Week 1: Python Fundamentals

Seminar for

J.P.Morgan

2021 ADAPT Training

Athena Data & Analytics Python Training

July 26 - 29, 2021

THE MARQUEE GROUP

TheMarqueeGroup.com
info@themarqueegroup.com

The Marquee Group

Leaders in Financial Modeling Since 2002

- We believe that spreadsheet-based financial models are the most important decision-making tools in modern finance
- We have developed a framework and discipline for model design, structure and development that leads to best-in-class, user-friendly financial models
- We help finance professionals use this framework to turn their models into powerful communication tools that lead to better, more effective decisions

The Marquee Group Offering

TRAINING	CONSULTING	ACCREDITATION
✓ Instructors have real-world experience and a passion for teaching ✓ Topics include: Modeling, Valuation, Excel, VBA ✓ Courses are interactive ✓ Clients include banks, corporations, business schools and societies	✓ Services include: – Model Development – Model Re-builds – Model Reviews – Model Audits ✓ Clients include a wide range of companies in various industries	✓ Offered by the Financial Modeling Institute (FMI) ✓ The Marquee Group was one of the founders of the FMI ✓ FMI administers official exams for three levels of financial modeling certifications

THE MARQUEE GROUP

Stay in touch with us!

YouTube	Instagram	LinkedIn	Website	The BenchMarq
Watch free Excel and financial modeling videos	Follow our feed to keep up with Marquee news	Follow your instructor and our company page	Check out events, articles, and our blog	Subscribe to our newsletter for deals and announcements

The Financial Modeling Institute

SKILL VALIDATION
- Demonstrate financial modeling proficiency to employers and clients

PERSONAL DEVELOPMENT
- Earn certifications that are challenging and revered by the industry

CAREER FLEXIBILITY
- Verify skills that are globally relevant and respected in many industries

Level 1	Level 2	Level 3
✓ Foundational level of certification program ✓ Proficiency in building beginner-to-intermediate financial models ✓ Skills in design and comprehension of finance, business, accounting and Excel	✓ Attainable following successful completion of Level 1 ✓ Thorough understanding of real-world applications of financial modeling ✓ Demonstrated ability in advanced Excel, financial analysis, and financial modeling	✓ Highest level of accreditation achievable ✓ Expert in the end-to-end financial modeling value chain ✓ Respected thought leader, mentor, and contributor to financial modeling education

EACH LEVEL IS RECOGNIZED AS ITS OWN QUALIFICATION

Table of Contents

Using Python

Using *Python* – What is Python

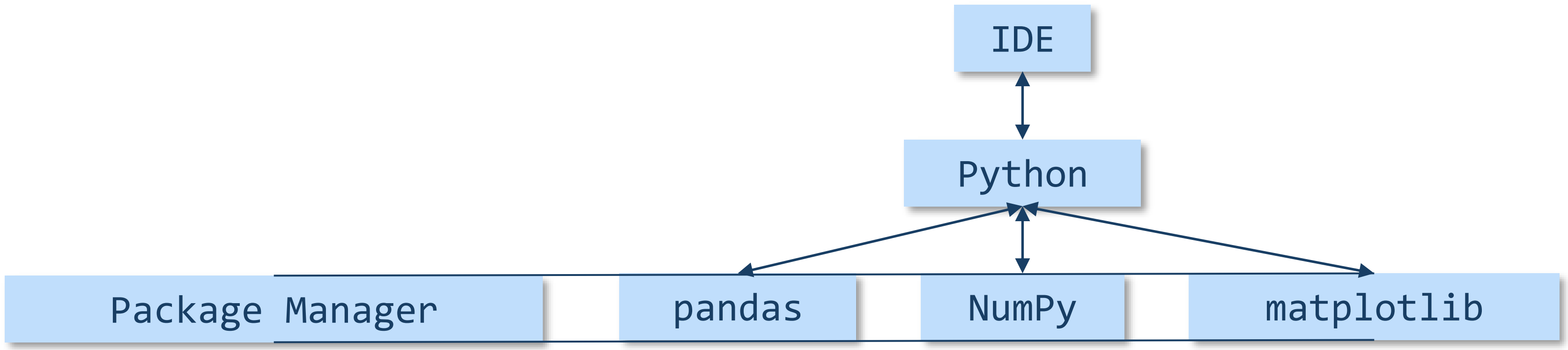
- What is Python?
  - Python is an interpreted object oriented programming language

$$\begin{tikzcd}\texttt{print(“Hello World”.upper())} \arrow[r] & \textit{Python Interpreter} \arrow[r] & \texttt{HELLO WORLD}\end{tikzcd}$$

  - Python allows for quick application development and is highly versatile; it can be used in applications ranging from machine control to data processing/AI
- Packages
  - The core functionality of Python can be extended by using packages
  - These packages provide an interface, sometimes referred to as API, to access commands that automate common routines
  - For instance, the *pandas* package adds the functionality to import a CSV file into an easy to manipulate format with only one command, whereas the core functionality would require a minimum of 3 lines of code and be less easy to handle

Using Python – Packages and IDE

- Package Manager
  - A package manager allows you to install/update/remove packages from your system
  - You can create specific environments if you want to add additional control to which packages and versions are installed
- Integrated Development Environment (IDE)
  - The IDE is where you will develop your Python application, similar to the VBA Environment for Excel
  - The main features of the Jupyter Notebook IDE are:
    - **Code-completion:** provides documentation for popular functions and auto completion
    - **Syntax-highlighting:** color-coded text to differentiate between user-generated variables and Python code
    - **Web-based:** ability to edit and execute code directly from browser
    - **Permanent Display:** allows outputs to be embedded in the notebook for future reference and sharing



Using Python – Distribution and IDE

- Anaconda on personal devices
  - We will be using the Anaconda Distribution of Python on personal devices which includes
    - Python 3.8
    - Conda package manager
    - Various IDEs including Jupyter Notebook
    - Variety of data science packages
- J.P. Morgan Athena on work devices
  - You will be using Athena as the development and runtime environment to run Python code on work devices
  - Athena handles package management and has an IDE called Athena Studio
  - Launching Jupyter Notebook:
    - https://go/jupyter then press the Start button

Using *Python* – Jupyter Notebook Shortcuts

Default Jupyter Notebook IDE Keyboard Shortcuts	
Run Cell	Ctrl + Enter
Run Cell and Advance to Next Cell	Shift + Enter
Edit Cell	Enter
Exit Edit Mode	Esc
Comment Line	CTRL + /
Delete Line	CTRL + D
Change Cell to Code	Y
Change Cell to Markdown	M
Insert a New Cell Above/Below	A / B
Copy/Paste/Cut Selected Cells	C / V / X
Delete Selected Cells	D, D
Hide/Unhide Output	O
Tooltips/Help Menu of Current Function	SHIFT + TAB

Using *Python* - Comments

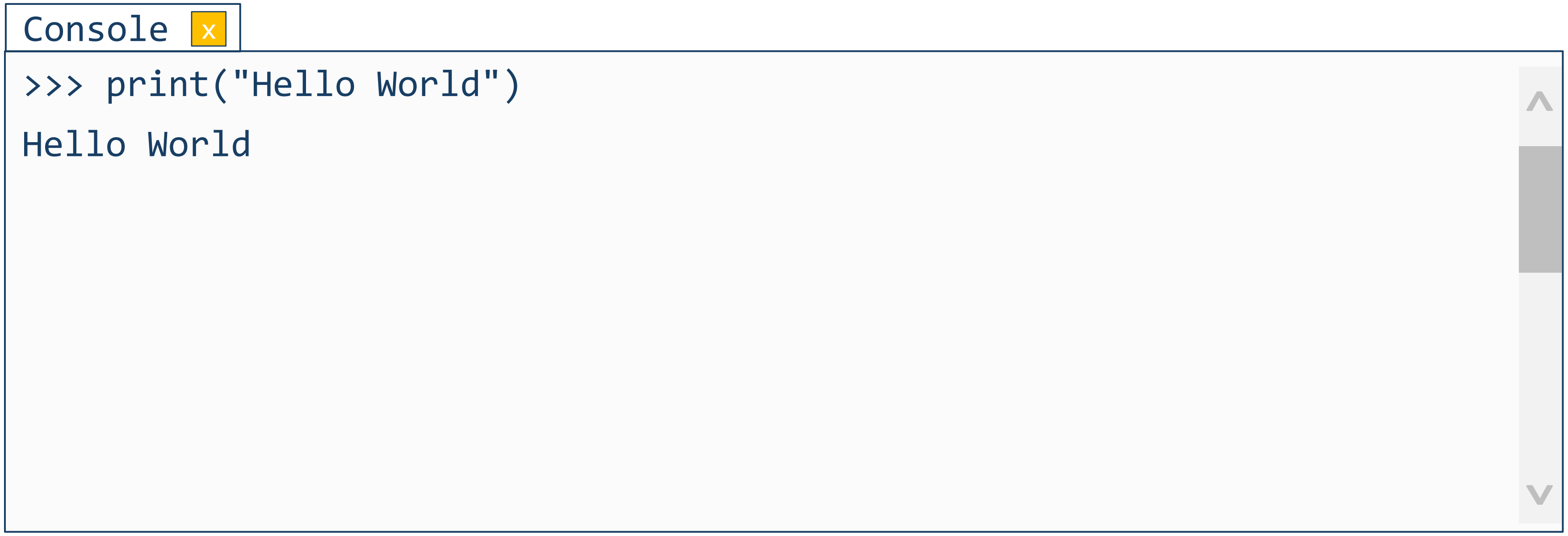
- Comments
  - Please comment your code! When you come back to something even a few weeks down the road it’s hard to remember what you were thinking at the time
  - Also helps with pseudo code, you can map out your code a bit more freely, keep track of variable assignments, and write notes so you don’t forget

Console x

```
# Single line comment
"""

Three quotes for a block, remember to close the block with another
three quotes
"""
```

Using *Python* – Hello World

- Print Hello World to console
  - Using print() command
  - Don’t forget to comment
  - Create a cell
- Console behavior
  - When running code line by line, the console will print outputs automatically
  - If running a series of lines/cells/file output is only guaranteed using the print() function
    - We will typically be running code line by line, see the appendix for details on running cells



```
>>> print("Hello World")
Hello World
```

Using *Python* – Before We Begin

- Before we begin, we will be learning a lot of functions whose purpose is not immediately obvious
- We are trying to develop the necessary toolkit to handle common issues typically encountered in data processing
  - Cleaning data, we can't use $100,000.00 as a number to do calculations
  - Merging data, ticker 'aapl' is not the same as 'AAPL' or 'Apple Inc.'
  - We need to take data from one source and import it into another, Python makes for a good "glue" language

Using *Python* – Data Types

Data Types

- We can group the Python Data Types into:
  - Numerical Values: Numbers (Integers, Floats); Dates
  - Strings: Text; Values with Symbols
- In Python there is no explicit variable declaration, rather you just assign a value to the variable and start using it
  - Python will interpret what you are trying to do and assign the appropriate data type
  - You can specify the data type by using the appropriate function
  - You can also change data types by “casting” the value using the function but be cautious that you have the desired result
    - Casting is when you transform one data type to another, like float to integer

Basic Python Data Types

Name	Function	Description	Example
Integer	int()	Non-Decimal Numbers	5
Float	float()	Decimal Numbers	5.3
String	str()	Stores either single or multiple characters	Hello World
List	list()	Stores several strings in a sequential list	['Cat', 'Dog', 'Bird']
Dictionary	dict()	A relational list Written as: {Key:Value}	{'AAPL':'Apple Inc'}
Tuple	tuple()	An immutable (unchangeable) list	('0.18','0.37','0.7')
Boolean	bool()	Logical type	True (1) or False (0)

Data Types

Operation	Symbol	Example
Addition	+	5 + 2 = 7
Subtraction	-	5 - 2 = 3
Multiplication	*	5 * 2 = 10
Division [1]	/	5 / 2 = 2.5
Modulus (remainder)	%	5 % 2 = 1
Exponent	**	5 ** 2 = 25
Floor Division [2]	//	5 // 2 = 2
(to negative infinity)		-5 // 2 = -3
Increment Variable	+=	a += 1
	-=	a -= 1

1. Note that regular division of numbers will always create a float date type
2. Floor division will create an integer data type if dividing by whole numbers

THE MARQUEE GROUP

Data Types – Strings

- String
  - This data type is used to store text type variables
  - Doesn’t matter if you use ‘My String’ or “My String” when assigning the string to the variable
  - The string is stored as an array in memory. This can be thought of as an indexed collection of characters

  - Python starts counting at 0

```
Hello World
```

H	e	l	l	o		W	o	r	l	d
0	1	2	3	4	5	6	7	8	9	10

Data Types – Strings

- We can concatenate (combine) strings together using the ‘+’ sign.
  - This only works with strings, if you want to concatenate a different data type to a string you must first cast it to a string using the str() method
- We can also create multiple copies of a value using the ‘*’ sign
  - Note that it will make exact copies, thus not adding spaces between occurrences.

```
>>>print("Hello " + 2) #will cause an error
>>>print("Hello " + str(2))
Hello 2

>>> print("Hello World" * 3)
Hello WorldHello WorldHello World
```

Data Types – String Methods

- In Python, some data types, such as Strings, have methods attached to them which differs from other programming languages
  - If you are familiar with programing you can think of the variable as an object which will execute a function on demand when called
  - When we get into other data types/structures in Python we will introduce properties that are also callable

String Methods

Method	Description
.strip()	Returns a string that has white spaces, or any character passed, removed from the beginning and end of a string.
.upper()	Returns a string that is all upper case.
.lower()	Returns a string that is all lower case.
.title()	Returns a string with the first letter of each word capitalized (Proper Case).
.replace()	Returns a string that has specified characters replaced by another character.
.split()	Returns a list that is comprised of strings divided up from the original string. Comma or Space are typical characters to split a string apart.
.count()	Returns the number of times a character set is found in the string.
.find()	Returns the position of a character set in the string.
.join()	Returns a string that is joined by the specified character.

Data Types – String Methods

- .strip([char])
  - By default, this method will remove ("strip") white spaces at the beginning and end of a string
  - The method accepts an optional character argument that it will look for instead of whitespaces
    - If you pass a '0' (zero), the method will remove leading and trailing zeros from the string
  - Useful for cleaning data while importing

```
>>> print('     Hello World  '.strip())
Hello World

>>> print('00 Hello World000'.strip('0'))
 Hello World

# Won't strip the leading space, need to strip twice
>>> print('00 Hello World000'.strip(str(0)).strip())
Hello World
```

Data Types – String Methods

- .upper()
  - Returns a string that is all in uppercase
  - Takes no arguments
- .lower()
  - Returns a string that is all in lowercase
- These are useful when comparing strings, such as a company name or ticker symbol
  - ‘A’ is not the same as ‘a’ to the computer
- There are other methods with similar behavior
  - .capitalize()
  - .title()

```
>>> print('Hello World'.upper())
HELLO WORLD
>>> print('Hello World'.lower())
hello world
>>> 'A'=='a'
False
>>> print('Hello World'.capitalize())
Hello world
>>> print('hELLO wORLD'.title())
Hello World
```

Data Types – String Methods

- .replace(find, replace, count)
  - Will search through the string for occurrences of the string you pass for ‘find’ and will replace them with the string you pass for ‘replace’
  - All occurrences are found by default unless you pass an integer to ‘count’ which will stop the search after that number of instances
  - Useful when cleaning data and removing punctuation, or if a number is stored as text with commas and currency symbols
    - Remember to cast as int/float if you need to use the number later in your program

```
>>> print('Hello World'.replace('l','L'))
HeLLo WorLd
>>> print('Hello World'.replace('l',''))
Heo Word
>>> print('Hello World'.replace('l','',2))
Heo World
```

Data Types – String Methods

- .split([char])
  - Returns a list, which can be stored as a single list variable or split into multiple variables each containing one portion of the string
  - The function will look for the character passed to the argument and create separate strings of what is contained to the left and right of the character until the end of the string
  - Useful when processing data from a "dirty" source
- .format(*any*)
  - Returns a string with the passed arguments formatted and inserted where placeholders are found
  - Placeholders are specified by {} in the main string. A specific key or index within the placeholder can reference values passed to the method

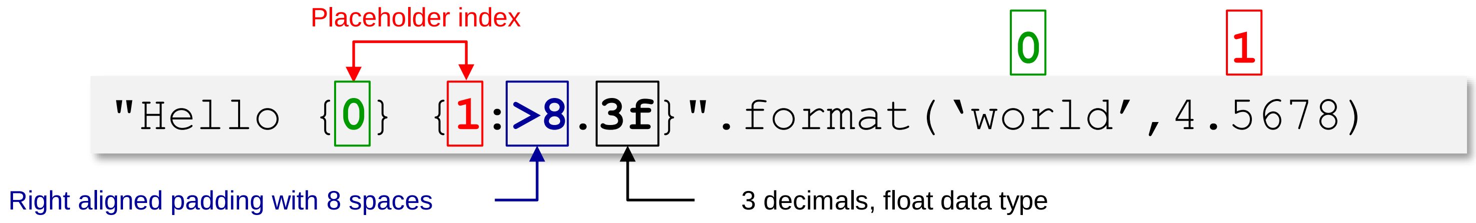
```
>>> print('Hello, World, Split'.split(','))
['Hello', ' World', ' Split']

>>> "Hello {}".format('world')
Hello world
```

Data Types – String Methods

- To format a number, with the placeholder use ‘:’ followed by the minimum number of spaces of padding before the decimal, then ‘.’ followed by the number of decimals, then the data type

String Format

Symbol	Description	Example	Argument	Output
d	Decimal integer	{:05d}	5	0 0 0 0 5
e or E	Exponential (scientific) notation, default 6 decimals	{:e}	1234	1 . 2 3 4 0 0 0e+03
f or F	Floating point number with fixed decimal places	{:8.3f}	1.234567	1. 2 3 4
%	Formats number as a percentage, will multiply by 100 and place % at end	{:%}	0.5	5 0%
<	Left align within the defined padded region	{:<8.3f}	1.234567	1. 2 3 4
^	Center align within the defined padded region	{:^8.3f}	1.234567	1. 2 3 4
>	Right align within the defined padded region	{:>8.3f}	1.234567	1. 2 3 4
=	Brings the negative sign to the left most position	{:=8.3f}	-1.234567	- 1. 2 3 5



Placeholder index
0
1
Right aligned padding with 8 spaces
3 decimals, float data type

```
"Hello {0} {1:>8.3f}".format(‘world’,4.5678)
```

```
>>> "Hello {0} {1:>8.3f}".format('world',4.5678)
Hello world    4.567
#There is an extra space before the 4 because of padding, and the
decimals after 3 are truncated (chopped off)
```

Data Types – String Methods

- .count([char])
  - Returns the number of times a character is found (counted) in a string
- .find([char])
  - Returns the first position (index) where the characters were found in a string
- These functions are useful in processing data from ‘dirty’ sources

```
>>> print('Hello World'.count('l'))
3
>>> print('Hello World'.count('L'))
0
>>> print('Hello World'.find('l'))
2
```

Data Types – String Methods

- .join([iterable])
  - Returns a string that is joined using the base string as the joiner
  - Used to convert a list data type (see next section) to a string
    - Some functions will return a list even if only one result is returned. This is done to make the function scalable
    - If a list only has one element, a string with one element is returned
    - If a list has multiple elements, then a long string of each element is returned with the joiner character between each
- len()
  - Returns the length of an array
    - When used on a string type it will return the number of characters. It might not be exactly the number of characters you see on screen if the string contains tabs, carriage returns etc.
    - When used on a list it will return the number of elements in the list
  - Useful when doing custom processing of data of unknown length, say for a counter

```
>>> '-'.join(i.strip() for i in ['Hello', ' World', ' Split'])
'Hello-World-Split'
>>> print(len('Hello World'))
11
```

Data Types – Lists

- List
  - Stores a series of variables in a one-dimensional array
  - Can be called as a group, or individually using the index
    - Remember that Python starts counting at zero
  - Useful for storing related data, or as a temporary storage before placing into a DataFrame (i.e. a two-dimensional table of data) or output to a file.
  - We can use the multiply to duplicate values in an array

```
['A','B','C'] # Simple list

['A',2,'C'] # Can mix data types

['A',['B','C'],'D'] # Can put a list in a list (nesting)

>>> mylist = [2] * 3
>>> print(mylist)
[2, 2, 2]
```

Data Types – Lists

- Lists and references
  - When assigning variables to a list, the value is copied so there is no link to the original variables
  - When assigning a list to another list, the references are maintained so any change to either matrix will be reflected to the other
    - A list can be thought of as a collection of pointers
    - This is very important if passing a list (or list type) to a function, that the function has the ability to modify the data passed to it. This only apply to lists, as variables are local

```
>>> a = 11
>>> b = 22
>>> c = 33
>>> mylist = [a, b, c]
>>> print(mylist)
[11, 22, 33]
>>> b = 44
>>> print(mylist)
[11, 22, 33]
```

```
>>> mylistA = [1,2,3]
>>> mylistB = mylistA
>>> print(mylistA, mylistB)
[1, 2, 3] [1, 2, 3]
>>> mylistA[1] = 4
>>> print(mylistA, mylistB)
[1, 4, 3] [1, 4, 3]
>>> mylistB[1] = 2
>>> print(mylistA, mylistB)
[1, 2, 3] [1, 2, 3]
```

Data Types – Lists

- Change a value by assigning a new value to the list index
- However, you cannot specify an index outside the range (length)
  - Logically to add ‘D’ to the end of the list, the index should be a 3, but this will generate an error because the 3 is outside of the index range
  - To add a value to the end of the list the .append() or .extend() methods are required
  - To add a value in the middle of the list, the .insert() method is required

```
>>> mylist = ['A','B','C']
>>> mylist[1] = 'D'
>>> print(mylist)
['A', 'D', 'C']

>>> mylist[3]='D'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Data Types – List Methods

List Methods

Method	**Description**
.append()	Adds the list or string to a new single element at the end of the list
.extend()	Adds the list or string to new elements the end of the list
.remove()	Searches for the passed string and deletes from the list if found
.pop()	Removes the element from the list based on the index
.insert()	Creates a new element at the specified index
.sort()	Sorts the list in either ascending or descending order
.reverse()	Will reverse the order of the list but does not sort the list

Data Types – List Methods

- .append([any])
  - Adds an item to the end of the list
    - If you pass a string, a string will be added to the end of the list as a single element
    - If you pass a list, a nested listed will be created at the end of the list
      - This can be accessed by using two indices
- .extend([list])
    - Adds the elements of the list passed as an argument to the end of the current list
    - This behaviour is different from .append() which only adds all the elements in the passed list into a single element at the end of the list

```
>>> mylist = ['A','B','C']
>>> mylist.append('D')
>>> print(mylist)
['A', 'B', 'C', 'D']
>>> mylist.append(['E','F'])
>>> print(mylist)
['A', 'B', 'C', 'D', ['E', 'F']]
>>> print(mylist[4][1])
'F'
```

```
>>> mylist = ['A','B','C']
>>> mylist.extend(['D','E'])
>>> print(mylist)
['A', 'B', 'C', 'D', 'E']
```

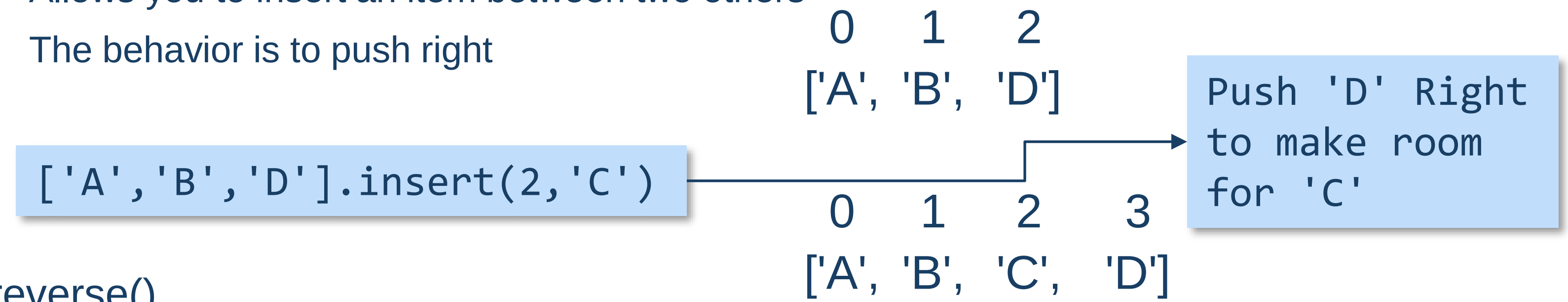
Data Types – List Methods

- .remove([any])
  - Will remove an item from the list based on the string passed.
  - Must match exactly and is case sensitive (‘D’ is not a ‘d’ and [‘D’] is not ‘D’)
- .pop(index)
  - Will remove an item from the list based on the index

```
>>> mylist.remove('D')
>>> print(mylist)
['A', 'B', 'C', 'E']
>>> mylist.pop(3)
'E'
>>> print(mylist)
['A', 'B', 'C']
>>> mylist = ['A','B','C','C']
>>> mylist.remove('C')
>>> print(mylist)
['A', 'B', 'C']
```

Data Types – List Methods

- .insert(*index, any)*
  - Allows you to insert an item between two others
  - The behavior is to push right



```
['A','B','D'].insert(2,'C')
```

```
 0    1    2
['A', 'B', 'D']
```

```
 0    1    2    3
['A', 'B', 'C', 'D']
```

```
Push 'D' Right
to make room
for 'C'
```

- .reverse()
  - Reverses the order of the list by index
- .sort(reverse=False, key=function)
  - Default is to sort ascending unless reverse is set to True
  - The key can specify a custom function for sorting such as len() to sort by length

```
>>> mylist = ['A','C']
>>> mylist.insert(1,'B')
>>> print(mylist)
['A', 'B', 'C']
```

```
>>> mylist.reverse()
>>> print(mylist)
['C', 'B', 'A']
>>> mylist.sort()
>>> print(mylist)
['A', 'B', 'C']
```

Data Types – Dictionaries

- Dictionary
  - Used for relational storage of data
  - For example, can be used to cross-reference ticker and company name
  - The entry is key:value
  - .keys() will return a list of all keys
  - .values() will return a list of all values
- Adding items to dictionary
  - Specify the key & value relationship
  - Same method will also change the value associated with a key
- Removing Items
  - .pop([char]) method specifying the key you wish to remove

```
stocks = {'AAPL':'Apple',
          'CAT':'Caterpillar',
          'MSFT':'Microsoft'}
>>> stocks.keys()
dict_keys(['AAPL', 'CAT', 'MSFT’])
>>> stocks.values()
dict_values(['Apple', 'Caterpillar',
'Microsoft'])
```

```
>>> stocks['BA'] = 'Boeing'
>>> print(stocks)
{'AAPL': 'Apple', 'CAT': 'Caterpillar',
'MSFT': 'Microsoft', 'BA': 'Boeing'}
```

Data Types – Tuples

- Tuple
  - Similar to a list but you cannot change the values
  - Useful if you have a set of constants that you don't want to accidentally change
    - Common error is = vs. ==, the first is for assignment the second is for checking if they are equal
  - Accessing values by index is same as list

```
>>> rates = (0.18,0.37,0.7)
>>> print(rates)
(0.18, 0.37, 0.7)
```

Using Python – Logic/Conditional Statements and Loops

Logic and Loops – Logical Operators

Logical Operators

Operation	Symbol	Description	Example
Equal	==	Checks if both sides are equal	a == b
Greater than	>	Checks if left side is greater than right	a > b
Less than	<	Checks if left side is less than right	a < b
Not Equal to	!=	Checks if left side is not equal to right	a != b
Not	not()	Flips the logical result	a not() b
Or	or	Checks two or more conditions, returns true if one of them is true	(a > 5) or (b < 5)
And	and	Check two or more conditions, return true only if they are all true	(a > 5) and (b < 5)

Logic and Loops – Booleans

- The logical data type is known as a Boolean
  - 0 is False
  - 1 is True
    - Most programming languages will interpret 0 as false, and all other numbers as true including negatives
    - This is an important distinction when using if statements or while loops, because you must ensure you have a valid False value to control the program

Logic and Loops – And/Or Logic

- To figure out what the result of using ‘and’/’or’ you can use the following table as a guide

Operator	Setup	A	B	Result
And	(A and B)	TRUE	TRUE	TRUE
		TRUE	FALSE	FALSE
		FALSE	TRUE	FALSE
		FALSE	FALSE	FALSE
Or	(A or B)	TRUE	TRUE	TRUE
		TRUE	FALSE	TRUE
		FALSE	TRUE	TRUE
		FALSE	FALSE	FALSE

- A simple rule is to consider the ‘and’ operator as multiplication and the ‘or’ operator as addition
  - Since True is 1 and False is 0, only way to get a True with the ‘and’ operator is if all are True
  - Same logic for ‘or’ operator, but now only way to get a False is if all are False (0 + 0 = 0)
- Applying the truth logic to the following:
  - A = 5
  - B = 6
  - ((A == 5) and (B < 6)) or ((A > 5) or (B == 6))
    - ((True) and (False)) or ((False) or (True))
    - (False) or (True)
    - True

Logic and Loops – If Statement

- *if* Statements
  - Only execute code if the condition is True
  - Treats False or 0 as not to enter (does not run code), and True or any other number as to enter (runs code)
  - Indents matter
    - The code that is indented is considered to be part of the if statement. Once the code is not indented, it is outside of the if statement and will be executed regardless of the conditional statement
  - Adding an *else* to the condition allows for code to be executed if the condition is not met

```
#Single Line if
>>> Result = 'Yes' if myvar == 5


#Multiline if with else
>>> if x-5:
...      #run this code if true
>>> else:
...      #run this code if false
```

Logic and Loops – If Statement

- *if – elif – else* statements
  - Adding elif (else if) will perform an additional check and execute the specified code if conditions are met
  - Can add as many *elif* statements as desired
  - The *elif* statements will be checked in order; once one *if* or *elif* condition is found True, the rest of the *elif* statements will not be checked

```
>>> if x-5:
...     print("Must be True")
>>> elif y==-5:
...     print("Y must be True")
>>> elif x==5:
...     print("X must be True")
>>> else:
...     print("Must be False")
```

Logic and Loops – For Loop

- For Loops
  - Syntax is *for (variable) in (iterable)*:
  - Allows the program to loop through any object that has multiple elements
    - Number range
    - Lists
- Number range
  - The range() function lets you quickly create a series of integers
  - The syntax is range(start=0, stop, step=1)
    - Calling range(10) will produce a series of ten numbers from 0 to 9. You can think of this as an open interval
    - Calling range (0,10,2) will produce a series of numbers counting by 2: 0, 2, 4, 6, 8

```
>>> for i in range(4):
...     print(i)
...
0
1
2
3
```

```
>>> for i in range(2, 10, 2):
...     print(i)
...
2
4
6
8
```

Logic and Loops – For Loop

- For loops can also iterate over items in a list
  - Each value in the list will be assigned to the variable before entering the loop
  - Ends when the end of the list is reached

```
>>> tickers = ['AAPL','CAT','MSFT']
>>> for ticker in tickers:
...     print(ticker)
...
AAPL
CAT
MSFT
```

Logic and Loops – While Loop

- While loops will terminate once a condition is reached
  - A condition that is possible to escape must be specified
  - If the condition can be skipped, it is possible to miss the escape clause
  - Using conditions based on floating point numbers create the chance of floating-point errors

```
>>> i=0
>>> while i < 5:
...     print(i)
...     i += 1 #same as i = i + 1
...
0
1
2
3
4
```

Using *Python* – Functions

Functions

- Functions
  - Useful when you need to repeat a set of instructions throughout your code
  - Variables (arguments) can be passed to the function to provide the necessary information to perform the task
  - A value/object can be returned and assigned to a variable if necessary

```
>>> def myfunction():
...     # code to run when called
...     return() # return a number if necessary

# call function and store return value in variable
>>> mynumber = myfunction()
```

Functions

- The function definition must be done before it is called
- Default values can be specified if desired
- Syntax
  - def functionName(arguments):
    some code indented
    return(variable)
  - If specifying variables with default values, they must be at the end of the arguments in the declaration

```
>>> from math import pi
>>>
>>> def fnArea(radius=1):
...     area = pi * (radius ** 2)
...     return(area)
...
>>> print(fnArea(2))
12.566370614359172
```

Functions – Lambda Functions

- Lambda functions work as regular functions but are typically "one liners"
  - Simple functions
  - Useful when manipulating datasets
- Syntax
  - *functionName* = **lambda** *arguments*: some operation code
  - Example:
    - x = lambda a, b : (a+1)*b

```
>>> x = lambda a, b : (a+1)*b
>>> print(x(1,2))
4

>>> fnCube = lambda num : num ** 3
>>> print(fnCube(2))
8
```

Using *Python* – NumPy

NumPy

- NumPy provides Python a powerful set of mathematical and statistical functions
  - Very similar to MATLAB
  - Will be primarily focusing on
    - Random Numbers
    - Statistical Operations
    - Matrix Operations

```
>>> import numpy as np
>>> print(np.__version__)
1.15.3
```

NumPy – Random Numbers

- Random Numbers
  - We can generate a single or series of random numbers using built in functions

Common Random Functions

Function	Description
.random.seed()	Sets a seed for the generator(s)
.random.randint()	Draws integers from the "discrete uniform" distribution
.random.random()	Draws floats from the "continuous uniform" distribution
.random.normal()	Draws samples from a normal (Gaussian) distribution ~ N(0,1)
.random.uniform()	Draws samples from a uniform distribution ~ U(0,1)
.random.rand()	Create an array of given size and populates with random samples from a uniform distribution over [0,1)
.random.randn()	Generates numbers from a standard normal distribution N(0,1)

NumPy – Random Number Functions

- .random.seed()
  - Computers generate a series of random number based on an algorithm
  - The pattern of random number is determined by its starting value, the seed
  - This is useful for replication of runs and auditing purposes
  - Don't set the seed too often
    - Preferably once at the beginning of the program; or,
    - At major sections in the code. This can be helpful if the code takes a while to run and you load interim calculations from a file

```
#sets a seed for the random number generator. Set once per execution.
>>> np.random.seed(42)
>>> print(np.random.rand(5))
[0.37454012 0.95071431 0.73199394 0.59865848 0.15601864]
>>> print(np.random.rand(5))
[0.15599452 0.05808361 0.86617615 0.60111501 0.70807258]
>>> np.random.seed(42) #notice how setting the seed again creates the same
sequence
>>> print(np.random.rand(5))
[0.37454012 0.95071431 0.73199394 0.59865848 0.15601864]
```

NumPy – Random Number Functions

- .random.randint(low, high=None, size, dtype)
  - Returns random integers from *low* (inclusive) to *high* (exclusive)
  - **Low**: specifies the lowest number that can be drawn (can be negative)
  - **High**: specifies the highest number that can be drawn
    - *low* must be specified, but *high* can be left blank where it defaults to the limit of the dtype
  - **Size**: specifies the number of draws if passing an integer, or shape of the output matrix if passing a tuple (m*n*k)

```
>>> print(np.random.randint(0,10, size=10)) #generates a random
integer from 0 to 10
[9 2 6 3 8 2 4 2 6 4]
```

NumPy – Random Number Functions

- .random.normal(loc=0, scale=1,size=1)
  - Provides draw(s) from a normal distribution
  - **Loc**: specifies the location (mean/average) of the normal distribution
  - **Scale**: specifies the standard deviation of the distribution
    - Remember that standard deviation is the square root of the variance
  - **Size**: specifies the number of draws if passing an integer, or shape of the output matrix if passing a tuple (m*n*k)
- .random.randn() can be called instead if want to generate numbers from a standard normal distribution N(0,1)

```
>>> print(np.random.normal()) #draws a random number from the normal distribution
-0.2933991463586617

>>> print(np.random.normal(5,2)) # can specify mean and standard dev
4.940322861280503

>>> print(np.random.normal(5,2, 10)) # can specify mean and standard dev and size
[5.19025157 6.32930869 4.71956301 4.9336132  3.50184695 3.44323599
 6.89768572 8.16170117 4.26365812 5.75112927]
```

NumPy – Random Number Functions

- .random.uniform(low=0,high=1,size=1)
  - Provides a draw or draws from a uniform distribution
  - **Low**: specifies the lower bound (inclusive)
  - **High**: specifies the upper bound (exclusive)
    - Can be thought of as an open interval [low,high)
  - **Size**: specifies the number of draws if passing an integer, or shape of the output matrix if passing a tuple (m*n*k)
- .random.rand() can be called instead if want to generate numbers from standard uniform [0,1)

```
>>> print(np.random.uniform(size=10)) #draws from uniform (0,1)
distribution
[0.37454012 0.95071431 0.73199394 0.59865848 0.15601864 0.15599452
 0.05808361 0.86617615 0.60111501 0.70807258]

>>> print(np.random.uniform(5,8,10)) #draws from uniform (5,8)
distribution
[5.06175348 7.90972956 7.49732792 5.63701733 5.5454749  5.55021353
 5.91272673 6.57426929 6.29583506 5.87368742]
```

NumPy – Statistical Functions

- NumPy has built in statistical functions
  - .mean() will return the average by rows or columns
  - .cov() will return the variance/covariance by rows or columns
- Other common math functions in NumPy
  - .log() finds the natural log
  - .exp() raises e to the number in the brackets (undoes log)
  - .sqrt() finds the square root
  - .linspace() will generate a series of numbers from min to max with the interval defined by the number of numbers desired
    - .linspace(start,stop, number of spaced numbers)
    - .linspace(0,10,100) will create 100 numbers from 0 to 10
  - .arrange() is similar to range, but lets you count by floats (decimals)

```
>>> print(np.mean(np.random.normal(5,2, 100)))
5.12884919781713
>>> print(np.cov(np.random.normal(5,2, 100)))
4.491550905995862

>>> print(np.linspace(1,4,13)) #makes 13 numbers from 1 to 4 inclusive
[1.   1.25 1.5  1.75 2.   2.25 2.5  2.75 3.   3.25 3.5  3.75 4.  ]

>>> print(np.arange(1,10.5,0.5))
[ 1.   1.5  2.   2.5  3.   3.5  4.   4.5  5.   5.5   6.   6.5  7.   7.5   8.   8.5  9.   9.5 10. ]
```

NumPy – Matrix

- To declare a matrix
  - Use the np.matrix function and pass the data as a list
  - Group rows using [ ], with commas separating values and commas separating rows
- To access the matrix
  - Use [row,col] after the name to specify the cell, row or column you which to retrieve
  - Remember that Python starts counting at zero
- Can also use built in functions to declare
  - Zero matrix
  - Identity matrix

```
>>>A = np.matrix([[1,2],[3,4]])
>>>print(A)
[[1 2]
[3 4]]
>>>print(A[0,1])
2
>>>print(A[1,])
[[3 4]]
```

```
>>>print(A[:,0])
[[1]
[3]]
>>> print(np.zeros([2,2]))
# prints a zeros matrix
[[0. 0.]
[0. 0.]]
>>>print(np.identity(2))
# prints an identity matrix square
[[1. 0.]
[0. 1.]]
```

NumPy – Matrix Functions

- Matrix Multiplication
  - Python will interpret A*b as matrix multiplication. Recommend using the np.matmul() function or @ because A*b is not always clear on the intention
- Quick review of matrix multiplication
  - The number of rows is m, and columns is n giving dimension of $m \times n$
  - Multiplying matrices follow the rule "row by column", so the matrix on the left has to have the same number of rows as the matrix on the right has columns.
  - Repeat for each row on the left by each column on the right
  - Will have an $m_L \times m_R$ matrix left

$$\begin{bmatrix} \boxed{1 \quad 2 \quad 3} \\ 4 \quad 5 \quad 6 \end{bmatrix} \begin{bmatrix} \boxed{\begin{matrix} 7 \\ 9 \\ 11 \end{matrix}} & \begin{matrix} 8 \\ 10 \\ 12 \end{matrix} \end{bmatrix} = \begin{bmatrix} \boxed{58} & 64 \\ 139 & 154 \end{bmatrix}$$

$$(1*7)+(2*9)+(3*11)=58$$

```
>>>A = np.matrix([[1,2],[3,4]])
>>>b = np.matrix([[5,6],[7,8]])
>>>print(A*b) #notice that this does matrix multiplication
[[19 22]
[43 50]]
>>>print(np.matmul(A,b)) #same result, I prefer this method because it’s not
ambiguous
[[19 22]
[43 50]]
```

NumPy – Matrix Functions

- Matrix Division
  - .divide or using a “/” between variables does element wise division
  - Find the inverse using .linalg.inv() and using the inverted matrix to divide through matrix multiplication
    - To check the inverse is correct, we can multiply by the original matrix and see if we get the identity matrix as the result

```
>>> print(np.divide(A,b)) #this is elemental division
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
>>> print(A/b)
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
>>> print(np.matmul(A,np.linalg.inv(b))) #this is  matrix division
[[ 3. -2.]
 [ 2. -1.]]
>>> print(np.matmul(A,np.linalg.inv(A))) #should be identity matrix
[[1.00000000e+00 1.11022302e-16]
 [0.00000000e+00 1.00000000e+00]]
```

NumPy – Matrix Functions

- Element Wise Matrix Division
  - By scalar

$$\begin{bmatrix} 3 & 9 \\ 12 & 15 \end{bmatrix} \div 3 = \begin{bmatrix} 1 & 3 \\ 4 & 5 \end{bmatrix}$$

  - By matrix

$$\begin{bmatrix} 3 & 9 \\ 12 & 15 \end{bmatrix} \oslash \begin{bmatrix} 3 & 3 \\ 2 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 6 & 3 \end{bmatrix}$$

- Matrix Division
  - Need to find the inverse of the matrix first, where

$$AA^{-1} = I$$

  - Division of $Ax = B$ is

$$A^{-1}Ax = A^{-1}B$$
$$Ix = A^{-1}B$$
$$x = A^{-1}B$$

Using *pandas*

What is *pandas*

- Stands for Python Data Analysis Library
- Allows Python to handle cross-sectional or time-series data easier
  - Stores data in “DataFrames”
  - Allows for labelling of variables
  - Provides built-in functionality to handle typical missing data, reshaping, merging, sub-setting and other operations
- Official Documentation
  - https://pandas.pydata.org/docs/
- Packages need to be imported into Python
- It is common practice to alias pandas as pd
- Using the alias will allow you to refer to the pandas package as pd
- It's common to refer to a generic DataFrame as df

```
>>>import pandas as pd
>>>pd.__version__
'0.23.4'
```

Using *pandas*

- Tidy Data Concepts
  - Data frames have rows and columns
  - Each line is one (indexed) observation
  - Each column is one (named) variable
  - Each value is assigned to a specific row & column
- This is an important concept to follow because we don't get to view the data like in Excel.
  - Some IDEs have variable explorers which will visualize the data like Excel, but they are slow especially for large datasets.

(index)	Date	Ticker	Close	Bk_mrkt	Summer_Dummy
1	2018-01-01	AAPL	$123	50	0
2	2018-01-02	AAPL	$123	50	0
3	2018-01-03	APPL	$125	50	0

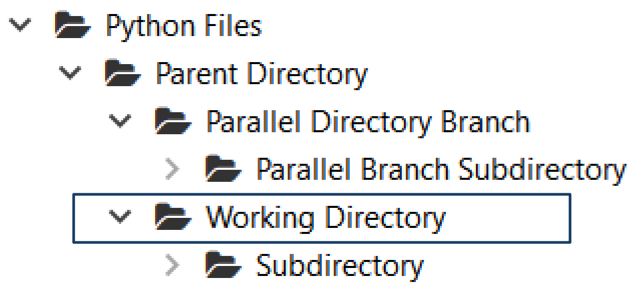
Importing Data with *pandas*

Importing Data – Reading Files

- Reading a csv file into a DataFrame
  - .read_csv(filepath, sep, header, nrows, chunksize, …)
  - Syntax:
    - **Filepath** (str)
      - Location relative to script or project directory
    - **Sep** (str) – Default , (comma)
      - Separator or Delimiter
      - Used in the file to separate columns
    - **Header** (int) – Default 0 (zero)
      - Row number that has the column names
    - **Nrows** (int) – Default None
      - If specified, will limit the number of rows to read, useful for "big data"
    - **Chunksize** (int) – Default None
      - If specified, the variable will become a generator where each time it is called the next chunk of data is read into memory.

Importing Data – Reading Files

- Referencing a filename in Python



Referencing a filename in Python

Filepath	Location of file
'filename.csv'	File is in the Working Directory
'Subdirectory/filename.csv'	File is in a subdirectory in the working directory
'../filename.csv'	File is in the parent directory of the working directory
'../Parallel Directory Branch/filename.csv'	File is in a subdirectory of the parent directory
'../Parallel Directory Branch/Subdirectory/filename.csv'	File is in a nested subdirectory of the parent directory

Importing Data – Useful Methods

- To get details and view the data in a pandas DataFrame, the following methods are helpful

Method	Description
.shape	Lists the number of rows and columns.
.head()	Prints the first 5 lines by default. Passing a number between the () will change the default.
.tail()	Prints the last 5 lines.
.columns	Prints the name of columns.
.info()	Provides index range, column (variable) names, datatypes, and number of observations.

```
>>> sp500 = pd.read_csv('sp500.csv')
>>> print(sp500.head())
         Date         Open         High          Low        Close    Adj Close      Volume
0  2013-09-30  1687.260010  1687.260010  1674.989990  1681.550049  1681.550049  3308630000
1  2013-10-01  1682.410034  1696.550049  1682.069946  1695.000000  1695.000000  3238690000
2  2013-10-02  1691.900024  1693.869995  1680.339966  1693.869995  1693.869995  3148600000
3  2013-10-03  1692.349976  1692.349976  1670.359985  1678.660034  1678.660034  3279650000
4  2013-10-04  1678.790039  1691.939941  1677.329956  1690.500000  1690.500000  2880270000
```

Importing Data – info Method

- .info() provides
  - RangeIndex which is an integer that references the rows starting at 0 for the first row. This index can be used to access the individual rows to extract data
  - The column names and data types are then listed in order of appearance in the DataFrame
  - The non-null count provides a count of total observations and is quick way to see if there is missing data

```
>>> print(sp500.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1260 entries, 0 to 1259
Data columns (total 7 columns):
Date         1260 non-null object
Open         1260 non-null float64
High         1260 non-null float64
Low          1260 non-null float64
Close        1260 non-null float64
Adj Close    1260 non-null float64
Volume       1260 non-null int64
dtypes: float64(5), int64(1), object(1)
memory usage: 69.0+ KB
```

Importing Data – Dates as Index

- If the dataset has dates for each observation, it is common to use them as the index
  - Makes it easier to access date ranges for data, or to filter by month/year
- First convert the Date column from object to a DateTime/DateTimeIndex
  - DateTime is a data type referenced by the pandas package
  - Default expected format “YYYY-mm-dd”
  - Follows POSIX (Unix) time standard
- Next, assign the Date column to the index and drop the column from the DataFrame
  - Dropping the column will delete it from the DataFrame
  - Setting *inplace* to True will modify the original DataFrame stored in memory

```
# Template df.colname = pandas.function()
>>>sp500.Date = pd.to_datetime(sp500['Date’])
>>>sp500.set_index(['Date'], drop = True, inplace = True)

# setting inplace=True is the same as
>>> sp500 = sp500.set_index(['Date'], drop = True)
```

Importing Data – Dates as Index

- Examining .info() after changing the index
  - The index now is a DatetimeIndex type
  - The Date column is no longer part of the DataFrame

```
         Date         Open         High
0  2013-09-30  1687.260010  1687.260010
1  2013-10-01  1682.410034  1696.550049
2  2013-10-02  1691.900024  1693.869995
3  2013-10-03  1692.349976  1692.349976
4  2013-10-04  1678.790039  1691.939941
```

```
                   Open         High
Date
2013-09-30  1687.260010  1687.260010
2013-10-01  1682.410034  1696.550049
2013-10-02  1691.900024  1693.869995
2013-10-03  1692.349976  1692.349976
2013-10-04  1678.790039  1691.939941
```

```
>>> print(sp500.info())
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1260 entries, 2013-09-30 to 2018-09-28
Data columns (total 6 columns):
Open         1260 non-null float64
High         1260 non-null float64
Low          1260 non-null float64
Close        1260 non-null float64
Adj Close    1260 non-null float64
Volume       1260 non-null int64
dtypes: float64(5), int64(1)
memory usage: 68.9 KB
```

Date column is no longer a data column

Importing Data – Sorting Data

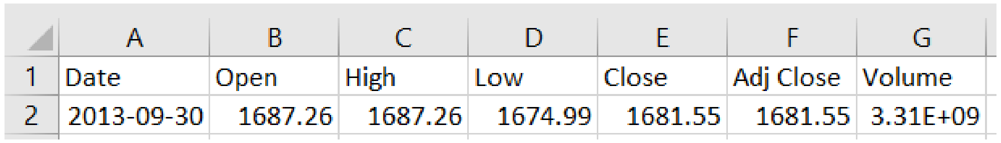
- We can use two methods to sort data imported into a DataFrame
- .sort_values(by, axis=0, ascending=True, inplace=False)
  - **By**: string or list of string in the order which you want to sort your data
  - **Axis**: 0 sorts the rows; 1 sorts the columns
- .sort_index(ascending=True, inplace=False)
  - **Ascending**: Order to sort the data
  - **Inplace**: set to True to modify the original DataFrame
  - Typically leave the other parameters of function as default

```
>>> sp500.sort_values(['Close'], inplace=True)
>>> sp500.loc[:'2013-10-08']
                   Open         High  ...    Adj Close      Volume
Date                                  ...
2013-10-08  1676.219971  1676.790039  ...  1655.449951  3569230000
2013-10-07  1687.150024  1687.150024  ...  1676.119995  2678490000
2013-10-03  1692.349976  1692.349976  ...  1678.660034  3279650000
2013-09-30  1687.260010  1687.260010  ...  1681.550049  3308630000
2013-10-04  1678.790039  1691.939941  ...  1690.500000  2880270000
2013-10-02  1691.900024  1693.869995  ...  1693.869995  3148600000
2013-10-01  1682.410034  1696.550049  ...  1695.000000  3238690000

>>> sp500.sort_index(inplace=True) # Data is sorted by date again
```

Accessing/Slicing Data with *pandas*

Accessing/Slicing – Rows

- There are many ways to access the data stored in the DataFrame
  - Some depend on operation; others will seem redundant (legacy)
  - Going to focus on the logic in accessing data
- Two main methods:
  - .iloc – integer location
  - .loc – label location (using index)



	A	B	C	D	E	F	G
1	Date	Open	High	Low	Close	Adj Close	Volume
2	2013-09-30	1687.26	1687.26	1674.99	1681.55	1681.55	3.31E+09

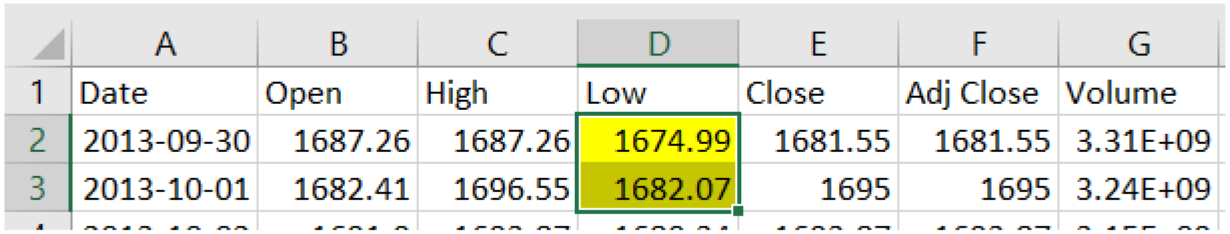
```
>>> print(sp500.iloc[0]) #integer location
Open          1.687260e+03
High          1.687260e+03
Low           1.674990e+03
Close         1.681550e+03
Adj Close     1.681550e+03
Volume        3.308630e+09
Name: 2013-09-30 00:00:00, dtype: float64
>>> print(sp500.loc['20130930']) #label location
Open          1.687260e+03
High          1.687260e+03
Low           1.674990e+03
Close         1.681550e+03
Adj Close     1.681550e+03
Volume        3.308630e+09
Name: 2013-09-30 00:00:00, dtype: float64
```

Name is the index of the record

Don't need dashes for date

Accessing/Slicing – Slicing

- Slicing is selecting a range of rows/columns
  - df.iloc[row start:row end, col start:col end]
  - df.iloc[row,col] will only draw a value from the specific cell
    - Similar to excel 'D2' would be fourth column second row
  - Python uses an open interval (up to not including)
  - Think of it as counting [start:# of rows/cols]
  - A blank before the : will default to a start at first row/column
  - A blank after the : will default to end at the last row/column



	A	B	C	D	E	F	G
1	Date	Open	High	Low	Close	Adj Close	Volume
2	2013-09-30	1687.26	1687.26	1674.99	1681.55	1681.55	3.31E+09
3	2013-10-01	1682.41	1696.55	1682.07	1695	1695	3.24E+09

```
# in words df.iloc[start at row 0:count 2 rows, col 2 only]
>>> print(sp500.iloc[:2,2])
Date
2013-09-30    1674.989990
2013-10-01    1682.069946
Name: Low, dtype: float64
```

Up to not including

Accessing/Slicing – Columns

- Extracting a specific column
  - Can access a specific column of a DataFrame either by
    - df.colname or df['colname']
  - Assign to a new variable as
    - Newvar = df['colname']

```
>>> close500 = sp500['Close']
>>> print(close500.head()) #you can see that this made a copy of the
DataFrame, keeping the index
Date
2013-09-30    1681.550049
2013-10-01    1695.000000
2013-10-02    1693.869995
2013-10-03    1678.660034
2013-10-04    1690.500000
Name: Close, dtype: float64
```

Accessing/Slicing – Filtering Data

- Filtering Data based on Condition
  - We can use a logical condition or multiple conditions to select data
  - The rows that are kept are True (1) and the ones that are dropped are False (0)
  - We can combine the filter with a column selection as well

```
>>> print(sp500[sp500['Open'] > sp500['Close']].head())
                   Open         High          Low        Close    Adj Close      Volume
Date
2013-09-30  1687.260010  1687.260010  1674.989990  1681.550049  1681.550049  3308630000
2013-10-03  1692.349976  1692.349976  1670.359985  1678.660034  1678.660034  3279650000
2013-10-07  1687.150024  1687.150024  1674.699951  1676.119995  1676.119995  2678490000
2013-10-08  1676.219971  1676.790039  1655.030029  1655.449951  1655.449951  3569230000
2013-10-09  1656.989990  1662.469971  1646.469971  1656.400024  1656.400024  3577840000

>>> print(sp500[sp500['Open'] > sp500['Close']][['Open','Close']].head())
                   Open        Close
Date
2013-09-30  1687.260010  1681.550049
2013-10-03  1692.349976  1678.660034
2013-10-07  1687.150024  1676.119995
2013-10-08  1676.219971  1655.449951
2013-10-09  1656.989990  1656.400024
```

Accessing/Slicing – Copying Warning

- SettingwithCopyWarning
  - You will get this warning if you try to chain assignments such as:
    rtn_2008 = df[df.year == 2008]['returns']
  - The reason is that pandas cannot guarantee if it will return a copy of the data, or a view of the data as it is dependant on how the data is mapped in memory
    - A copy is a stand alone DataFrame completely detached from the original DataFrame where the information was copied from; a change to one DataFrame does not effect the other
    - A view is still accessible as a standalone DataFrame, but is still referenced to the original DataFrame
    - If you modify a view, it will modify the original data. This is similar to the relative referencing with lists
  - This warning will also occur doing mundane tasks, like .shift()
    - This is important to consider if passing a slice of a DataFrame to a function
- There are two methods to work around this problem
  - Using the .copy() to force pandas to create a copy and not a view
    - Typically used when passing data to a function
  - Removing the chained index by slicing as df.loc[:, (cols)]
- This is important when doing an assignment based on a condition

Manipulating Data with *pandas*

Manipulating Data – Calculations

- To create a new column (variable) in the DataFrame
  - df['newcol'] = df['col'] * 2
    - Will create a new column that is two times the previous
  - In the code below we are taking the natural logarithm of the index close
    - A lot of functions can handle DataFrame or array/list type objects
    - Functions that cannot handle DataFrame objects can still be used using the .apply() method

```
# Template df['new col(var)'] = code/operation to generate values
>>> sp500['logClose'] = np.log(sp500['Close'])
>>> print(sp500.head())
                   Open         High    ...       Volume  logClose
Date                                    ...
2013-09-30  1687.260010  1687.260010    ...   3308630000  7.427471
2013-10-01  1682.410034  1696.550049    ...   3238690000  7.435438
2013-10-02  1691.900024  1693.869995    ...   3148600000  7.434771
2013-10-03  1692.349976  1692.349976    ...   3279650000  7.425751
2013-10-04  1678.790039  1691.939941    ...   2880270000  7.432780
```

Manipulating Data – Resampling

- *pandas* has a built-in function for resampling data frequency
  - Rule based, can be specific to each column stored in a dictionary.

Common Resampling Rules

Name	Description of Action
first	The first valid value
last	The last valid value
max	The maximum value over the resampling range
min	The minimum value over the resampling range
sum	The sum of all values of the resampling range
mean	The average of all values over the resampling range
median	The median (center) value over the resampling range

Common Resampling Rules

Name	Description of Action
std	The standard deviation of the values over the resampling range
count	The number of non-null data points over the resampling range
nunique	The number of unique values over the resampling range
bfill	When down sampling (going to a higher frequency, weeks to days) takes the later value and fills backwards for missing values
ffill	When down sampling (going to a higher frequency, weeks to days) takes the current value and fills forward for missing values

```
# Example transforming daily to monthly returns
# ohlc is the open high low close resampling rules (added in volume)
>>> ohlc_rule = {'Open':'first', 'High':'max', 'Low':'min',
                          'Close':'last', 'Volume':'sum', 'Adj Close':'last', 'logRtns':'sum'}
>>> mon500 = sp500.resample('M').agg(ohlc_rule)
>>> mon500.head()
                   Open         High     ...          Volume    Adj Close
Date                                     ...
2013-09-30  1687.260010  1687.260010     ...      3308630000  1681.550049
2013-10-31  1682.410034  1775.219971     ...     76647400000  1756.540039
2013-11-30  1758.699951  1813.550049     ...     63628190000  1805.810059
2013-12-31  1806.550049  1849.439941     ...     64958820000  1848.359985
2014-01-31  1845.859985  1850.839966     ...     75871910000  1782.589966
```

Manipulating Data – apply Function

- .apply(func, axis=0, args=())
  - Applies a function (custom or standard) across a row or column of the data
  - **Func**: Specify the function name, no parenthesis ()
  - **Axis**:
    - 0 to apply by column
    - 1 to apply by row (use this if you are working across columns)
  - **Args** (tuple): To pass additional arguments to the function
- Using .apply()
  - If you are looking at summary statistics, following Tidy Data principles you will be working along columns
  - If you are performing a calculation, you are working along rows

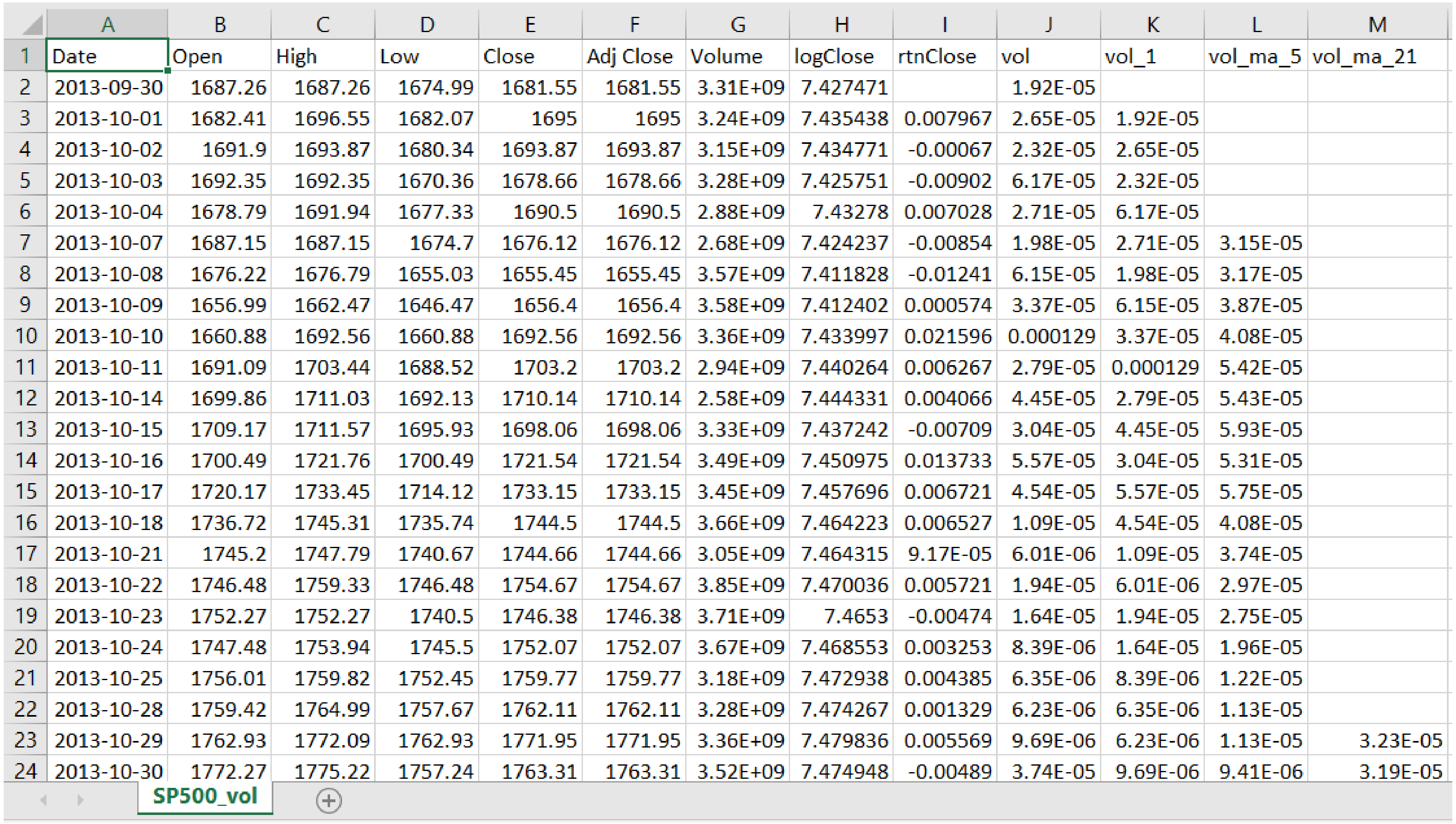
```
#Apply function
#Used to take a row or group of data and apply a function to them
#Mean of each Column
>>> sp500.apply(np.mean, axis=0)

#Calculates an estimate of vol using Parkinson’s Volatility
>>> sp500['vol'] = sp500.apply(lambda x:
          ((np.log(x['High'])- np.log(x['Low']))**2)/(4*np.log(2)), axis=1)
```

Manipulating Data – rolling Function

- Pandas has some built in functions, so you don't always have to use apply
  - Rolling will create a rolling window where you can apply sum, count, mean and other standard statistical functions
  - Shift will lag a variable by the specified amount
  - Useful for times series analysis
- Every time we create a lagged variable, we lose an observation for analysis
  - If you require 60 observations (5 years of monthly data), each lag will require an additional month to keep 60 observations

```
#Shift
>>> sp500['vol_1'] = sp500['vol'].shift(1)

#Moving Average
>>> sp500['vol_ma_5'] = sp500['vol_1'].rolling(window = 5,
                                              min_periods=5).mean()
>>> sp500['vol_ma_21'] = sp500['vol_1'].rolling(window = 21,
                                              min_periods=21).mean()
```

Manipulating Data – rolling Function

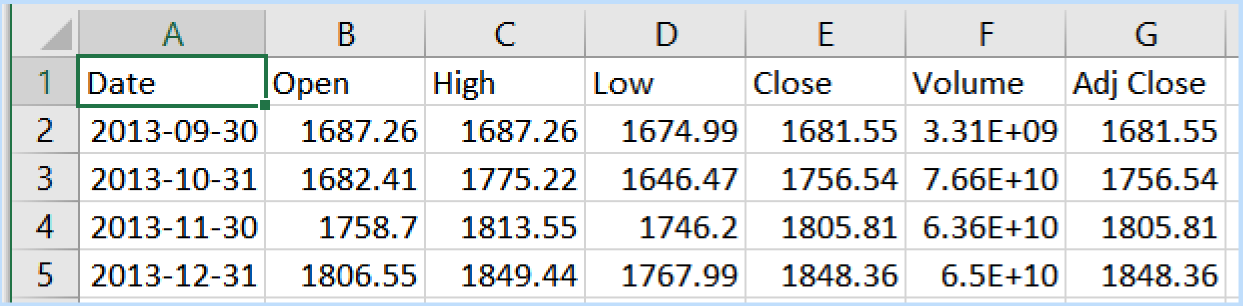


	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Date	Open	High	Low	Close	Adj Close	Volume	logClose	rtnClose	vol	vol_1	vol_ma_5	vol_ma_21
2	2013-09-30	1687.26	1687.26	1674.99	1681.55	1681.55	3.31E+09	7.427471		1.92E-05			
3	2013-10-01	1682.41	1696.55	1682.07	1695	1695	3.24E+09	7.435438	0.007967	2.65E-05	1.92E-05		
4	2013-10-02	1691.9	1693.87	1680.34	1693.87	1693.87	3.15E+09	7.434771	-0.00067	2.32E-05	2.65E-05		
5	2013-10-03	1692.35	1692.35	1670.36	1678.66	1678.66	3.28E+09	7.425751	-0.00902	6.17E-05	2.32E-05		
6	2013-10-04	1678.79	1691.94	1677.33	1690.5	1690.5	2.88E+09	7.43278	0.007028	2.71E-05	6.17E-05		
7	2013-10-07	1687.15	1687.15	1674.7	1676.12	1676.12	2.68E+09	7.424237	-0.00854	1.98E-05	2.71E-05	3.15E-05	
8	2013-10-08	1676.22	1676.79	1655.03	1655.45	1655.45	3.57E+09	7.411828	-0.01241	6.15E-05	1.98E-05	3.17E-05	
9	2013-10-09	1656.99	1662.47	1646.47	1656.4	1656.4	3.58E+09	7.412402	0.000574	3.37E-05	6.15E-05	3.87E-05	
10	2013-10-10	1660.88	1692.56	1660.88	1692.56	1692.56	3.36E+09	7.433997	0.021596	0.000129	3.37E-05	4.08E-05	
11	2013-10-11	1691.09	1703.44	1688.52	1703.2	1703.2	2.94E+09	7.440264	0.006267	2.79E-05	0.000129	5.42E-05	
12	2013-10-14	1699.86	1711.03	1692.13	1710.14	1710.14	2.58E+09	7.444331	0.004066	4.45E-05	2.79E-05	5.43E-05	
13	2013-10-15	1709.17	1711.57	1695.93	1698.06	1698.06	3.33E+09	7.437242	-0.00709	3.04E-05	4.45E-05	5.93E-05	
14	2013-10-16	1700.49	1721.76	1700.49	1721.54	1721.54	3.49E+09	7.450975	0.013733	5.57E-05	3.04E-05	5.31E-05	
15	2013-10-17	1720.17	1733.45	1714.12	1733.15	1733.15	3.45E+09	7.457696	0.006721	4.54E-05	5.57E-05	5.75E-05	
16	2013-10-18	1736.72	1745.31	1735.74	1744.5	1744.5	3.66E+09	7.464223	0.006527	1.09E-05	4.54E-05	4.08E-05	
17	2013-10-21	1745.2	1747.79	1740.67	1744.66	1744.66	3.05E+09	7.464315	9.17E-05	6.01E-06	1.09E-05	3.74E-05	
18	2013-10-22	1746.48	1759.33	1746.48	1754.67	1754.67	3.85E+09	7.470036	0.005721	1.94E-05	6.01E-06	2.97E-05	
19	2013-10-23	1752.27	1752.27	1740.5	1746.38	1746.38	3.71E+09	7.4653	-0.00474	1.64E-05	1.94E-05	2.75E-05	
20	2013-10-24	1747.48	1753.94	1745.5	1752.07	1752.07	3.67E+09	7.468553	0.003253	8.39E-06	1.64E-05	1.96E-05	
21	2013-10-25	1756.01	1759.82	1752.45	1759.77	1759.77	3.18E+09	7.472938	0.004385	6.35E-06	8.39E-06	1.22E-05	
22	2013-10-28	1759.42	1764.99	1757.67	1762.11	1762.11	3.28E+09	7.474267	0.001329	6.23E-06	6.35E-06	1.13E-05	
23	2013-10-29	1762.93	1772.09	1762.93	1771.95	1771.95	3.36E+09	7.479836	0.005569	9.69E-06	6.23E-06	1.13E-05	3.23E-05
24	2013-10-30	1772.27	1775.22	1757.24	1763.31	1763.31	3.52E+09	7.474948	-0.00489	3.74E-05	9.69E-06	9.41E-06	3.19E-05

Manipulating Data – Output Data

- Pandas has built in methods to output the data stored in a DataFrame to several file types, most common is to use CSV
- .to_csv(path, sep=',', columns=None, header=True, index=True)
  - **Path:** file path with file name. Could also be a variable with the path string (useful in loops)
  - **Columns:** specify which columns to output to the file
  - **Sep:** specify the delimiter, default is comma (,)
  - **Header:** Default is to include the header of the DataFrame
  - **Index:** Default is to include the index, but it can be meaningless if it is the integer index, useful if the index was set to dates

```
>>> mon500.to_csv('../StockData/SP500_mon.csv')
```



	A	B	C	D	E	F	G
1	Date	Open	High	Low	Close	Volume	Adj Close
2	2013-09-30	1687.26	1687.26	1674.99	1681.55	3.31E+09	1681.55
3	2013-10-31	1682.41	1775.22	1646.47	1756.54	7.66E+10	1756.54
4	2013-11-30	1758.7	1813.55	1746.2	1805.81	6.36E+10	1805.81
5	2013-12-31	1806.55	1849.44	1767.99	1848.36	6.5E+10	1848.36

Concatenating, Joining, Merging Data with *pandas*

Concatenating Data

- There are instances where you are importing data that is divided into separate files
  - This is common when dealing with large datasets
  - However, when performing analysis it is easiest to work when they are all contained in the same DataFrame
  - We can use the concatenate method to stack the DataFrames together
- .concat(objs)
  - **Objs:** List of DataFrames that you wish to concatenate

```
#Reload Data before continuing
# Splitting into months for illustrative purposes
>>> sp201802 = sp500['2018-02']
>>> sp201803 = sp500['2018-03']

#Concat Feb and March
>>> spcat = pd.concat([sp201802,sp201803])
>>> print(spcat)
```

Joining Data

- .join(other, how='left', lsuffix='', rsuffix='', sort=False)
  - **Other**: the DataFrame you want to join in
  - **How**:
    - Left: use left index
    - Right: use right index
    - Inner: only the intersection
  - **Suffix**: For overlapping columns, specify the suffix to be used to distinguish
  - **Sort**: Will resort the index after joining

```
>>> bb = bbcad.join(bbus, lsuffix='_tse', rsuffix='_nyse')
>>> bb.loc['2018-01-12':'2018-01-17']

>>> print(bb.loc['2018-01-12':'2018-01-17'])
            High_tse Low_tse Close_tse ... High_nyse Low_nyse Close_nyse
Date
2018-01-12  17.16    16.82     17.00 ...     13.73    13.45      13.65
2018-01-15  17.75    17.25     17.48 ...       NaN      NaN        NaN
2018-01-16  18.07    16.87     17.18 ...     14.55    13.55      13.81
2018-01-17  17.35    16.93     17.00 ...     13.98    13.57      13.67
```

Concat vs Join

- Concatenating can be thought of as stacking on top of each other

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

- Joining is like trying to interweave

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3
6	NaN	NaN	NaN	NaN	B6	D6	F6
7	NaN	NaN	NaN	NaN	B7	D7	F7

Note: Images adapted from: https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html

Merging Data

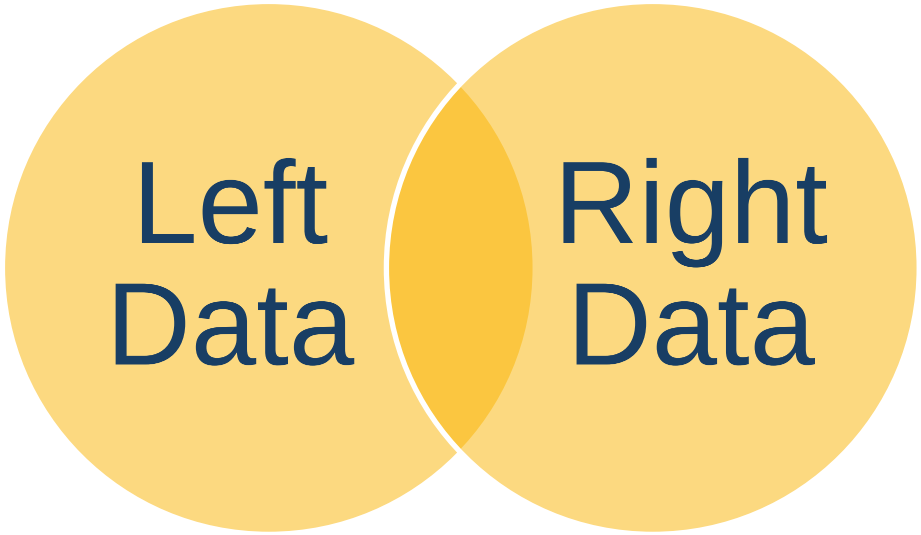
- Merging allows you to combine two DataFrames on column or index match(s)
- df.merge(right df, how='inner', on=None, left_index=False, right_index=False)
  - **How**: specifies the merging method
    - ‘left’ will keep all the 'left' DataFrame rows and only the rows that are matched from the right DataFrame
    - ‘right’ will keep all the 'right' DataFrame rows and only the rows that are matched from the left DataFrame
    - ‘inner’ will keep only the matched pairs from the left and right DataFrames
    - ‘outer’ will keep all the rows from both DataFrames, even if there is no match
  - **On**: the column(s) or index to merge on
    - If the index or column is not common across the two DataFrames (different name or different index), then use **left_on** and **right_on**
  - **Left_Index** and **Right_Index**
    - Boolean (True/False), specify if want to merge on the index
    - Can be efficient when the indices are both DateTime and merging the DataFrames by dates only
- Depending on the index, can have:
  - One to one matching (ideal)
  - One to many (can cause serious problems, exploding DataFrames)
  - Many to One (not a problem, value is just duplicated)

Merging Data – How Parameter

- Think of Venn Diagrams, the two DataFrames are referenced left and right just like the circles

Outer Match
Keeps all data from both DataFrames

Left Match
Keeps all the data from the left frame and the matched data

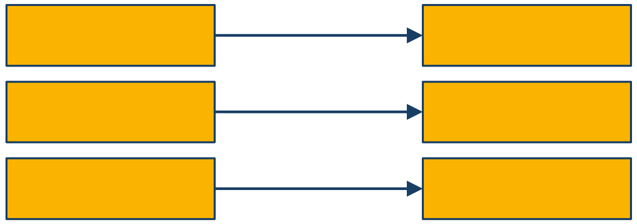


Right Match
Keeps all the data from the right frame and the matched data

Inner Match
Only keep the matched data

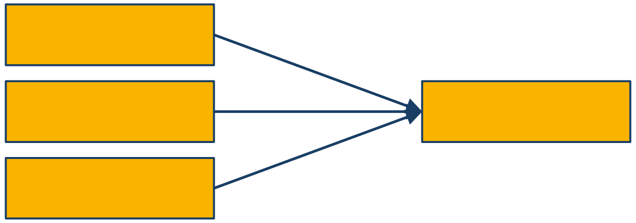
Merging Data – Resulting Size

- One to One matching



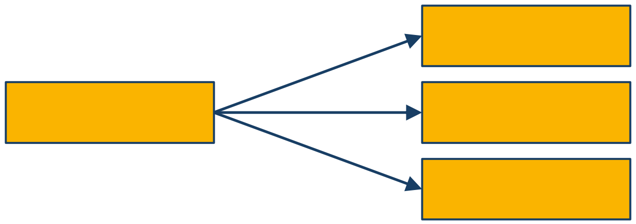
DataFrame will be the same size after matching

- Many to One matching



DataFrame will be the same size after matching

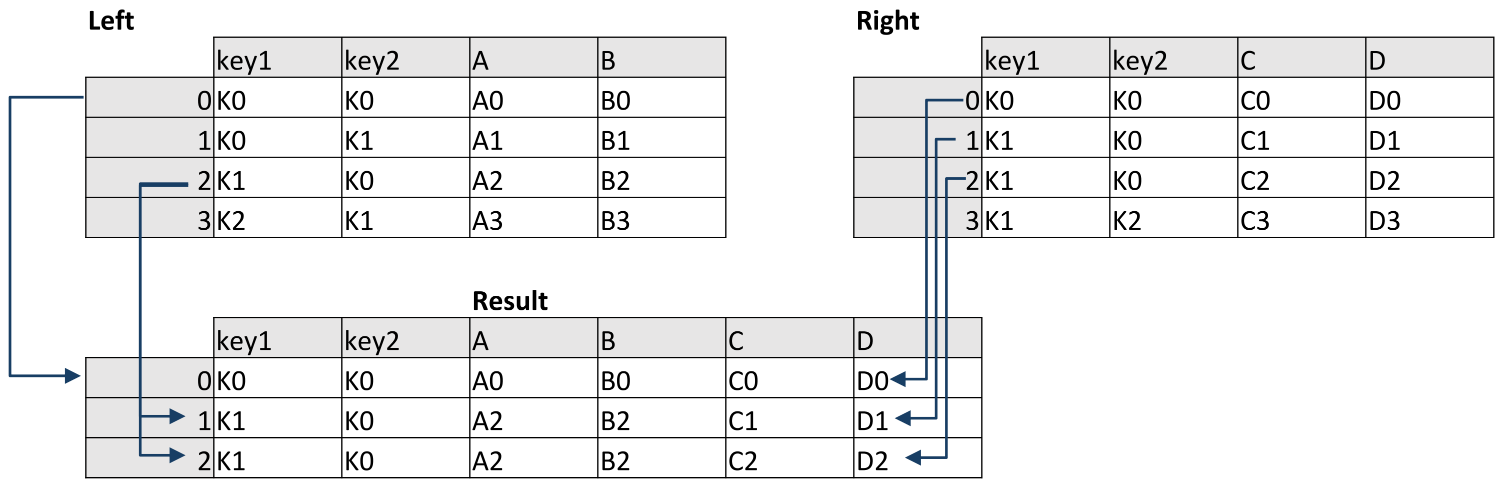
- One to Many matching



DataFrame can become significantly bigger after matching

Merging Data

- Merging is applying logic/control to concatenating and/or joining



Left

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

Right

	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K1	K2	C3	D3

Result

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2

- Merging SP500 & Apple data
  - This will create a new DataFrame with the Open and Close for the two stocks

```
>>> sp_aapl201802 = pd.merge(sp201802[['Open','Close']],
         aapl201802[['Open','Close']],
         left_index=True, right_index=True,
         suffixes=('_sp500','_aapl'))
>>> sp_aapl201802.head()
```

Merging Data

- Merging with Financials is a bit trickier
  - Financial data is only periodical, quarterly or yearly
  - We also have to pay attention to when the information is available
    - If disclosure is Jan 1st 2018 for FY2017, we shouldn't use that data for 2017 dates but rather use FY2016
    - Since we are using daily data, we have to be careful that even if the disclosure happens in the month of September, we shouldn't use it until the next month
- Pandas merge only allows for exact matching
  - We need to create a match dimension that is common between the two datasets
  - This will create a sparse match, there will be gaps between the matches

```
# Convert to date time type (provides functions to easier move dates around)
>>> aapl['Date'] = pd.to_datetime(aapl['Date'])
# Convert to month end to handle alignment with reporting date
>>> aapl['Match Date'] = aapl['Date'] + pd.offsets.MonthEnd(-1)
# using the previous month end to not have information to early
>>> print(aapl.head(30))

         Date       Open       High        Low      Close  Adj Close     Volume Match Date
0  2013-10-01  68.349998  69.877144  68.339996  69.708572  63.523125   88470900 2013-09-30
1  2013-10-02  69.375717  70.257141  69.107140  69.937141  63.731426   72296000 2013-09-30
…
28 2013-11-08  73.511429  74.447144  73.227142  74.365715  68.162682   69829200 2013-10-31
29 2013-11-11  74.284286  74.524284  73.487144  74.150002  67.964958   56863100 2013-10-31
```

Merging Data

- If there are gaps in the DataFrame after merging, the N/As can be filled using the data that is available before and after the missing observation
- .fillna(value=None, method=None, axis=None, inplace=False, limit=None)
  - **Value**, if blank will use the last known value in the column according to index
  - **Method**
    - Backward fill: bfill, will pull the next value and back fill
    - Forward fill: ffill, will push the last value and forward fill
  - **Axis**: 0 by index, 1 by columns (across the row)
  - **Limit** will stop fill the current value after a specific number of times

Index	Known Values	bfill	ffill
2019-01-01	5	5	5
2018-12-31		5	2
2018-12-30		5	2
2018-12-29	2	2	2
2018-12-28		2	6
2018-12-27		2	6
2018-12-26	6	6	6
2018-12-25		6	1
2018-12-24		6	1
2018-12-23	1	1	1

Merging Data – drop Function

- .drop(labels=None, axis=0, inplace=False)
  - **Labels**: specify the index (rows) or columns (variable names) you wish to drop
  - **Axis**:
    - 0 – Index (rows)
    - 1 – Columns (variable names)
  - **Inplace**:
    - Set to True to modify the DataFrame without having to reassign it

```
>>> print(aapl.head())
        Date       Open       High        Low      Close  Adj Close    Volume Match Date
0 2013-10-01  68.349998  69.877144  68.339996  69.708572  63.523125  88470900 2013-09-30
1 2013-10-02  69.375717  70.257141  69.107140  69.937141  63.731426  72296000 2013-09-30
2 2013-10-03  70.072861  70.335716  68.677139  69.058571  62.930801  80688300 2013-09-30
3 2013-10-04  69.122856  69.228569  68.371429  69.004288  62.881340  64717100 2013-09-30
4 2013-10-07  69.508568  70.378571  69.335716  69.678574  63.495800  78073100 2013-09-30
>>> aapl.drop(labels='Match Date',axis=1, inplace=True)
>>> print(aapl.head())
        Date       Open       High        Low      Close  Adj Close    Volume
0 2013-10-01  68.349998  69.877144  68.339996  69.708572  63.523125  88470900
1 2013-10-02  69.375717  70.257141  69.107140  69.937141  63.731426  72296000
2 2013-10-03  70.072861  70.335716  68.677139  69.058571  62.930801  80688300
3 2013-10-04  69.122856  69.228569  68.371429  69.004288  62.881340  64717100
4 2013-10-07  69.508568  70.378571  69.335716  69.678574  63.495800  78073100
```

Merging Data – groupby Function

- We can use the groupby method to control the manipulation, calculation or aggregation of data
  - Python doesn't know which data belong together
  - If you have several stocks prices in the same DataFrame and applied the .shift() method it would just shift the prices over without paying attention to the change in tickers
- .groupby(by=None, axis=0, as_index=True)
  - **By**: Pass column name(s) which to use values to create groups on
  - **Axis**:
    - 0 – rows (typical for Tidy datasets)
    - 1 – columns
  - **As_index**: The keys (by) that are passed will become the index. For example if grouping by ticker, then calling the ticker will bring up all stock prices for that ticker
- The .groupby() creates what is known as a generator
  - A generator is an object that gathers the data only when called
  - Generators are typically iterable, so we can loop over the chunks of data
    - The generator returns a tuple containing the group name (keys), and a DataFrame that contains the data
- We can also call specific columns (variables) like a normal DataFrame, but it is still a generator
  - Need to apply a method to actually get/manipulate the data

Melting Data

- Melting data can be thought of as unpivoting a DataFrame, taking wide-form data and converting it into long-form tidy data
- .melt(df, id_vars=None, value_vars=None, var_name=None, value_name=None)
  - **df**: the DataFrame to melt
  - **id_vars**: list of columns to use as identifiers
  - **value_vars**: columns to melt, if blank will use all columns that are not contained in id_vars
  - **var_name**: name of the variable column after melting
  - **value_name**: name of the value column.

```
# %% Loading WIPO Patent data and melting into Tidy data
>>> patent = pd.read_csv('ExData/patent_data.csv', header=6)
# Column name across to rows, extract names and then remove the rows
>>> new_columns = patent.iloc[0, : 4].values.tolist()
>>> new_columns.extend(patent.columns[4:])
>>> patent.columns = new_columns
>>> patent.drop(0, inplace = True)
>>> patent.head()
# Melting the DataFrame and renaming columns after the fact
>>> patent_melt = pd.melt(patent, id_vars=['Office', 'Technology'], value_vars=['2010', '2011’,
'2012'])
>>> patent_melt.columns = ['country', 'class', 'year', 'number_patents']
```

Melting Data

Office	Office (Code)	Origin	Technology	1985	1986	1987
Australia	AU	Total	Unknown	86	54	57
Australia	AU	Total	1 - Electrical machinery, apparatus, energy	550	603	573
Australia	AU	Total	2 - Audio-visual technology	370	391	349
Australia	AU	Total	3 - Telecommunications	252	267	270
Australia	AU	Total	4 - Digital communication	92	93	99
Australia	AU	Total	5 - Basic communication processes	106	101	102

Office	Technology	value_var	value
Australia	Unknown	1985	86
Australia	1 - Electrical machinery, apparatus, energy	1985	550
Australia	2 - Audio-visual technology	1985	370
Australia	3 - Telecommunications	1985	252
Australia	4 - Digital communication	1985	92
Australia	5 - Basic communication processes	1985	106
Australia	Unknown	1986	54
Australia	1 - Electrical machinery, apparatus, energy	1986	603
Australia	2 - Audio-visual technology	1986	391
Australia	3 - Telecommunications	1986	267
Australia	4 - Digital communication	1986	93
Australia	5 - Basic communication processes	1986	101
Australia	Unknown	1987	57
Australia	1 - Electrical machinery, apparatus, energy	1987	573
Australia	2 - Audio-visual technology	1987	349
Australia	3 - Telecommunications	1987	270
Australia	4 - Digital communication	1987	99
Australia	5 - Basic communication processes	1987	102

Specified in *value_vars*, the columns are transposed and repeated for each identifier(s) specified in *id_vars*

The corresponding values associated with *value_vars* are copied/aggregated as required to align with the *id_vars*

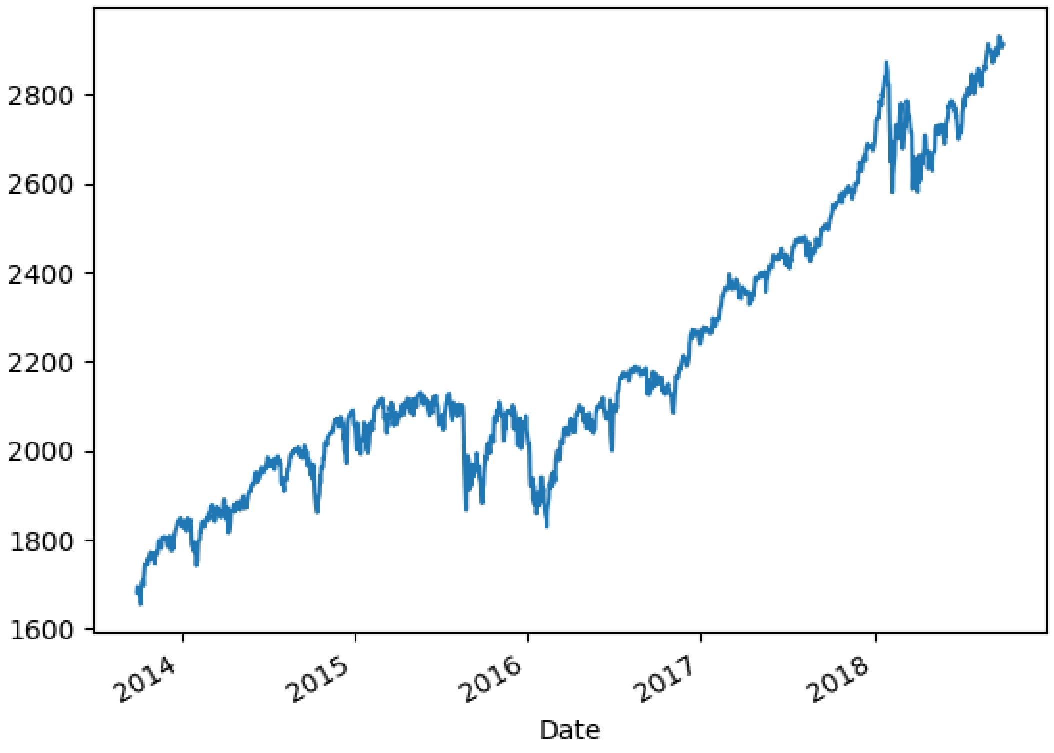
Plotting with *pandas*

Using *pandas* – Plotting

- The pandas package allows to quickly plot
  - Need to import matplotlib.pyplot to show the plot
- Scatter plots are used to plot relationship between two variables
  - Including time series
  - Remember X (horizontal axis), Y (vertical axis)
- Leveraging integration and built in functionality to quickly create a plot
  - Can call .plot() from a *pandas* DataFrame
  - plt.show() is the command to display the plot

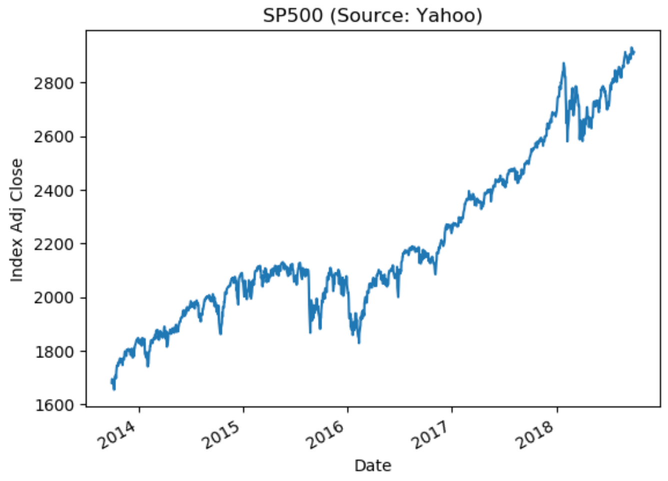
```
# Plotting
#load sp500 data set, with parsed dates
>>> sp500 = pd.read_csv("../StockData/SP500.csv", index_col=0, parse_dates=True)

>>> print(sp500.describe())

#The packages are integrated so you can pick a column and just plot it
>>> import matplotlib.pyplot as plt

>>> sp500['Adj Close'].plot()
>>> plt.show()
>>> plt.ylabel("Index Adj Close")
>>> plt.title("SP500 (Source: Yahoo)")
```

Using *pandas* – Plotting



Using *pandas* – Plotting

- Adding axis labels is good practice
  - The axis is labeled automatically because the package understands the series date index
  - All formatting is done before the plt.show() command
- Axis labels
  - .xlabel() & .ylabel() take strings
  - .tltle() takes a string



Using *Python* - Appendix

Appendix

- Lexicon
  - **Method:** a function that you can call from an object
  - **Function:** a subset of code within your program (script) that can be called by referencing it's name. The function can accept variables from the main program and return values if required.
  - **Cast:** to change a variable/object type to a specified type
- Cells/Chunks
  - Some Python IDEs can interpret a section of code as a "chunk" or a "cell"
  - When the cell is executed, it will run until it reaches the start of the next cell.
  - This is a useful way to run just sections of the code instead of the entire code

```
# %% Cell 1

# The two percent signs after the hashtag denote the start of a
cell. To denote the end, start a new cell.

# %% Cell 2
```

For more information on
The Marquee Group:

56 Temperance Street, Suite 801
Toronto, ON M5H 3V5

TheMarqueeGroup.com
info@themarqueegroup.com

+1 416 583 1802

THE
MARQUEE
GROUP