):
    “””Docuentation string goes here”””
    # Code goes here
```

Python Basics





- Comment text preceeded by a # is ignored
- The first assignment to a variable creates it
- Assignment uses =, comparison uses ==
- Indentation and white space matter to the meaning of the code!
- For numbers + - * / % are as expected
- Logical operators are words (and, or, not) and not symbols
- Simple printing can be done with print()

Built-In Python Scalar Data Types

Type	Description
None	The Python “null” value (only one instance of the None object exists)
str	String type; holds Unicode (UTF-8 encoded) strings
bytes	Raw ASCII bytes (or Unicode encoded as bytes)
float	Double-precision (64-bit) floating-point number (note there is no separate double type)
bool	A True or False value
int	Arbitrary precision signed integer

- Note: using Python modules can add additional data types. For example, the `datetime` module provides `datetime`, `date`, and `time` types

Assignment

- “Binding a variable” in Python means setting a “name” to hold a “reference” to some “object”
  - Assignment creates references, not copies!
- A name is created the first time it appears on the left side of an assignment expression:

```
x = 3
```

Accessing Non-Existent Names

- If you try to access a name before it's been created, you'll get an error:

```
In [19]: new_variable

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-19-a22a888b0e87> in <module>
----> 1 new_variable

NameError: name 'new_variable' is not defined
```

Multiple Assignment

You can also assign multiple names at the same time:

```
In [21]: x,y = 2,3
         x

Out[21]: 2

In [22]: y

Out[22]: 3
```

Naming Rules

- Names are case sensitive and cannot begin with a number
- They can contain letters, numbers, and underscores

`bob Bob _bob _2_bob_ bob_2 BoB` (all different)

- Python has “reserved words” that cannot be used:

and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while

Python Reference Semantics

- Assignment (x=y) makes x reference the object y references
- Assignment does not make a copy of the object y references!

```
In [23]: a = [1,2,3]
         b = a
         a.append(4)
         print(a)
         print(b)

[1, 2, 3, 4]
[1, 2, 3, 4]
```

Python Basics

Objects and Functions

- Everything in Python is an object
  - Each object has an associated type, data, attributes, and methods
  - Attributes are characteristics of the object
  - Methods are basically pre-defined functions that are called by appending to a variable a “.” followed by the method name:
    - Function call: result = some_function(x,y,x) Assigns the result to the variable result
    - Object method: obj.some_method(x,y,z) Performs an action using internal data in the object

Python Basics

Numeric Functions

Python Numeric Functions

```
In [81]: x = 123
         abs(x)

Out[81]: 123

In [82]: bin(x)

Out[82]: '0b1111011'

In [83]: complex(x)

Out[83]: (123+0j)

In [84]: float(x)

Out[84]: 123.0
```

Python Basics

Numeric Methods

Python Numeric Methods

```
In [92]: x=123.4
         x.is_integer()
Out[92]: False

In [93]: x.as_integer_ratio()
Out[93]: (4341751515761869, 35184372088832)

In [95]: x.hex()
Out[95]: '0x1.ed9999999999ap+6'
```

Python Basics

String Methods

Python String Methods

```
In [46]: str1 = "Hello World"
         str1.lower()

Out[46]: 'hello world'

In [47]: str1.upper()

Out[47]: 'HELLO WORLD'

In [48]: str1

Out[48]: 'Hello World'

In [42]: str1.count("l")

Out[42]: 3

In [44]: str1.endswith("d")

Out[44]: True
```

Python string methods documentation:
https://docs.python.org/3/library/stdtypes.html#string-methods

Python Basics

String Functions

Python functions that operate on strings

In [62]:
```
str1 = "Hello World"
len(str1)
```
Out[62]:
```
11
```

In [69]:
```
type(str1)
```
Out[69]:
```
str
```

Methods vs Functions

What's the Difference? Why Both??

- Generally, methods and functions are very similar and you will often see the terms used interchangeably (but don't you do that!)
- Methods are specific to object types (more on that later) while functions can apply to multiple object types
- Another confusion is whether or not the calling object is modified
  - Functions never change the calling object
  - Methods sometimes do and sometimes don't (caution is needed)

Binary Math Operations

Most binary math operations and comparisons operate as expected:

Binary Math Operations

```
In [28]: 5-7
Out[28]: -2

In [29]: 12 + 21.5
Out[29]: 33.5

In [30]: 5 <= 2
Out[30]: False
```

Binary Operators

Operation	Description
a + b	Add a and b
a - b	Subtract b from a
a * b	Multiply a by b
a / b	Divide a by b
a // b	Floor-divide a by b, dropping any fractional remainder
a ** b	Raise a to the b power
a & b	True if both a and b are True; for integers, take the bitwise AND
a | b	True if either a or b is True; for integers, take the bitwise OR
a ^ b	For booleans, True if a or b is True, but not both; for integers, take the bitwise EXCLUSIVE-OR

Binary Boolean Operators

Operation	Description
a == b	True if a equals b
a != b	True if a is not equal to b
a <= b, a < b	True if a is less than (less than or equal) to b
a > b, a >= b	True if a is greater than (greater than or equal) to b
a is b	True if a and b reference the same Python object
a is not b	True if a and b reference different Python objects

Python Language Basics

Control Flow

Language Basics

Flow Control

Flow Control

If-Then-Else

In [111]:

```
value = 99
if value == 99:
    print('That is fast')
elif value > 200:
    print('That is too fast')
else:
    print('That is safe')
```

```
That is fast
```

Language Basics

Flow Control

For-Loop

In [112]:

```
# For-Loop
for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

Language Basics

Flow Control

While-Loop

In [113]:

```
i = 0
while i < 10:
    print(i)
    i += 1
```

```
0
1
2
3
4
5
6
7
8
9
```

Python Language Basics

Data Structures and Sequences

Basic Python

Data Structures

Ordered Data Structures (Sequences)

- List: Mutable (changeable) ordered sequence of items of mixed types
- Tuple: An immutable (unchangeable) ordered sequence of items of mixed types

Unordered Data Structures

- Set: Unordered and unindexed collection of unique elements
- Dictionary: Ordered collection of key-value pairs
  - Also referred to as a *hash map* or an *associative array*

Base Python Data Structures: Lists

Lists: The "Workhorse" of Vanilla Python

Lists

Lists use the square bracket notation and can be modified.

Creating Lists

In [8]:
```
bikes = ["trek", "redline", "giant"]
bikes
```
Out[8]:
```
['trek', 'redline', 'giant']
```

In [13]:
```
first_10 = list(range(10))
first_10
```
Out[13]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [14]:
```
mixed_list = ["trek", 500, "redline", 600, "giant", 750]
mixed_list
```
Out[14]:
```
['trek', 500, 'redline', 600, 'giant', 750]
```

Base Python Data Structures: Lists

Accessing List Elements

Accessing list elements

Get the first item in a list

In [81]:
```
bikes[0]
```

Out[81]: 'trek'

In [82]:
```
# Get the first item in a list
print(bikes[0])

# Get the last item in a list
print(first_10[-1])
```

```
trek
9
```

In [83]:
```
b = [1,2,3]
print(b)
print(b[0])
print("Zeroth value: " + str(b[0]))
print("list Length: " + str(len(b)))
for value in b:
    print(value)
```

```
[1, 2, 3]
1
Zeroth value: 1
list Length: 3
1
2
3
```

Base Python Data Structures: Lists

“Slicing” a List

```
x = [1,3,5,8,2,4]
```

```
x
```

```
[1, 3, 5, 8, 2, 4]
```

show first 4

```
x[:4]
```

```
[1, 3, 5, 8]
```

show all beyond the first 4

```
x[4:]
```

```
[2, 4]
```

show values with index 1 and 2

```
x[1:3]
```

```
[3, 5]
```

show last value

```
x[-1]
```

```
4
```

show all excluding last value

```
x[:-1]
```

```
[1, 3, 5, 8, 2]
```

Base Python Data Structures: Lists

Functions for Lists

`append(x)` adds `x` to the end of the list

`count(x)` counts how many times `x` appears in the list

`extend(L)` adds the elements in list `L` to the end of the original list

`index(x)` returns the index of the first element of the list to match `x`

`insert(i, x)` inserts element `x` at location `i` in the list, moving everything else along

`pop(i)` removes the item at index `i`

`remove(x)` deletes the first element that matches `x`

`reverse()` reverses the order of the list

`sort()` we've already seen

Base Python Data Structures: Lists

Methods That Operate on Lists

In [68]:
```python
x = [1,3,5,8,2,4]
x
```
Out[68]: `[1, 3, 5, 8, 2, 4]`

In [69]:
```python
x.append(9)   # Add 9 to the end of the list
x
```
Out[69]: `[1, 3, 5, 8, 2, 4, 9]`

In [70]:
```python
L = [0,5,8]
x.extend(L)
x
```
Out[70]: `[1, 3, 5, 8, 2, 4, 9, 0, 5, 8]`

In [71]:
```python
x.count(5)   # Number of times 5 appears in the list
```
Out[71]: `2`

In [42]:
```python
x.index(8)   # Position where 8 first occurs
```
Out[42]: `3`

Base Python Data Structures: Lists

Methods That Operate on Lists

In [72]:
```python
x.insert(3,6)   # Insert 6 in position 3
x
```
Out[72]: `[1, 3, 5, 6, 8, 2, 4, 9, 0, 5, 8]`

In [73]:
```python
x.pop(3)   # Deletes item from position 3
x
```
Out[73]: `[1, 3, 5, 8, 2, 4, 9, 0, 5, 8]`

In [74]:
```python
x.remove(8)   # Remove the first 8 in the list
x
```
Out[74]: `[1, 3, 5, 2, 4, 9, 0, 5, 8]`

In [75]:
```python
x.reverse()   #  Reverse the list
x
```
Out[75]: `[8, 5, 0, 9, 4, 2, 5, 3, 1]`

In [76]:
```python
y = x.copy()   #  Make a copy of the list x
y
```
Out[76]: `[8, 5, 0, 9, 4, 2, 5, 3, 1]`

In [77]:
```python
x.sort()   # Sort the list
x
```
Out[77]: `[0, 1, 2, 3, 4, 5, 5, 8, 9]`

Base Python Data Structures: Tuples

Tuples

- Tuples are similar to lists, but the items in a tuple can't be modified
- Uses () notation instead of [] used by lists

Tuples

Tuples are read-only collections of items defined with the parenthesis notation.

In [84]:

```python
a = (1,2,3)
print(a)
print (a[1])
a[1] = 6
```

```
(1, 2, 3)
2

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-84-aa194a3b7f62> in <module>
      2 print(a)
      3 print (a[1])
----> 4 a[1] = 6

TypeError: 'tuple' object does not support item assignment
```

Base Python Data Structures: Tuples

Methods That Operate on Tuples

Python has two built-in methods that operate on tuples

In [102]:
```
a = (1,2,3,2,1)
a.count(2)    # Returns the number of times a specified value occurs in a tuple
```
Out[102]: 2

In [104]:
```
a.index(3)  # Searches the tuple for a specified value and returns the position of where it was found
```
Out[104]: 2

Why Does Python Have Tuples?

- Efficiency. Tuples are much quicker for Python to process than lists
- Protection. Sometimes you want to make sure that a tuple never gets modified, particularly when each element has *semantic value*

time.localtime()

(2008, 2, 5, 11, 55, 34, 1, 36, 0)

Each element of the tuple has a specific meaning (year, month, day, etc.) and you wouldn’t want any individual item deleted

range(10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

We may care about the order, but individual values are functionally equivalent

Sets

- Unordered, immutable sequences with no duplicate values
- Highly efficient data structure for search, add, and delete opeations
  - Implemented with a hash table, not a linked list

Base Python Data Structures: Sets

Sets

Sets

Sets are unordered, immutable sequences with no duplicate values

A set can be created in two ways: via the set function or via a set literal with curly braces:

In [107]:
```
Set1 = set([2,2,2,1,3,3])
Set1
```

Out[107]:
```
{1, 2, 3}
```

In [111]:
```
Set2 = {2,2,2,1,3,3}
Set2
```

Out[111]:
```
{1, 2, 3}
```

Base Python Data Structures: Sets

Sets

Sets support mathematical set operations:

In [116]:
```
a = {1,2,3,4,5}
b = {3,4,5,6,7,8}
print(a.union(b))
print(a|b)
print(a.intersection(b))
print(a&b)
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
{1, 2, 3, 4, 5, 6, 7, 8}
{3, 4, 5}
{3, 4, 5}
```

Base Python Data Structures: Sets

Commonly Used Set Methods

Function	Alternate Syntax	Description
a.add(x)	N/A	Add element x to the set a
a.clear()	N/A	Reset the set a to an empty state, discarding all of its elements
a.remove(x)	N/A	Remove element x from the set a
a.pop()	N/A	Remove an arbitrary element from the set a, raising KeyError if the set is empty
a.union(b)	a | b	All of the unique elements in a and b
a.update(b)	a |= b	Set the contents of a to be the union of the elements in a and b
a.intersection(b)	a & b	All of the elements in *both* a and b
a.intersection_update(b)	a &= b	Set the contents of a to be the intersection of the elements in a and b

Dictionaries

- A mapping between a set of indices (keys) and a set of values
  - Each item in a dictionary is a key-value pair
- Keys can be any Python data type, but because they are used for indexing, they should be immutable
- Values can be any Python data type
  - Values can be mutable or immutable

Base Python Data Structures: Dictionaries

Dictionaries

Dictionaries

Dictionaries are mappings of names to values, like key-value pairs that are defined using the curly bracket and colon notations.

In [85]:

```
mydict = {'a': 1, 'b': 2, 'c': 3}
print("A value: " + str(mydict['a']))
mydict['a'] = 11
print("A value: " + str(mydict['a']))
print("Keys: " + str(mydict.keys()))
print("Values: " + str(mydict.values()))
for key in mydict.keys():
    print(mydict[key])
```

```
A value: 1
A value: 11
Keys: dict_keys(['a', 'b', 'c'])
Values: dict_values([11, 2, 3])
11
2
3
```

Comprehensions

- Comprehensions provide a short and concise way to construct new sequences (such as lists, sets, dictionaries, etc.) using sequences that have already been defined

Comprehensions

List Comprehnsions

Basic format:

```
output_list = [output_exp for var in input_list if (var satisfies this condition)]
```

Equivalent for-loop:

```
result = []
for val in collection:
    if condition:
        result.append(expr)
```

Comprehensions

List Comprehension Example

Comprenehsions

List comprehensions

In [120]:
```
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
[x.upper() for x in strings if len(x) > 2]
```

Out[120]:
```
['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Functions

Language Basics

Functions

Function definition begins with def

Function name and its arguments

Functions

Be sure to have an empty new line after the indented code in a function.

In [114]:

```
# Sum function
def mysum(x, y):
    return x + y

# Test sum function
result = mysum(1, 3)
print(result)
```

```
4
```

Colon

“return” indicates the value to be returned from the funtion

First line with less indentation is considered to be outside of the function devinition

Functions Without Returns

- All functions in Python have a return value
  - Even if no *return* line inside the code
- Functions without a *return* return the special value *None*
  - None is a special constant
  - None is also logically equivalent to False

Functions Are Objects

- Functions can be used like any other data
  - Assigned to variables
  - Parts of tuples, lists, etc
  - Arguments to other functions
  - Return values of functions

Default Values for Arguments

- You can provide default values for a function's arguments
- These arguments are optional when the function is called

```
In [122]:
def mysum(b, c=3, d = 'hello'):
    return b+c

print(mysum(5,3,"hello"))
print(mysum(5,3))
print(mysum(5))
```

```
8
8
8
```

Keyword Arguments

- Functions can be called with arguments out of order:

```
In [124]:
def mysum(a,b,c):
    return a-b

print(mysum(2,1,43))
print(mysum(c=43, b=1, a=2))
print(mysum(2, c=43, b=1))
```

```
1
1
1
```

Iterables and the Map Function

- An *iterable* is a Python object that can be used as a sequence (list, tuple, etc.)
- map() function returns a map object (which is an iterator) of the results after applying the given function to each item of a given iterable.
- Syntax: `map(fun, iter)`
  - fun: Function to which map passes each element of a given iterable
  - iter: Iterable which is to be mapped

Iterables and the Map Function

Iterables and the map function

In [128]:
```
def double_value(n):
    return 2*n

numbers = (1,2,3,4)  # tuple to be used as an interable
result= map(double_value, numbers)
list(result)  #  Map returns another iterable.  Converting the iterable to a list.
```

Out[128]: [2, 4, 6, 8]

Lambda Functions

- Python Lambda Functions are "anonymous" functions
  - Syntax: `lambda arguments: expression`
- Often handy to use a lambda function in a map function:

```
In [131]:  1  f = lambda x:2*x
           2  f(2.5)

Out[131]: 5.0

In [134]:  1  list(map(lambda x:2*x, range(10)))

Out[134]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Python Libraries

Python Libraries

- Python is implemented and distributed via libraries
- The Python Standard Library represents “base Python” and, together with the Python Language Reference, fully describes the basic Python language

Python Standard Library

Contains an extensive collection of components, for example:

- math: Mathematical functions
- cmath: Mathematical functions for complex numbers
- decimal: Decimal fixed point and floating point arithmetic
- fractions: Rational numbers
- random: Generate pseudo-random numbers
- statistics: Mathematical statistics functions

Using Standard Library Functions

Python Libraries

In [5]:
```
exp(1)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-a9c50c82dfa5> in <module>
----> 1 exp(1)

NameError: name 'exp' is not defined
```

In [6]:
```
math.exp(1)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-6-d6be2f03fd11> in <module>
----> 1 math.exp(1)

NameError: name 'math' is not defined
```

In [8]:
```
import math
math.exp(1)
```

Out[8]: 2.718281828459045

We want to use the exp() function which is part of the Python math library.

- We have to tell Ipython what library it belongs to
- However, even though it's part of a standard library, we must first import it into our notebook file

Third-Party Libraries

- In addition to standard libraries, the Python ecosystem has a large number of *third-party libraries*
- Several of these third-party libraries have become standard to use in data science applications:
  - NumPy (numerical Python): Adds high performance numeric arrays
  - Pandas (Python Data Analysis): Adds heterogeneous array types
  - Matplotlib: Basic plotting functions (from Matlab)
  - Seaborn: Ggplot-like plotting functions
  - Statsmodels: Statistical modeling (similar syntax to R)
  - Scikit-Learn: Implementation of many standard machine learning algorithms
  - TensorFlow: More complex library for distributed numerical computation

Revisiting Python Objects

- Python is an object-oriented language
- In Python, everything that can be named (variable, function, etc.) is an *object*
  - Every object has *attributes*
  - *Methods* can be applied to an object via the dot syntax
- Objects have *types* (also referred to as *classes*)
- Libraries often define new *types/classes* (which have associated attributes and methods)

Python Libraries

- A library is a collection of high-level functions
  – Allow users to develop applications without having to code low-level details
- In Python:
  – A *library* is a collection of one or more *modules (Python files)*
  – *Modules* are made of one or more *classes*
  – *Classes* include *methods* and *attributes*

Importing Third-Party Libraries

- There are several options for importing a library or part of a library:

$\texttt{import } \underbrace{\texttt{numpy}}_{\text{library name}} \texttt{ as } \underbrace{\texttt{np}}_{\text{alias}}$

Import the entire library

$\texttt{import } \underbrace{\texttt{matplotlib}}_{\text{library name}}\texttt{.}\underbrace{\texttt{pyplot}}_{\text{module}} \texttt{ as } \underbrace{\texttt{plt}}_{\text{alias}}$

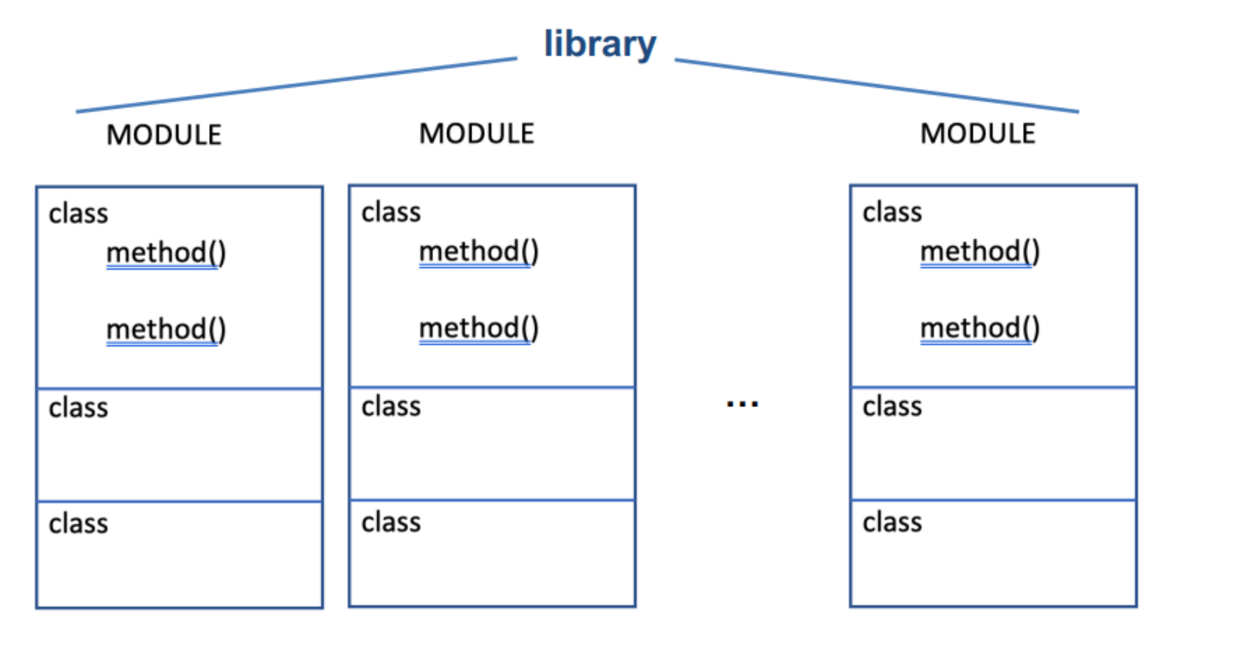
Import a specific module from a library

$\texttt{from } \underbrace{\texttt{sklearn}}_{\text{library name}}\texttt{.}\underbrace{\texttt{model\_selection}}_{\text{module}} \texttt{ import } \underbrace{\texttt{kfold}}_{\text{class}}$

Import a specific class (with associated methods/attributes) from a module in a library

Python Libraries

Generic Library Structure and Notation



- library
- library.module
- library.module.class
- library.module.class.method()

Example of Importing and Using a Library Function

```
In [1]: import numpy as np
```

module

```
In [2]: import statsmodels.api as sm
```

class

```
In [3]: import statsmodels.formula.api as smf

# Load data
In [4]: dat = sm.datasets.get_rdataset("Guerry", "HistData").data

# Fit regression model (using the natural log of one of the regressors)
In [5]: results = smf.ols('Lottery ~ Literacy + np.log(Pop1831)', data=dat).fit()
```

method

```
# Inspect the results
In [6]: print(results.summary())
                            OLS Regression Results
==============================================================================
Dep. Variable:                Lottery   R-squared:                       0.348
Model:                            OLS   Adj. R-squared:                  0.333
Method:                 Least Squares   F-statistic:                     22.20
Date:                Fri, 21 Feb 2020   Prob (F-statistic):           1.90e-08
Time:                        13:59:15   Log-Likelihood:                -379.82
No. Observations:                  86   AIC:                             765.6
```

NumPy

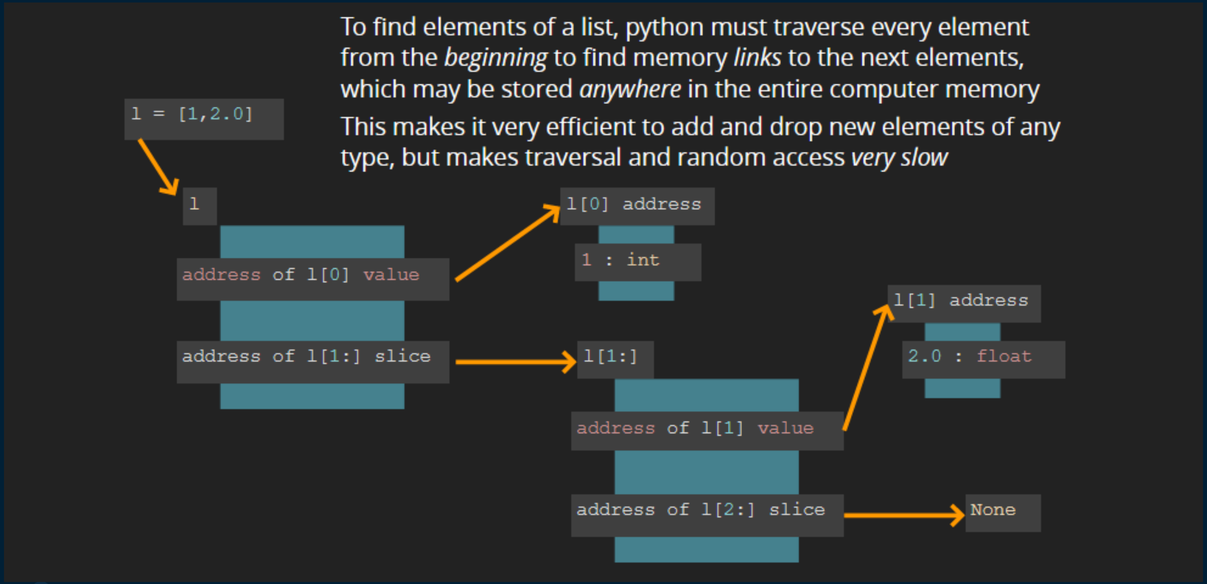
Data in the Python Data Science Ecosystem

Reminders

- *Basic Python* provides three basic data types (numeric, string, Boolean) which can be structured in four basic structures:
  - List: Changeable (mutable) ordered sequence of mixed types
  - Tuple: Unchangeable (unmutable) ordered sequence of mixed types
  - Set: Unordered and unindexed collection of unique elements of mixed types
  - Dictionary: Ordered collection of key-value pairs of mixed types

Data Structures: Basic Python

In order to provide the flexibility to handle mixed types, Python implements its data structures as linked lists:

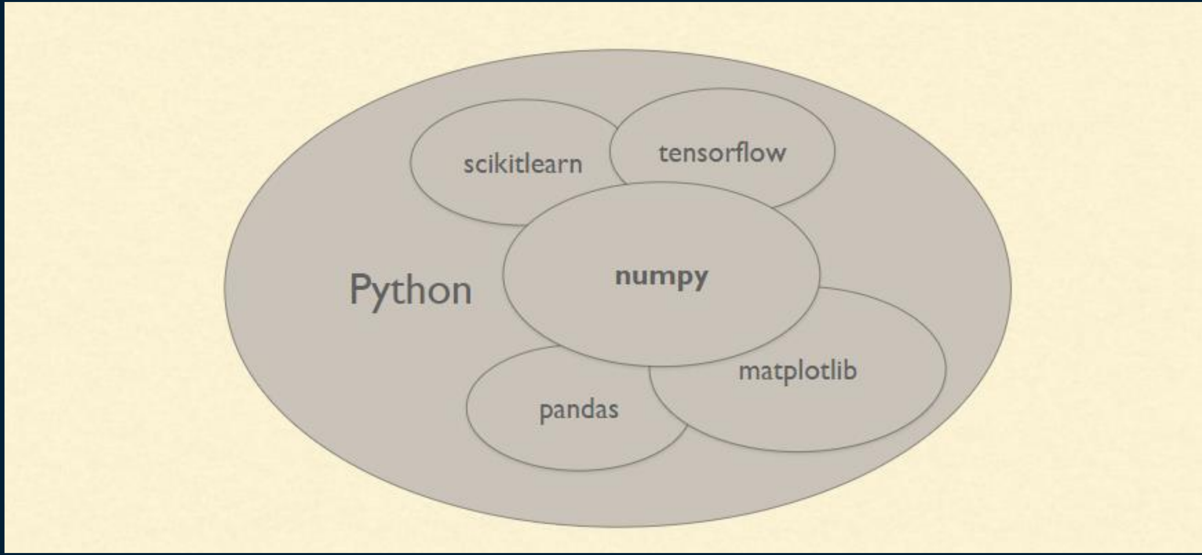


A Partial Solution: NumPy (“Numerical Python”)

- Python was not originally designed for numerical computing, but its other features quickly attracted the scientific and engineering community
- NumPy provides a new core data type for large, homogenous, multi-dimensional arrays and matrices (“ndarrays”) along with a large collection of high-level mathematical functions to operate on these arrays.
- Very similar to Matlab functionality
- Most more modern data science packages, including Pandas, SciKitLearn, and TensorFlow are built on top of NumPy

Libraries

NumPy and Pandas



NumPy

Library contents:

- ndarray: an efficient multidimensional array type
- Mathematical functions for fast operations on arrays without having to write loops
- Linear algebra, random number generation, and other transforms

NumPy

Basic Functionality

- Fast vectorized array operations for data manipulation and cleaning, subsetting and filtering, transformation, and many other kinds of computations
- Common array algorithms like sorting, unique, and set operations
- Efficient descriptive statistics and aggregating/summarizing data
- Relational data manipulations for merging and joining heterogeneous datasets
- Expressing conditional logic as array expressions instea of lloops
- Group-wise data manipulations (aggregation, transformation, function application)

NumPy Overview

Outline

- Creating NumPy arrays
- Arithmetic with NumPy arrays
- Accessing NumPy array data
- Universal Functions (ufuncs)

The NumPy ndarray ("NumPy Array")

Creating a NumPy Array

- The np.array function converts any sequence-like object to a NumPy array containing the passed data:

Create arrays from Python lists

In [10]:

```python
import numpy as np
mylist = [1, 2, 3]
myarray = np.array(mylist)
print(type(myarray))
print(myarray)
print(myarray.shape)
```

```
<class 'numpy.ndarray'>
[1 2 3]
(3,)
```

The NumPy ndarray ("NumPy Array")

Creating a NumPy Array

- Alternately, there are a number of numpy functions for creating new arrays:

Create arrays from scratch

In [24]:
```python
# Create an length-10 integer array filled with zeros
myarray = np.zeros(10, dtype=int)
print(type(myarray))
print(myarray.shape)
myarray
```

```
<class 'numpy.ndarray'>
(10,)
```

Out[24]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

In [25]:
```python
#  Create a 3x5 floating-point array filled with 1s
mymatrix = np.ones((3,5), dtype=float)
print(type(mymatrix))
print(mymatrix.shape)
mymatrix
```

```
<class 'numpy.ndarray'>
(3, 5)
```

Out[25]:
```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

NumPy Array Attributes

Function	Description
ndarray.shape	Tuple of array dimensions.
ndarray.ndim	Number of array dimensions.
ndarray.itemsize	Length of one array element in bytes.
ndarray.size	Number of elements in the array.
ndarray.dtype	Data-type of the array’s elements.
ndarray.T	The transposed array.
ndarray.real	The real part of the array.
ndarray.imag	The imaginary part of the array.
ndarray.flags	Information about the memory layout of the array.

The NumPy ndarray ("NumPy Array")

Data Types

- The .dtype attribute returns the datatype of the elements in the NumPy array:

```
In [36]:
arr1 = np.array([1, 2, 3], dtype=np.float64)
print(type(arr1))
print(arr1.dtype)

<class 'numpy.ndarray'>
float64
```

NumPy Overview

Arithmetic with NumPy Arrays

Arithmetic with NumPy arrays

In [42]:
```
myarray1 = np.array([1,2,3])
myarray2 = np.array([4,5,6])
myarray1 + myarray2
```

Out[42]:
```
array([5, 7, 9])
```

In [43]:
```
myarray1 * myarray2
```

Out[43]:
```
array([ 4, 10, 18])
```

NumPy Overview

Arithmetic with NumPy Arrays

Scalars are extended in NumPy arrays

In [44]:
```
1/myarray1
```
Out[44]:
```
array([1.        , 0.5       , 0.33333333])
```

In [45]:
```
myarray2 ** 2
```
Out[45]:
```
array([16, 25, 36], dtype=int32)
```

NumPy Overview

Arithmetic with NumPy Arrays

In [51]:
```python
# Comparisons between arrays
ndarray1 = np.array([[1,2,3], [4,5,6]])
ndarray2 = np.array([[4,1,3], [5,5,5]])
ndarray1 > ndarray2
```

Out[51]:
```
array([[False,  True, False],
       [False, False,  True]])
```

NumPy Overview

Accessing NumPy Array Data

Accessing NumPy Array Data

One-dimensional arrays act very similar to Python lists:

In [54]:
```
arr = np.arange(10)
arr
```
Out[54]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [55]:
```
arr[5]
```
Out[55]:
```
5
```

In [56]:
```
arr[5:8]
```
Out[56]:
```
array([5, 6, 7])
```

In [57]:
```
arr[5:8] = 12
arr
```
Out[57]:
```
array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

NumPy Overview

Accessing NumPy Array Data

NumPy array slices are views on the original data, not copies:

In [60]:
```
array_slice = arr[5:8]
array_slice
```

Out[60]: `array([12, 12, 12])`

In [62]:
```
array_slice[1] = 12345
arr
```

Out[62]:
```
array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,
           9])
```

NumPy Overview

Accessing NumPy Array Data

Higher dimensional arrays are more complicated

- In a two-dimensional array, the elements at each index are one-dimensional arrays:

In [70]:
```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr2d.shape)
arr2d
```

```
(3, 3)
```

Out[70]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [64]:
```
arr2d[2]
```

Out[64]:
```
array([7, 8, 9])
```

In [65]:
```
arr2d[0][2]
```

Out[65]:
```
3
```

In [67]:
```
arr2d[0,2]   # Equivalent to arr2d[0][2]
```

Out[67]:
```
3
```

NumPy Overview

Accessing NumPy Array Data

- In multidimensional arrays, if you omit later indices, the returned object will be a lower dimensional ndarray consisting of all the data along the higher dimensions:

In [74]:
```
arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr3d.shape)
arr3d
```

```
(2, 2, 3)
```

Out[74]:
```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

In [76]:
```
#  arr3d[0] is a 2x3 array
arr3d[0]
```

Out[76]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [79]:
```
# arr3d[1,0] is a one-dimensional array
arr3d[1,0]
```

Out[79]:
```
array([7, 8, 9])
```

NumPy Overview

Accessing NumPy Array Data

Indexing With Slicing

Line one-dimensional objets such as Python lists, ndarrays can be sliced with the familiar syntax:

```
In [80]:  arr

Out[80]: array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,
                 9])

In [81]:  arr[1:6]

Out[81]: array([ 1,  2,  3,  4, 12])
```

NumPy Overview

Accessing NumPy Array Data

Two dimensional array slicing is a bit different:

```
In [83]: arr2d
Out[83]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

In [85]: arr2d[:2]  # "Select the first two rows of arr2d"
Out[85]: array([[1, 2, 3],
                [4, 5, 6]])

In [89]: arr2d[:2, :1]  # "Select the first two rows and the first column of arr2d"
Out[89]: array([[1],
                [4]])
```

NumPy Overview

Accessing NumPy Array Data

Boolean Indexing

```
Suppose we have a 7x4 array of numbers and 7-item array of names where each data row corresponds to one of the names:
```

In [94]:
```python
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
print(names)
data = np.random.randn(7, 4)
data
```

```
['Bob' 'Joe' 'Will' 'Bob' 'Will' 'Joe' 'Joe']
```

Out[94]:
```
array([[-6.54134621e-01, -1.08030049e+00,  1.55292805e+00,
        -6.27196559e-01],
       [ 6.51766070e-01,  1.06891461e+00,  3.72774427e-01,
         2.40737263e-03],
       [ 4.96049869e-01,  6.77194860e-01,  2.45571971e-01,
         5.74721359e-02],
       [-1.79142071e-01, -1.32380290e+00,  1.01670969e+00,
         1.64646942e+00],
       [ 7.28499563e-04,  1.13817708e+00,  6.30855573e-01,
        -1.48660171e+00],
       [ 6.84287038e-01, -1.40920969e+00,  2.71532915e-01,
         1.05548597e+00],
       [-8.24158232e-01, -2.20244395e+00,  1.34045174e+00,
        -1.66697121e+00]])
```

Now, suppose we want to return the rows that correspond to "Bob":

In [95]:
```python
names == "Bob"
```

Out[95]:
```
array([ True, False, False,  True, False, False, False])
```

In [96]:
```python
data[names == 'Bob']
```

Out[96]:
```
array([[-0.65413462, -1.08030049,  1.55292805, -0.62719656],
       [-0.17914207, -1.3238029 ,  1.01670969,  1.64646942]])
```

NumPy Overview

Accessing NumPy Array Data

"Fancy Indexing"

In [99]:
```python
#  Create an 8x4 array:
arr = np.empty((8, 4))
for i in range(8):
    arr[i] = i
arr
```

Out[99]:
```
array([[0., 0., 0., 0.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [3., 3., 3., 3.],
       [4., 4., 4., 4.],
       [5., 5., 5., 5.],
       [6., 6., 6., 6.],
       [7., 7., 7., 7.]])
```

NumPy Overview

Accessing NumPy Array Data

In [102]:
```
# To select out a subset of rows in a particular order, you pass a list or ndarray of integers specifying the orcer:
arr[[-4,-3,-6]]

```

Out[102]:
```
array([[4., 4., 4., 4.],
       [5., 5., 5., 5.],
       [2., 2., 2., 2.]])
```

In [103]:
```
#  Using negative indices selects rows from the end:
arr[[-4,3,0,6]]
```

Out[103]:
```
array([[4., 4., 4., 4.],
       [3., 3., 3., 3.],
       [0., 0., 0., 0.],
       [6., 6., 6., 6.]])
```

NumPy Overview

Unary Universal Functions (ufuncs)

Universal Functions (ufunc)

NumPy universal functions perform element-wise opearations on data in ndarrays

Unary ufunce operate on a single array and return a single array

In [106]:
```
arr = np.arange(10)
arr
```

Out[106]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

NumPy Overview

Unary Universal Functions (ufuncs)

```
In [107]:  1  np.sqrt(arr)

Out[107]: array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
                 2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ])

In [108]:  1  np.exp(arr)

Out[108]: array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
                 5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
                 2.98095799e+03, 8.10308393e+03])
```

Unary ufuncs (1 of 2)

Function	Description
abs, fabs	Compute the absolute value element-wise for integer, floating-point, or complex values
sqrt	Compute the square root of each element (equivalent to arr ** 0.5)
square	Compute the square of each element (equivalent to arr ** 2)
exp	Compute the exponent e^x of each element
log, log10, log2, log1p	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
sign	Compute the sign of each element: 1 (positive), 0 (zero), or –1 (negative)
ceil	Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)
floor	Compute the floor of each element (i.e., the largest integer less than or equal to each element)

Unary ufuncs (3 of 2)

Function	Description
rint	Round elements to the nearest integer, preserving the dtype
modf	Return fractional and integral parts of array as a separate array
isnan	Return boolean array indicating whether each value is NaN (Not a Number)
isfinite, isinf	Return boolean array indicating whether each element is finite (non-inf, non-NaN) or infinite, respectively
cos, cosh, sin, sinh, tan, tanh	Regular and hyperbolic trigonometric functions
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	Inverse trigonometric functions
logical_not	Compute truth value of not x element-wise (equivalent to ~arr).

NumPy Overview

Binary Universal Functions (ufuncs)

Binary ufuncs take two arrays and return a single array as a result:

In [111]:
```
x = np.random.randn(8)
print(x)
y = np.random.randn(8)
print(y)
```

```
[ 0.45351625  0.23879131  0.3489304   2.02527308  0.482391   -0.06742147
 -0.52005574  0.69599422]
[ 2.48585555  0.61008131  0.02557682 -0.50306261  0.62009976 -1.87348698
 -0.53826574  1.91759217]
```

In [112]:
```
np.maximum(x,y)
```

Out[112]:
```
array([ 2.48585555,  0.61008131,  0.3489304 ,  2.02527308,  0.62009976,
       -0.06742147, -0.52005574,  1.91759217])
```

Binary ufuncs (1 of 2)

Function	Description
add	Add corresponding elements in arrays
subtract	Subtract elements in second array from first array
multiply	Multiply array elements
divide, floor_divide	Divide or floor divide (truncating the remainder)
power	Raise elements in first array to powers indicated in second array
maximum, fmax	Element-wise maximum; fmax ignores NaN
minimum, fmin	Element-wise minimum; fmin ignores NaN

Binary ufuncs (2 of 2)

Function	Description
mod	Element-wise modulus (remainder of division)
copysign	Copy sign of values in second argument to values in first argument
greater, greater_equal, less, less_equal, equal, not_equal	Perform element-wise comparison, yielding boolean array (equivalent to infix operators >, >=, <, <=, ==, !=)
logical_and, logical_or, logical_xor	Compute element-wise truth value of logical operation (equivalent to infix operators & |, ^)

NumPy Overview

NumPy where() method

NumPy where method

numpy.where is a vectorized version of the ifelse function

In [126]:
```
xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
cond = np.array([True, False, True, True, False])
```

In [129]:
```
#  Suppose we want to take a value from xarr when the corresponding value in cond is TRUE
#  and otherwise take the value from yarr:
```

In [131]:
```
np.where(cond, xarr, yarr)
```

Out[131]:
```
array([1.1, 2.2, 1.3, 1.4, 2.5])
```

Pandas

The Rest of the Solution: Pandas

“Python Data Analysis Library”

Most data science datasets are heterogeneous (contain columns with different datatypes)

Pandas provides heterogeneous array types with naming for rows and columns:

    Series: One-dimensional array
    Dataframes: Multi-dimensional array

It also contains a third data structure known as an *Index Object*

Pandas Overview

Series Objects

- A Series is a one-dimensional array-like object containing:
  - A sequence of values (similar to NumPy arrays)
  - An associated array of data labels called its index

Pandas Overview

Series Objects (Default Index Values)

```
In [134]:
1 import pandas as pd
2 my_series = pd.Series([4, 7, -5, 3])
3 my_series

Out[134]:
0    4
1    7
2   -5
3    3
dtype: int64
```

Index Values

Pandas Overview

Series Objects: User-Specified Index

```
In [136]:  1  my_series_2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
           2  my_series_2

Out[136]: d    4
          b    7
          a   -5
          c    3
          dtype: int64
```

Index   Values

Pandas Overview

Using Index Labels to Select Values or Sets of Values

```
In [137]:  my_series_2['a']

Out[137]: -5

In [138]:  my_series_2['d'] = 6
           my_series_2[['c', 'a', 'd']]

Out[138]: c    3
          a   -5
          d    6
          dtype: int64
```

Note notation. Series takes a list of index values as a single parameter.

Pandas Overview

Series Arithmetic Examples

```
In [139]:  1  my_series_2[my_series_2 > 0]

Out[139]: d    6
          b    7
          c    3
          dtype: int64

In [141]:  1  my_series_2 * 2

Out[141]: d    12
          b    14
          a   -10
          c     6
          dtype: int64
```

Returns an array of Booleans

Pandas Overview

Series Arithmetic Examples

```
In [143]:  1  np.exp(my_series_2)

Out[143]: d       403.428793
          b      1096.633158
          a         0.006738
          c        20.085537
          dtype: float64
```

Pandas Overview

Dictionary Functions

Dictionary Functions

In [144]:
```
'b' in my_series_2
```
Out[144]: True

In [145]:
```
'e' in my_series_2
```
Out[145]: False

Pandas Overview

Creating Series From Python Dictionary

In [147]:
```
my_dict =  {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
my_series_3 = pd.Series(my_dict)
my_series_3
```

Out[147]:
```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

Pandas Overview

Specifying the Index Order

Specifying the index order

In [149]:
```
states = ['California', 'Ohio', 'Oregon', 'Texas']
my_series_4 = pd.Series(my_dict, index=states)
my_series_4
```

Out[149]:
```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

No value for “California” was found, so NaN (not a number) was returned

Pandas Overview

Testing for NaNs

```
In [151]:  1  pd.isnull(my_series_4)

Out[151]: California     True
          Ohio          False
          Oregon        False
          Texas         False
          dtype: bool

In [152]:  1  my_series_4.isnull()

Out[152]: California     True
          Ohio          False
          Oregon        False
          Texas         False
          dtype: bool
```

Available in Pandas as a function or a method (equivalent)

Pandas Overview

Automatic Alignment by Index Label

```
In [153]:  1  my_series_3

Out[153]: Ohio      35000
          Texas     71000
          Oregon    16000
          Utah       5000
          dtype: int64

In [154]:  1  my_series_4

Out[154]: California        NaN
          Ohio          35000.0
          Oregon        16000.0
          Texas         71000.0
          dtype: float64

In [155]:  1  my_series_3 + my_series_4

Out[155]: California         NaN
          Ohio           70000.0
          Oregon         32000.0
          Texas         142000.0
          Utah               NaN
          dtype: float64
```

Pandas Overview

Both Series and its Index Have Names

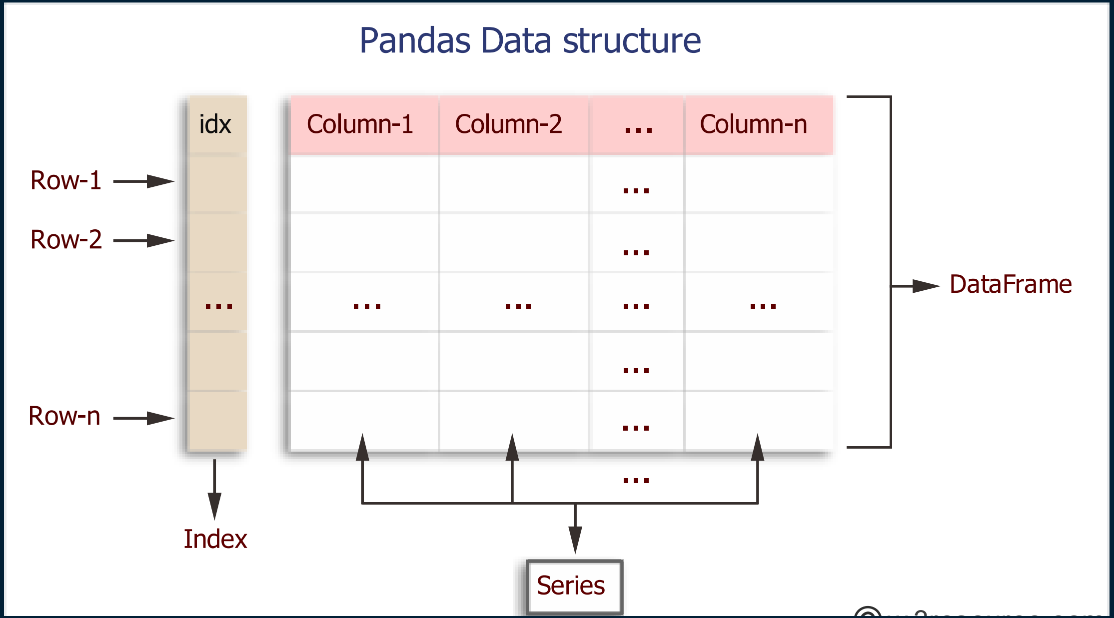
In [158]:
```
my_series_4.name = "Population"
my_series_4.index_name = "State"
my_series_4
```

Out[158]:
```
California          NaN
Ohio            35000.0
Oregon          16000.0
Texas           71000.0
Name: Population, dtype: float64
```

Pandas Dataframes

- The primary data structure used in data science
- Designed for working with tabular, heterogeneous data
- Consists of a number of Pandas series "glued together" with a common index
  - Every column (series) must be of the same length

Pandas Dataframes



Pandas Dataframes

Creating Dataframe From Python Dictionary

Creating dataframes

In [168]:

```
#  Creating a dataframe from a Python dictionary

data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
type(data)
```

Out[168]: dict

In [171]:

```
import pandas as pd
dataframe_2 = pd.DataFrame(data)
dataframe_2
```

Out[171]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

Pandas Dataframes

Retrieving Pandas Series From Dataframe Columns

Retrievine Pandas Series objects from dataframe columns

```
In [172]:  1  dataframe_2['state']
```

```
Out[172]: 0      Ohio
          1      Ohio
          2      Ohio
          3    Nevada
          4    Nevada
          5    Nevada
          Name: state, dtype: object
```

```
In [173]:  1  dataframe_2.state
```

```
Out[173]: 0      Ohio
          1      Ohio
          2      Ohio
          3    Nevada
          4    Nevada
          5    Nevada
          Name: state, dtype: object
```

Pandas Dataframes

Retrieve Rows by Row Number

Retrieve rows by row number

In [199]:
```
dataframe_2.iloc[3]
```

Out[199]:
```
state     Nevada
year        2001
pop          2.4
Name: 3, dtype: object
```

In [200]:
```
dataframe_2.iloc[0:4]
```

Out[200]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4

Pandas Dataframes

Retrieve Rows by Content

Retriev rows by content

In [201]:
```
dataframe_2[dataframe_2['state'] == "Ohio"]
```

Out[201]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6

Pandas Dataframes

Modifying or Creating Columns by Assignment

Modifying or creating columns by assignment

In [206]:

```
dataframe_2['debt'] = 16.5
dataframe_2
```

Out[206]:

	state	year	pop	debt
0	Ohio	2000	1.5	16.5
1	Ohio	2001	1.7	16.5
2	Ohio	2002	3.6	16.5
3	Nevada	2001	2.4	16.5
4	Nevada	2002	2.9	16.5
5	Nevada	2003	3.2	16.5

In [207]:

```
dataframe_2['debt'] = np.arange(6)
dataframe_2
```

Out[207]:

	state	year	pop	debt
0	Ohio	2000	1.5	0
1	Ohio	2001	1.7	1
2	Ohio	2002	3.6	2
3	Nevada	2001	2.4	3
4	Nevada	2002	2.9	4
5	Nevada	2003	3.2	5

Pandas Dataframes

Adding a New Column of Booleans Based on Conditional Test

Adding a New Column of Booleans Based on Conditional Test

In [208]:
```
dataframe_2['eastern'] = dataframe_2['state'] == "Ohio"
dataframe_2
```

Out[208]:

	state	year	pop	debt	eastern
0	Ohio	2000	1.5	0	True
1	Ohio	2001	1.7	1	True
2	Ohio	2002	3.6	2	True
3	Nevada	2001	2.4	3	False
4	Nevada	2002	2.9	4	False
5	Nevada	2003	3.2	5	False

Pandas Dataframes

Deleting a Column

Deleting a Column

In [209]:

```python
del dataframe_2['eastern']
dataframe_2
```

Out[209]:

	state	year	pop	debt
0	Ohio	2000	1.5	0
1	Ohio	2001	1.7	1
2	Ohio	2002	3.6	2
3	Nevada	2001	2.4	3
4	Nevada	2002	2.9	4
5	Nevada	2003	3.2	5

Pandas Dataframes

Values Attribute

As with Series, the values attribute returns the data contained in a DataFrame as a two-dimensional ndarray:

```
In [217]:  1  print(dataframe_2.values)
           2  type(dataframe_2.values)

[['Ohio' 2000 1.5 0]
 ['Ohio' 2001 1.7 1]
 ['Ohio' 2002 3.6 2]
 ['Nevada' 2001 2.4 3]
 ['Nevada' 2002 2.9 4]
 ['Nevada' 2003 3.2 5]]

Out[217]: numpy.ndarray
```

Pandas Dataframes

Transforming a Dataframe

Transforming a dataframe

In [210]:
```
dataframe_2.T
```

Out[210]:

	0	1	2	3	4	5
state	Ohio	Ohio	Ohio	Nevada	Nevada	Nevada
year	2000	2001	2002	2001	2002	2003
pop	1.5	1.7	3.6	2.4	2.9	3.2
debt	0	1	2	3	4	5

Pandas Dataframes

Creating a Dataframe From NumPy Arrays and Python Lists

Creating a dataframe from NumPy arrays and Python lists

In [204]:

```
myarray = np.array([[1, 2, 3], [4, 5, 6]])
rownames = ['a', 'b']
colnames = ['one', 'two', 'three']
dataframe_3 = pd.DataFrame(myarray, index=rownames, columns=colnames)
dataframe_3
```

Out[204]:

	one	two	three
a	1	2	3
b	4	5	6

Pandas Index Objects

- Pandas index objects hold the axis labels and other meta-data (like the axis name or names)
- Any array or sequence used when constructing a Series of a DataFrame is automatically converted to an an Index:

Pandas Index Objects

Index Object

In [227]:
```
series_2 = pd.Series(range(3), index=['a', 'b', 'c'])
series_2_index = series_2.index
series_2_index
```

Out[227]:
```
Index(['a', 'b', 'c'], dtype='object')
```

In [228]:
```
series_2_index[1:3]
```

Out[228]:
```
Index(['b', 'c'], dtype='object')
```

Pandas Index Objects

Index Objects are Immutable

Index objects are immutable

In [229]:

```
1 series_2_index[1] = 'd'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-229-944413992c45> in <module>
----> 1 series_2_index[1] = 'd'

~\anaconda3\lib\site-packages\pandas\core\indexes\base.py in __setitem__(self, key, value)
   4275     @final
   4276     def __setitem__(self, key, value):
-> 4277         raise TypeError("Index does not support mutable operations")
   4278 
   4279     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

Basic Pandas Functionality

Reindexing

Reindexing

Create a new Pandas object with the data conformed to a new index

In [231]:
```
Series_3 = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
Series_3
```

Out[231]:
```
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

In [234]:
```
Series_4 = Series_3.reindex(['a', 'b', 'c', 'd', 'e'])
Series_4
```

Out[234]:
```
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

Basic Pandas Functionality

Dropping Entities from an Axis - Series

In [236]:
```python
obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
obj
```

Out[236]:
```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

In [238]:
```python
new_obj = obj.drop('c')
new_obj
```

Out[238]:
```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

In [240]:
```python
new_obj = obj.drop(['d','c'])
new_obj
```

Out[240]:
```
a    0.0
b    1.0
e    4.0
dtype: float64
```

Basic Pandas Functionality

Dropping Entities from an Axis - DataFrame

In [245]:
```
dataframe_3 = pd.DataFrame(np.arange(16).reshape((4, 4)),
                           index=['Ohio', 'Colorado', 'Utah', 'New York'],
                           columns=['one', 'two', 'three', 'four'])
dataframe_3
```

Out[245]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Basic Pandas Functionality

Dropping Entities from an Axis - DataFrame

In [247]:
```
#   Drop rows
dataframe_3.drop(['Colorado', 'Ohio'])
```

Out[247]:

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

In [249]:
```
# Drop columns
dataframe_3.drop('two', axis=1)
```

Out[249]:

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

Basic Pandas Functionality

Indexing, Selection, and Filtering - Series

- Series indexing works similar to NumPy arrays except that you can use the Series's index values instead of only integers:

Indexing, Selection, and Filtering

In [250]:
```
obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
obj
```

Out[250]:
```
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64
```

Basic Pandas Functionality

Indexing, Selection, and Filtering - Series

```
In [251]: obj['b']
Out[251]: 1.0

In [252]: obj[1]
Out[252]: 1.0

In [253]: obj[2:4]
Out[253]: c    2.0
          d    3.0
          dtype: float64

In [254]: obj[['b','a','d']]
Out[254]: b    1.0
          a    0.0
          d    3.0
          dtype: float64

In [255]: obj[[1,3]]
Out[255]: b    1.0
          d    3.0
          dtype: float64

In [256]: obj[obj<2]
Out[256]: a    0.0
          b    1.0
          dtype: float64
```

Basic Pandas Functionality

Indexing, Selection, and Filtering - Series

In [257]:

```python
#  Setting values in a series
obj['b':'c'] = 5
obj
```

Out[257]:

```
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

Basic Pandas Functionality

Indexing, Selection, and Filtering - Dataframe

In [260]:

```python
# Dataframe Indexing
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                    index=['Ohio', 'Colorado', 'Utah', 'New York'],
                    columns=['one', 'two', 'three', 'four'])
data
```

Out[260]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Basic Pandas Functionality

Indexing, Selection, and Filtering - Dataframe

In [266]:

```
print(data['two'])
type(data['two'])
```

Returns a series

```
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int32
```

Out[266]: pandas.core.series.Series

In [267]:

```
print(data[['two']])
type(data[['two']])
```

Returns a dataframe

```
          two
Ohio        1
Colorado    5
Utah        9
New York   13
```

Out[267]: pandas.core.frame.DataFrame

In [263]:

```
data[['three', 'one']]
```

Out[263]:

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12

Basic Pandas Functionality

Indexing, Selection, and Filtering - Dataframe

In [272]:
```
data[:2]
```
Out[272]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

In [270]:
```
data[data['three'] > 5]
```
Out[270]:

	one	two	three	four
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Basic Pandas Functionality

Indexing, Selection, and Filtering - Dataframe

In [273]:
```
# Indexing with a Boolean dataframe
data < 5
```

Out[273]:

	one	two	three	four
Ohio	True	True	True	True
Colorado	True	False	False	False
Utah	False	False	False	False
New York	False	False	False	False

In [276]:
```
data[data< 5] = 0
data
```

Out[276]:

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Basic Pandas Functionality

Indexing, Selection, and Filtering - Dataframe

Selection with loc (using axis labels) and iloc (using integers):

```
In [277]:
# Selection using loc
data.loc['Colorado', ['two', 'three']]

Out[277]:
two      5
three    6
Name: Colorado, dtype: int32

In [278]:
# selection using iloc
data.iloc[2, [3, 0, 1]]

Out[278]:
four    11
one      8
two      9
Name: Utah, dtype: int32
```

Basic Pandas Functionality

Indexing, Selection, and Filtering - Dataframe

Selection with loc (using axis labels) and iloc (using integers):

In [279]:
```
data.iloc[2]
```

Out[279]:
```
one       8
two       9
three    10
four     11
Name: Utah, dtype: int32
```

In [280]:
```
data.iloc[[1, 2], [3, 0, 1]]
```

Out[280]:

	four	one	two
Colorado	7	0	5
Utah	11	8	9

Basic Pandas Functionality

Indexing, Selection, and Filtering - Dataframe

Indexing functions with slices

In [281]:
```
# Indexing functions with slicing
data.loc[:'Utah', 'two']
```

Out[281]:
```
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int32
```

In [282]:
```
data.iloc[:, :3][data.three > 5]
```

Out[282]:

	one	two	three
Colorado	0	5	6
Utah	8	9	10
New York	12	13	14

Basic Pandas Functionality

Indexing Options with DataFrames

Type	Notes
df[val]	Select single column or sequence of columns from the DataFrame; special case conveniences: boolean array (filter rows), slice (slice rows), or boolean DataFrame (set values based on some criterion)
df.loc[val]	Selects single row or subset of rows from the DataFrame by label
df.loc[:, val]	Selects single column or subset of columns by label
df.loc[val1, val2]	Select both rows and columns by label
df.iloc[where]	Selects single row or subset of rows from the DataFrame by integer position
df.iloc[:, where]	Selects single column or subset of columns by

Basic Pandas Functionality

Indexing Options with DataFrames

Type	Notes
df.iloc[:, where]	Selects single column or subset of columns by integer position
df.iloc[where_i, where_j]	Select both rows and columns by integer position
df.at[label_i, label_j]	Select a single scalar value by row and column label
df.iat[i, j]	Select a single scalar value by row and column position (integers)
reindex method	Select either rows or columns by labels
get_value, set_value methods	Select single value by row and column label

Basic Pandas Functionality

Arithmetic and Data Alignment

Arithmetic operations between Series or Dataframes with different indexes requires special handling:

In [284]:
```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'
print(s1)
print(s2)
```

```
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

In [285]:
```
s1+s2
```

Out[285]:
```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

Basic Pandas Functionality

Arithmetic and Data Alignment - Dataframes

In [287]:

```python
# Dataframes
df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'), index=['Ohio', 'Texas', 'Colorado'])
df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'), index=['Utah', 'Ohio', 'Texas', 'Oregon'])
print(df1)
print(df2)
```

```
            b    c    d
Ohio      0.0  1.0  2.0
Texas     3.0  4.0  5.0
Colorado  6.0  7.0  8.0
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0
```

Basic Pandas Functionality

Arithmetic and Data Alignment - Dataframes

In [288]:

```
df1 + df2
```

Out[288]:

	b	c	d	e
Colorado	NaN	NaN	NaN	NaN
Ohio	3.0	NaN	6.0	NaN
Oregon	NaN	NaN	NaN	NaN
Texas	9.0	NaN	12.0	NaN
Utah	NaN	NaN	NaN	NaN

Basic Pandas Functionality

Function Application and Mapping

NumPy ufuncs also work with Pandas objects:

In [289]:

```python
# NumPy ufuncs also work with Pandas objects:
frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
                     index=['Utah', 'Ohio', 'Texas', 'Oregon'])
frame
```

Out[289]:

	b	d	e
Utah	1.531637	-0.685977	0.041550
Ohio	-0.609879	-0.731267	0.281189
Texas	0.744102	-0.163679	0.817626
Oregon	1.733892	-1.285659	-0.137677

In [290]:

```python
np.abs(frame)
```

Out[290]:

	b	d	e
Utah	1.531637	0.685977	0.041550
Ohio	0.609879	0.731267	0.281189
Texas	0.744102	0.163679	0.817626
Oregon	1.733892	1.285659	0.137677

Basic Pandas Functionality

Function Application and Mapping

In addition, the datagrame .apply method applys a function on one-dimensional arrays to each column or row:

Pandas apply method

```
In [291]: f = lambda x: x.max() - x.min()
          frame.apply(f)

Out[291]: b    2.343770
          d    1.121980
          e    0.955303
          dtype: float64
```

```
In [292]: frame.apply(f, axis = "columns")

Out[292]: Utah      2.217614
          Ohio      1.012456
          Texas     0.981305
          Oregon    3.019551
          dtype: float64
```

Basic Pandas Functionality

Sorting Series by Index

```
In [294]:
# Sorting a series by index
obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])
obj.sort_index()

Out[294]:
a    1
b    2
c    3
d    0
dtype: int64
```

Basic Pandas Functionality

Sorting Dataframes by Index

In [295]:
```
# Dataframes can be worted by either axis:
frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
                     index=['three', 'one'],
                     columns=['d', 'a', 'b', 'c'])
frame
```

Out[295]:

	d	a	b	c
three	0	1	2	3
one	4	5	6	7

In [297]:
```
frame.sort_index() # sort by row index
```

Out[297]:

	d	a	b	c
one	4	5	6	7
three	0	1	2	3

In [298]:
```
frame.sort_index(axis=1) # sort by column index
```

Out[298]:

	a	b	c	d
three	1	2	3	0
one	5	6	7	4

Basic Pandas Functionality

Sorting Series by Values

```
In [299]:  # Sort a series by its values
           obj = pd.Series([4, 7, -3, 2])
           obj.sort_values()

Out[299]: 2   -3
          3    2
          0    4
          1    7
          dtype: int64
```

Basic Pandas Functionality

Sorting Dataframe by Values in Single Column

In [300]:

```python
# Sort a dataframe by values of one column
frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
frame
```

Out[300]:

	b	a
0	4	0
1	7	1
2	-3	0
3	2	1

In [302]:

```python
frame.sort_values(by="b")
```

Out[302]:

	b	a
2	-3	0
3	2	1
0	4	0
1	7	1

Basic Pandas Functionality

Sorting Dataframe by Values in Multiple Columns

In [303]:

```
# Sort a dataframe by multiple columns:
frame.sort_values(by=['a','b'])
```

Out[303]:

	b	a
2	-3	0
0	4	0
3	2	1
1	7	1

Pandas Overview

Summarizing and Computing Descriptive Statistics

- Pandas objects include a set of common mathematical and statistical methods
  - Most are summary statistics which extract a single value (like a mean) from a Series or from rows or columns of dataframes.
  - Methods include built-in handling for missing data

Pandas Overview

Summarizing and Computing Descriptive Statistics

Summarizing and Computing Descriptive Statistics

In [304]:

```python
df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],
                    [np.nan, np.nan], [0.75, -1.3]],
                   index=['a', 'b', 'c', 'd'],
                   columns=['one', 'two'])
df
```

Out[304]:

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

In [308]:

```python
# sum method returns a series containing column sums
df.sum()
```

Out[308]:

```
one     9.25
two    -5.80
dtype: float64
```

Pandas Overview

Summarizing and Computing Descriptive Statistics

In [312]:
```
# idxmin and idxmax return "indirect statistics" like the index of the minimum or maximum values
df.idxmax()
```

Out[312]:
```
one      b
two      d
dtype: object
```

In [313]:
```
# Other methods are accumulations:
df.cumsum()
```

Out[313]:

	one	two
a	1.40	NaN
b	8.50	-4.5
c	NaN	NaN
d	9.25	-5.8

Pandas Overview

Summarizing and Computing Descriptive Statistics

In [314]:

```python
# describe produces multiple summary satistics in one call:
df.describe()
```

Out[314]:

	one	two
count	3.000000	2.000000
mean	3.083333	-2.900000
std	3.493685	2.262742
min	0.750000	-4.500000
25%	1.075000	-3.700000
50%	1.400000	-2.900000
75%	4.250000	-2.100000
max	7.100000	-1.300000

Pandas Overview

Summarizing and Computing Descriptive Statistics

```
In [316]:  1  # Describe with non-mumeric data:
           2  obj = pd.Series(['a', 'a', 'b', 'c'] * 4)
           3  obj.describe()

Out[316]: count     16
          unique     3
          top        a
          freq       8
          dtype: object
```

Pandas Overview

Summarizing and Computing Descriptive Statistics

Method	Description
count	Number of non-NA values
describe	Compute set of summary statistics for Series or each DataFrame column
min, max	Compute minimum and maximum values
argmin, argmax	Compute index locations (integers) at which minimum or maximum value obtained, respectively
idxmin, idxmax	Compute index labels at which minimum or maximum value obtained, respectively
quantile	Compute sample quantile ranging from 0 to 1
sum	Sum of values
mean	Mean of values
median	Arithmetic median (50% quantile) of values

Pandas Overview

Summarizing and Computing Descriptive Statistics

Method	Description
mad	Mean absolute deviation from mean value
prod	Product of all values
var	Sample variance of values
std	Sample standard deviation of values
skew	Sample skewness (third moment) of values
kurt	Sample kurtosis (fourth moment) of values
cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute first arithmetic difference (useful for time series)
pct_change	Compute percent changes

Pandas Overview

Unique Values, Value Counts, and Membership

```
In [319]: obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
          obj.unique()

Out[319]: array(['c', 'a', 'd', 'b'], dtype=object)

In [320]: obj.value_counts()

Out[320]: c    3
          a    3
          b    2
          d    1
          dtype: int64
```

Pandas Overview

Unique Values, Value Counts, and Membership

```
In [322]:
mask = obj.isin(['b', 'c'])
mask

Out[322]:
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool
```

Pandas Overview

Unique Values, Value Counts, and Membership

```
In [323]:  1  obj[mask]

Out[323]: 0    c
          5    b
          6    b
          7    c
          8    c
          dtype: object
```

Pandas Overview

Unique Values, Value Counts, and Membership Methods

Method	Description
isin	Compute boolean array indicating whether each Series value is contained in the passed sequence of values
get_indexer	Compute integer indices for each value in an array into another array of distinct values; helpful for data alignment and join-type operations
unique	Compute array of unique values in a Series, returned in the order observed
value_counts	Return a Series containing unique values as its index and frequencies as its values, ordered count in descending order

Reading Data From Files

Python Basics

Loading CSV Files Using Pandas

Loading Data From CSV Files

There are mechanisms to read CSV files in standard Python, NumPy, and Pandas. Generally, in data science usage, the Pandas read_csv function is preferred due to its flexibility and the fact that we almost always want to end up in a dataframe.

Considerations When Loading CSV Data

1. Does the file have a header? If so, it can be used to automatically assign names to each column
2. Does the file have comments indicated by a hash (#)?
3. What is the field delimiter (if not a comma)?
4. Field values with spaces are often in quotes. The default quote character is the double quotation mark. If your file uses something else, you must specify it

Python Basics

Loading CSV Files Using Pandas

Loading Diabetes Dataset with Pandas

Full documentation is here: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

In [107]:

```
from pandas import read_csv
diabetes = read_csv('diabetes.csv')
diabetes.head(10)
```

Out[107]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
5	5	116	74	0	0	25.6	0.201	30	0
6	3	78	50	32	88	31.0	0.248	26	1
7	10	115	0	0	0	35.3	0.134	29	0
8	2	197	70	45	543	30.5	0.158	53	1
9	8	125	96	0	0	0.0	0.232	54	1

DataFrame Objects

Cars Dataset

File: Cars Data

Description: Basic data on cars produced in 2018

Make	Model	DriveTrair	Origin	Type	Cylinders	Engine Siz	Horsepow	Invoice	Length (IN)	MPG (City)	MPG (High	MSRP	Weight (LI	Wheelbase (IN)
Acura	3.5 RL 4dr	Front	Asia	Sedan	6	3.5	225	$39,014	197	18	24	$43,755	3880	115
Acura	3.5 RL w/N	Front	Asia	Sedan	6	3.5	225	$41,100	197	18	24	$46,100	3893	115
Acura	MDX	All	Asia	SUV	6	3.5	265	$33,337	189	17	23	$36,945	4451	106
Acura	NSX coupe	Rear	Asia	Sports	6	3.2	290	$79,978	174	17	24	$89,765	3153	100
Acura	RSX Type S	Front	Asia	Sedan	4	2	200	$21,761	172	24	31	$23,820	2778	101
Acura	TL 4dr	Front	Asia	Sedan	6	3.2	270	$30,299	186	20	28	$33,195	3575	108
Acura	TSX 4dr	Front	Asia	Sedan	4	2.4	200	$24,647	183	22	29	$26,990	3230	105
Audi	A4 1.8T 4d	Front	Europe	Sedan	4	1.8	170	$23,508	179	22	31	$25,940	3252	104
Audi	A4 3.0 4dr	Front	Europe	Sedan	6	3	220	$28,846	179	20	28	$31,840	3462	104
Audi	A4 3.0 con	Front	Europe	Sedan	6	3	220	$38,325	180	20	27	$42,490	3814	105
Audi	A4 3.0 Qua	All	Europe	Sedan	6	3	220	$31,388	179	18	25	$34,480	3627	104
Audi	A4 3.0 Qua	All	Europe	Sedan	6	3	220	$30,366	179	17	26	$33,430	3583	104
Audi	A4 3.0 Qua	All	Europe	Sedan	6	3	220	$40,075	180	18	25	$44,240	4013	105
Audi	A41.8T cor	Front	Europe	Sedan	4	1.8	170	$32,506	180	23	30	$35,940	3638	105

read_csv example

Cars Dataset

Reading cars dataset and skipping header lines ¶

In [329]:

```
cars = pd.read_csv('cars.csv', skiprows = 3)
cars
```

Out[329]:

	Make	Model	DriveTrain	Origin	Type	Cylinders	Engine Size (L)	Horsepower	Invoice	Length (IN)	MPG (City)	MPG (Highway)	MSRP	Weight (LBS)	Wheelbase (IN)
0	Acura	3.5 RL 4dr	Front	Asia	Sedan	6.0	3.5	225	$39,014	197	18	24	$43,755	3880	115
1	Acura	3.5 RL w/Navigation 4dr	Front	Asia	Sedan	6.0	3.5	225	$41,100	197	18	24	$46,100	3893	115
2	Acura	MDX	All	Asia	SUV	6.0	3.5	265	$33,337	189	17	23	$36,945	4451	106
3	Acura	NSX coupe 2dr manual S	Rear	Asia	Sports	6.0	3.2	290	$79,978	174	17	24	$89,765	3153	100
4	Acura	RSX Type S 2dr	Front	Asia	Sedan	4.0	2.0	200	$21,761	172	24	31	$23,820	2778	101
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
423	Volvo	S80 2.9 4dr	Front	Europe	Sedan	6.0	2.9	208	$35,542	190	20	28	$37,730	3576	110
424	Volvo	S80 T6 4dr	Front	Europe	Sedan	6.0	2.9	268	$42,573	190	19	26	$45,210	3653	110
425	Volvo	V40	Front	Europe	Wagon	4.0	1.9	170	$24,641	180	22	29	$26,135	2822	101
426	Volvo	XC70	All	Europe	Wagon	5.0	2.5	208	$33,112	186	20	27	$35,145	3823	109
427	Volvo	XC90 T6	All	Europe	SUV	6.0	2.9	268	$38,851	189	15	20	$41,250	4638	113

428 rows × 15 columns

Skip the first 3 rows

Plotting with Matplotlib and Seaborn

Data Visualization

Matplotlib Package

- 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

https://matplotlib.org/

Data Visualization

Seaborn Package

- Based on Matplotlib
- Provides high-level interface for drawing attractive statistical graphs
- Similar (in style) to the popular ggplot2 library in R

https://seaborn.pydata.org/

Matplotlib Basics

- General usage
  - Call a plotting function with some data – for example, *plot()*
  - Call multiple functions to configure various properties of the plot (color, labels, etc.)
  - Make the plot visible – show()
- For this lecture, I use the following “standards”:
  - Use the classic Matplotlib style: *plt.style.use(‘classic’)*
  - Use the “inline” mode (not the “notebook”) mode:
    - *%matplotlib inline*
- Figures can be saved to files using the savefig() method:
  - fig.savefig(‘my_figure.png’)

Basic Matplotlib Visualizations

- Univariate
  - Histograms
  - Density Plots
  - Box and Whisker Plots
- Multivariate
  - Scatter Plots
  - Correlation matrices

Matplotlib Examples

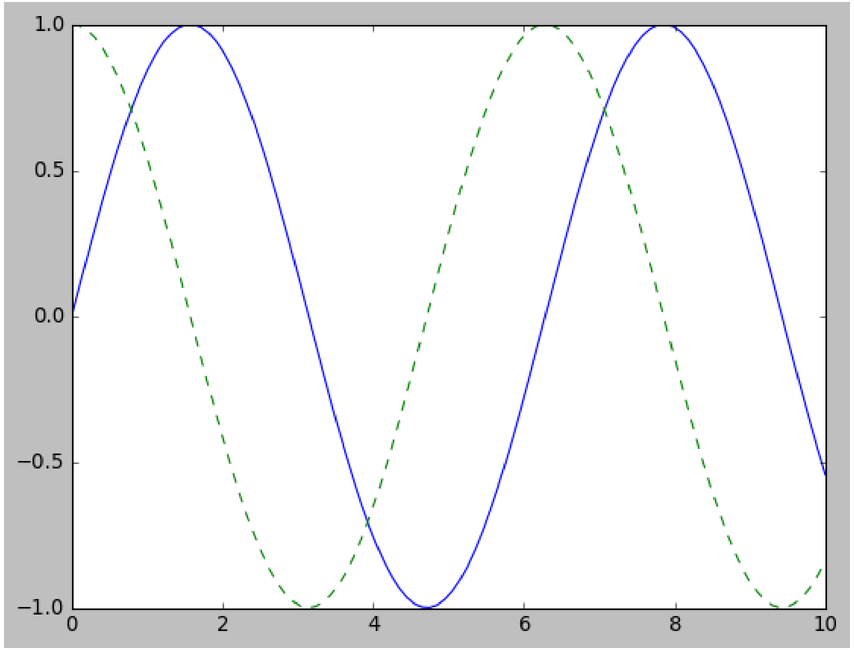
Introductory Example

In [119]:
```
import pandas as pd
import numpy as np
from pandas import read_csv
from matplotlib import pyplot as plt
%matplotlib inline
plt.style.use('classic')
x = pd.Series(np.linspace(0,10,100))
x
```

Out[119]:
```
0      0.00000
1      0.10101
2      0.20202
3      0.30303
4      0.40404
        ...
95     9.59596
96     9.69697
97     9.79798
98     9.89899
99    10.00000
Length: 100, dtype: float64
```

Matplotlib Examples

```
fig = plt.figure()
plt.plot(x, np.sin(x), '-')
plt.plot(x, np.cos(x), '--')
plt.show
fig.savefig('test_matplotlib_figure.png')
```



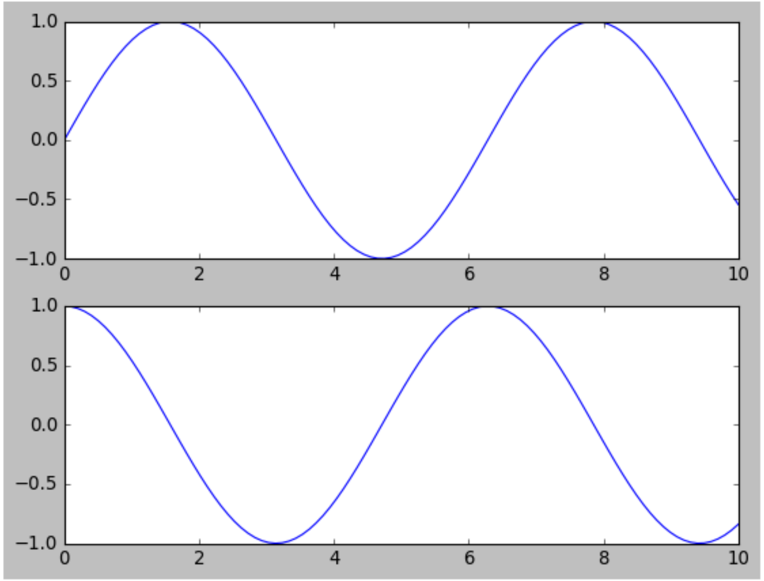
Matplotlib Examples

```
#  Create a two-panel plot

#  First of two sub-plots and set current axis
plt.subplot(2,1,1)  # (rows, columns, number)
plt.plot(x, np.sin(x))

#  Second sub-plot
plt.subplot(2,1,2)  # (rows, columns, number)
plt.plot(x, np.cos(x))

plt.show()
```



Matplotlib Examples

In [122]:
```
diabetes = read_csv('diabetes.csv')
diabetes
```

Out[122]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
...	...	...	...	...	...	...	...	...	...
763	10	101	76	48	180	32.9	0.171	63	0
764	2	122	70	27	0	36.8	0.340	27	0
765	5	121	72	23	112	26.2	0.245	30	0
766	1	126	60	0	0	30.1	0.349	47	1
767	1	93	70	31	0	30.4	0.315	23	0

768 rows × 9 columns

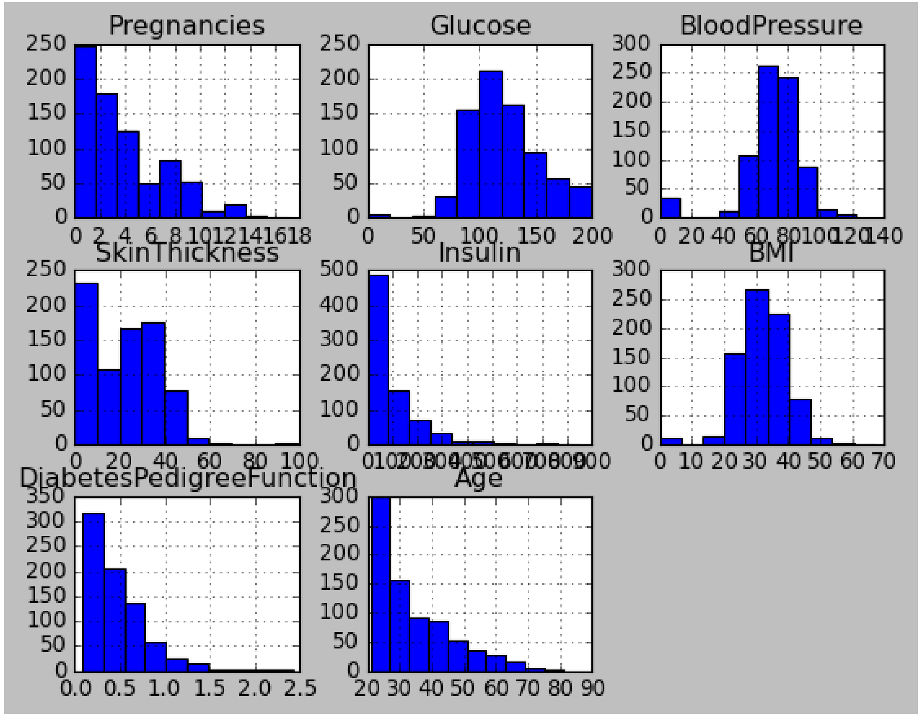
Convert Outcome to be a category

In [123]:
```
diabetes['Outcome'] = diabetes['Outcome'].astype("category")
```

Matplotlib Examples

Histograms

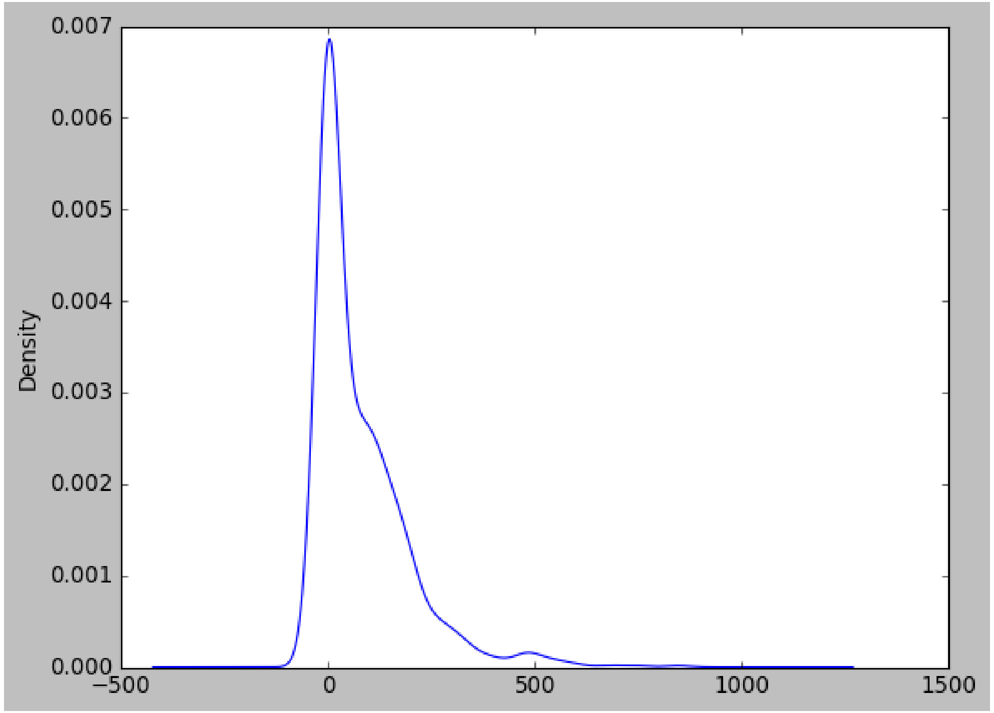
```
diabetes.hist()
plt.show()
```



Matplotlib Examples

Density Plots

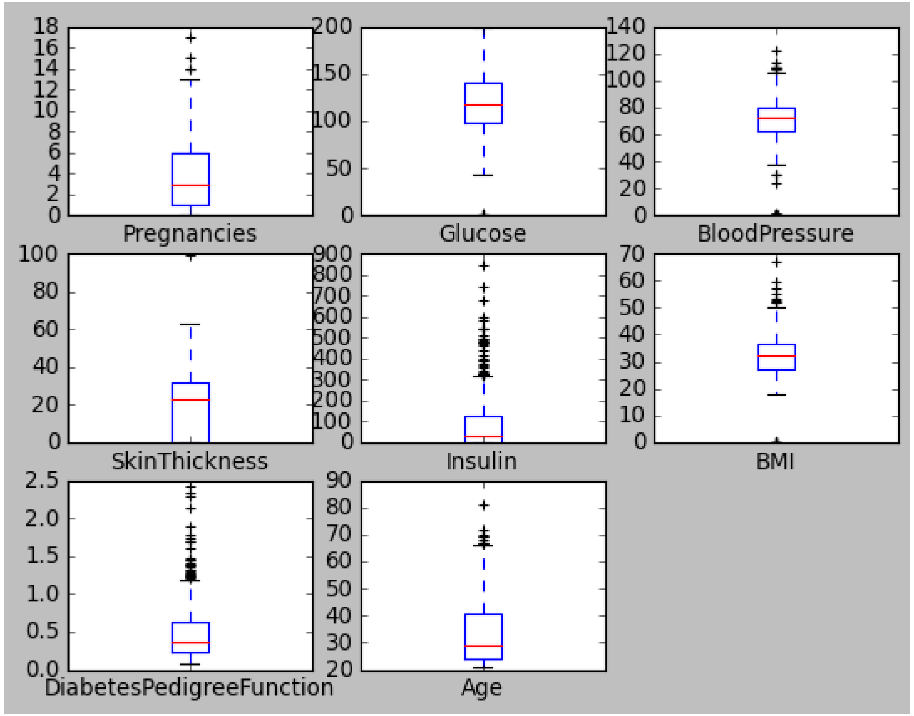
```
diabetes['Insulin'].plot(kind='density', subplots=True, sharex=False)
plt.show()
```



Matplotlib Examples

Box and Whisker Plots

```
diabetes.plot(kind='box', subplots=True, layout=(3,3), sharex=False, sharey=False)
pyplot.show()
```



Matplotlib Examples

Correlation Matrix Plot

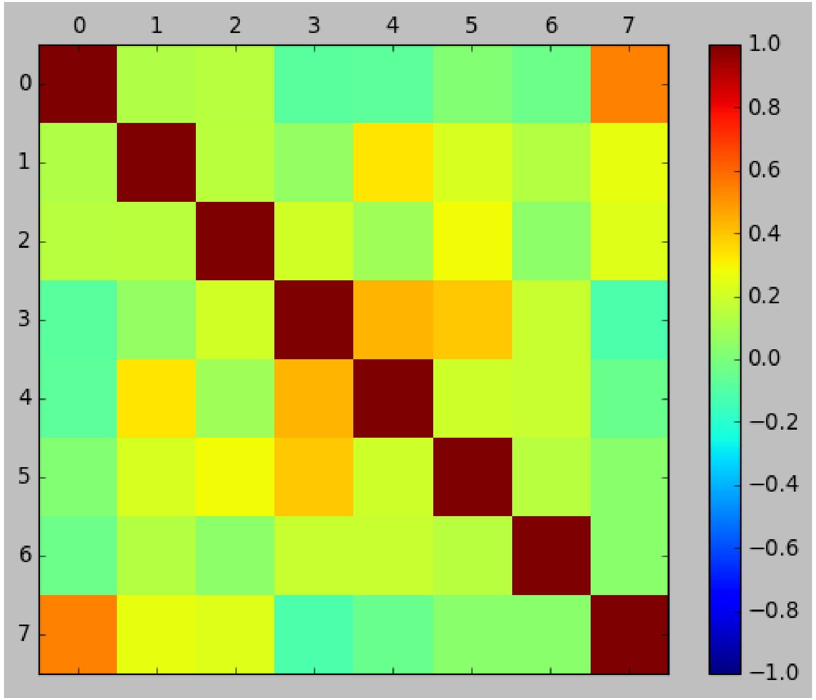
In [127]:
```
corr = diabetes.corr()
corr
```

Out[127]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
Pregnancies	1.000000	0.129459	0.141282	-0.081672	-0.073535	0.017683	-0.033523	0.544341
Glucose	0.129459	1.000000	0.152590	0.057328	0.331357	0.221071	0.137337	0.263514
BloodPressure	0.141282	0.152590	1.000000	0.207371	0.088933	0.281805	0.041265	0.239528
SkinThickness	-0.081672	0.057328	0.207371	1.000000	0.436783	0.392573	0.183928	-0.113970
Insulin	-0.073535	0.331357	0.088933	0.436783	1.000000	0.197859	0.185071	-0.042163
BMI	0.017683	0.221071	0.281805	0.392573	0.197859	1.000000	0.140647	0.036242
DiabetesPedigreeFunction	-0.033523	0.137337	0.041265	0.183928	0.185071	0.140647	1.000000	0.033561
Age	0.544341	0.263514	0.239528	-0.113970	-0.042163	0.036242	0.033561	1.000000

Matplotlib Examples

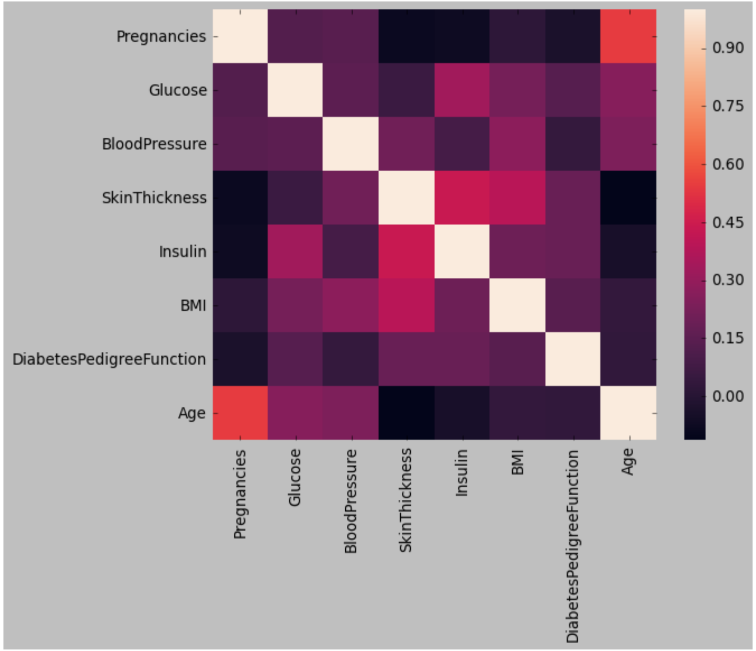
```
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
cax = ax.matshow(corr, vmin = -1, vmax = 1)
fig.colorbar(cax)
pyplot.show()
```



Matplotlib Examples

Seaborn Version

```
# Seaborn package does much better with correlation matrices
#
import seaborn as sns
sns.heatmap(corr)
pyplot.show()
```



Matplotlib Examples

Scatter Plot

In [130]:
```
plt.scatter(x = diabetes['Glucose'], y = diabetes['Insulin'])
plt.title('Glucose / Insulin Scatterplot')
plt.xlabel('Glucose')
plt.ylabel('Insulin')
```

Out[130]:
```
Text(0, 0.5, 'Insulin')
```