HDN

National Geospatial – Intelligence College

FUNDAMENTALS
OF PROBLEM SOLVING USING PYTHON

COMPUTATIONAL THINKING

A set of data-driven problem-solving techniques that deconstruct a problem into a format that can be processed by a human, machine, or both.

DECOMPOSITION

DIVISION

Divide the given problem into smaller, independent sub-problems.

CONQUEST

If the problem is small enough, then solve it.

MERGER

Combine the solutions to the sub-problems to get the solution to the given problem.

PATTERN RECOGNITION

APPROACH

Contextual themes among questions or concepts that coordinate analyisis.

VISUAL PATTERNS

Patterns that can be seen while viewing spatial or graphical data.

TEXTUAL PATTERNS

Patterns that can be seen among text-based, referential data.

ABSTRACTION

PROBLEM IDENTIFICATION

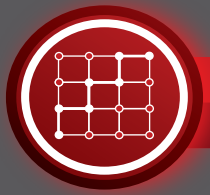
Pinpointing the problem or queston to be answered.

GENERALIZATION

Finding commonalities and removing the details.

FILTERATION

Extracting unnecessary information.



ALGORITHMIC DESIGN

UNIQUENESS

Limited to a single outcome or results; without alternative possibilities.

PRECISION

Definitely or strictly stated, defined, or fixed.

FINITY

Having bounds or limits; not infinite, measurable.

DEFINE | Break down a complex problem or system into smaller parts that are more manageable and easier to understand.

Problem

Sub-Problem

Sub-Problem

Sub-Problem

Planning Phase

DECOMPOSITION

DIVIDE

CONQUER

MERGE

Parallelization

DEFINE | Focus analysis on the important information and exclude or hide irrelevant details.

Significant vs. Insignificant

What are you looking to answer?

Input

Output

ABSTRACTION

PROBLEM ID

GENERALIZE

FILTER

Scale and prioritize information

Tree

Pine Pear
Elm Oak
Fir

APPROACH

VISUAL

TEXTUAL

Identify patterns to solve current and future problems

PATTERN RECOGNITION

Organize data into a common or structured language and terminology

DEFINE | Identify similarities, differences, trends, or other defining characteristics in or across data.

UNIQUENESS

PRECISION

FINITE

Collaborate

Perform Calculations
Process Data
Automated Reasoning

Workflows Pseudocode

ALGORITHMIC DESIGN

Input + Instructions + Stopping Point

DEFINE | Creating a mathematical process to efficiently solve problems or compete a task.

Step 1: Plan
Step 2: Analyze
Step 3: Implement
Step 4: Experiment

Table of Contents

PYTHON
JUST THE BASICS

Python Cheat Sheet

Just the Basics

Created by Arianne Colton and Sean Chen

GENERAL

Python is case sensitive.

Python index starts at **0.**

Python uses **whitespace** (tabs or spaces) to indent code instead of using braces.

HELP

Help Home Page	help ()
Function Help	help (str. replace)
Module Help	help (re)

MODULE (AKA LIBRARY)

Python module is simply a '**.py**' file

List Module Contents	dir (module1)
Load Module	import module1*
Call Function from Module	module1.func1 ()

*Import statement creates a new namespace and executes all the statements in the associated .py file within that namespace. If you want to load the module's content into current namespace, use '**from module1 import**.'

SCALAR TYPES

Check Data Type: type (variable)

SIX COMMONLY USED DATA TYPES

1 **int/long***	**Large** int **automatically** converts to long.
2 **float***	**64 bits,** there is no 'double' type.
3 **bool***	**True** or **False.**
4 **str***	**ASCII** valued in **Python 2x** and **Unicode** in **Python 3.** > String can be single/double triple quotes. > String is a sequence of characters, thus can be treated like other sequences. > Special character can be done via \ or preface with strl = r'this\f?ff' > String formatting can be done in a number of ways template = '%.2f %s ha ha $%d'; strl = template % (4.88, 'hola', 2)

*str(), bool(), int() and float() are also explicit type cast functions.

Continued on Next Page

NOTES

SCALAR TYPES

SIX COMMONLY USED DATA TYPES (Cont.)

5 NoneType (None)	**Python "null" value (ONLY one instance of Non object exists)** > None is not a reserved **keyword** but rather a unique instance of "**NoneType**" > **None** is common default value for optional function arguments: def func1 (a, b, c = None) > Common usage of **None:** if variable is None:
6 datetime	**Built-in python "datetime" module provides "datetime," date," "time" types.** > "datetime" combines information stored in "date" and "time."
Create **date-time** from String	dtl = datetime. strptime ('20091031', '%Y%m%d')
Get "**date**" object	dtl.date ()
Get "**time**" object	dtl.time ()
Format **datetime** to String	dtl.strftime ('%m/%d/%Y%H:%M')
Change Field Value	dt2 = dtl.replace (minute = 0, second = 30)
Get Difference	diff = dt1 - dt2 # diff is a datetime. timedelta object

Note: Most object in Python are nutable except for "strings" and "tuples"

NOTES

DATA STRUCTURES

Note: All non-Get function call i.e. list1.sort() examples below are in-place (without creating a new object) operations unless noted otherwise.

TUPLE

One-dimensional, fixed-length, **immutable** sequence of **Python** objects of **ANY** type.

Create **Tuple**	tup1 = 4, 5, 6 or tup1 = (6, 7, 8)
Create **Nested Tuple**	tup1 = (4, 5, 6), (7, 8)
Convert **Sequence** or **Iterator** to **Tuple**	tuple ([1, 0, 2])
Concatenate **Tuples**	tup1 + tup2
Unpack **Tuple**	a, b, c = tup1

Application of **Tuple**

Swap Variables	b, a, = a, b

LIST

One-dimensional, variable length, mutable (i.e., contents can be modified) sequence of **Python** objects of ANY type.

Create **List**	list1 = [1, 'a', 3] or list1 = list (tupl)
Concatenate Lists*	list1 + list2 or list1.extend(list2)
Append to **End of List**	list1.append ('b')
Inset to **Specific Position**	list1.insert(posIdx, 'b')**
Inverse of **Insert**	valueAtIdx = list1. pop(posIdx)
Remove **First Value** from **List**	list1.remove('a')
Check **Membership**	3 in list1 => True***
Sort **List**	list1.sort()
Sort with **User Supplied Function**	list1.sort(key = len) # sort by length

*List concatenation using "+" is expensive since a new list must be created and objects copied over. Thus, **extend ()** is preferable.

**Insert is comutationally expensive compared with append.

***Checking that a list contains a value is lot slower than dicts and sets as **Python** makes a linear scan where others (based on hash tables) in constant time.

Continued on Next Page

NOTES

DATA STRUCTURES

LIST (Cont.)

Built-in "bisect module":

> Implements **binary search** and **insertion** into a sorted list.

> "bisect.bisect" finds the location, where "bisect.insort" actually inserts into that location.

■ WARNING: bisect module functions do not check whether the list is sorted, doing so would be computationally expensive. Thus, using them in an unsorted list will succeed without error but may lead to incorrect results.

SLICING FOR SEQUENCE TYPES*

*Sequence types include "**str**," "**array**," "**tuple**", "**list**", etc.

Notation	list1 [start:stop]
	list1 [start:stop:-step] (If step is used)§

Note:

> "start" index is included, but "stop" index is NOT.

> **start/stop** can be omitted in which they default to the **start/end**.

§ **Application** of '**step**':

Take every other element	list1 [: : 2]
Reverse a string	strl [: : -1]

DICT (HASH MAP)

Create **Dict**	dict1 = {'key1' : 'value1', 2 : [3, 2]}
Create **Dict** from **Sequence**	dict (zip (keyList, valueList))
Get/Set/Insert Element	dict1 ['key1']* dict1 ['key1'] = 'newValue'
Get with Default Value	dict1.get ('key', defaultValue)**
Check if Key Exists	'key1' in dict1
Delete Element	del dict1 ['key']
Get Key List	dict1.keys() ***
Get Value List	dict1.values() ***
Update Values	dict1.update (dict2) #**dict1** values are replaced by **dict2**

*"**KeyError**" exception if thekey does not exist.

"get()**" by default (aka no 'defaultValue') will return "None" if the key does not exist.

***Returns the list of keys and valuers in the same order. However, the order is not any particualr order, aka ist is most likely not sorted.

Continued on Next Page

NOTES

DATA STRUCTURES

DICT (HASH MAP)

Valid dict key types

> Keys have to be imutable like scalar types (int, float, string) or tuples (all the objects in the tuple need to be immutable too).

> The technical term here is 'hashability' check whether an object is hashable wit the hash ("this is string"), hash ([1, 2]) - this would fail.

SET

> A set is an **unordered** collection of **UNIQUE** elements.

> You can think of them like dicts but keys only.

Create **Set**	set ([3, 6, 3]) or {3, 6, 3}
Test Subset	set1. issubset (set2)
Test Superset	set1.issuperset (set2)
Test sets have same content	set == set 2

> **Set Operations:**

Union (aka "or")	set1 | set2
Intersection (aka "and")	set1 & set2
Difference	set1 - set2
Symmetric Diffence (aka "xor")	set1 ^ set2

NOTES

PYTHON
MATPLOTLIB

Python Data Science

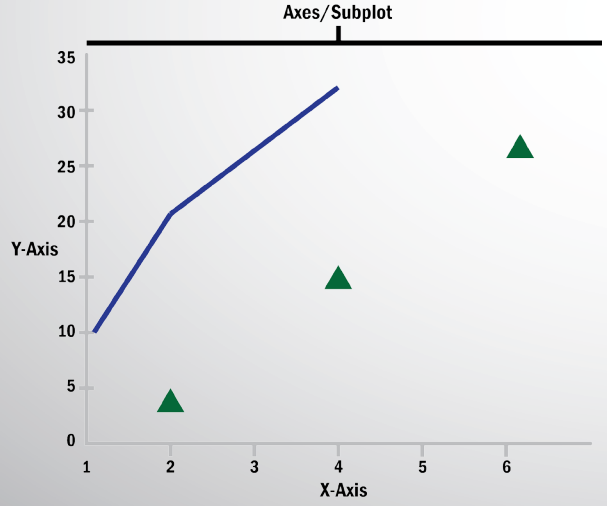
Cheat Sheet | Matplotlib

Learn Python Interactively at www.DataCamp.com

MATPLOTLIB

Matplotlib is a **Python 2D** plotting library that produces **publication-quality** figures in a variety of **hardcopy** formats and **interactive** environments across platforms.

PLOT ANATOMY



❶ Prepare Data ❷ Create Plot ❸ Plot ❹ Customize Plot
❺ Save Plot ❻ Show Plot

```
>>> import matplotlib.pyplot as plt
>>> x = [1, 2, 3, 4]  ❶
>>> y = [10, 20, 25, 30]
>>> fig = plt.figure ()  ❷
>>> ax = fig.add_subplot (111) ❸
>>> ax. plot (x, y, color='lightblue',
      linewidth=3) ❸❹
>>> ax. scatter ([2, 4, 6],
                 [5, 15, 25],
                 color='darkgreen'
                 marker='^') ❸❹
>>> ax.set_xlim (1, 6.5)
>>> plt.savefig ('foo.png') ❺
>>> plt.show() ❻
```

Continued on Next Page

NOTES

MATPLOTLIB

❶ PREPARE THE DATA

ALSO SEE LISTS & NUMPY

1D Data

```
>>> import numpy as np
>>> x = np.linspace (0, 10, 100)
>>> y = np.cos (x)
>>> z = np.sin (x)
```

2D Data or Images

```
>>> data = 2 * np.random.random
    ( (10, 10) )
>>> data2 = 3 * np.random.random
    ( (10, 10) )
>>> Y, X = np.mgrid[-3:3:100j, -3:3:100j]
>>> U = -1 - X**2 + Y
>>> V = 1 + X - Y**2
>>> from matplotlib.cbook import get_sample
    _data
>>> img = np.load(get_sample data
    ('axes_grid/bivariate_normal.npy'))
```

❷ CREATE PLOT

```
>>> import matplotlib.pyplot as plt
```

Figure

```
>>> fig = plt.figure ()
>>> fig2 = plt.figure
    (figsize=plt.figaspect (2.0) )
```

Axes

All **plotting** is done with respect to an Axes. In most cases, a **subplot** will fit your needs. A subplot is an **axes** on a **grid system**.

```
>>> fig.add_axes ()
>>> ax1 = fig.add_subplot (221)
    # row-col-num
>>> ax3 = fig.add_subplot (212)
>>> fig3, axes = plt.subplots (nrows=2,
    ncols=2)
>>> fig4, axes2 = plt.subplots (ncols=3)
```

NOTES

MATPLOTLIB

❸ PLOTTING ROUTINES

ID Data

```
>>> lines = ax.plot (x, y)
```
Draw **points** with **lines** or **markers** connecting them

```
>>> ax.scatter (x, y)
```
Draw **unconnected** points, scaled or colored

```
>>> axes [0, 0] .bar ( [1, 2, 3,], [3, 4, 5] )
```
Plot **vertical rectangles** (constant width)

```
>>> axes [1, 0] .barh ([0.5, 1, 2.5], [0, 1, 2])
```
Plot **horizontal rectangles** (constant height)

```
>>> axes [1, 1] .axhline (0.45)
```
Draw a **horizontal line** across axes

```
>>> axes [0, 1] .axvline (0.65)
```
Draw a **vertical line** across axes

```
>>> ax.fill (x, y, color='blue')
```
Draw **filled** polygons

```
>>> ax.fill_between (x, y, color='yellow')
```
Fill between **y-values** and **0**

2D Data or Images

```
>>> fig, ax = plt.subplots()
>>> im = ax.imshow (img,
                    cmap = 'gist_earth',
                    interpolation = 'nearest',
                    vmin=-2,
                    vmax=2
```
Colormapped or **RGB** arrays

Vector Fields

```
>>> axes [0, 1] .arrow (0, 0, 0.5, 0.5)
```
Add an **arrow** to the **axes**

```
>>> axes [1, 1] .quiver (y, z)
```
Plot a **2D** field of **arrows**

```
>>> axes [0, 1] .streamplot (X, Y, U, V)
```
Plot **2D vector** fields

Data Distributions

```
>>> ax1.hist (y)
```
Plot a **histogram**

Continued on Next Page

NOTES

Data Distributions (Cont.)

```
>>> ax3.boxplot (y)
```
Make a **box** and **whisker** plot

```
>>> ax3.violinplot (z)
```
Make a **violin** plot

❹ CUSTOMIZE PLOT

Colors, Color Bars & Color Maps

```
>>> plt.plot (x, x, x, x**2, x,  x**3)
>>> ax.plot (x, y, alpha = 0.4)
>>> ax.plot (x, y, c='k')
>>> fig.colorbar (im, orientation=
    'horizontal')
>>> im = ax.imshow (img,
                    cmap='seismic')
```

Markers

```
>>> fig, ax = plt.subplots ()
>>> ax.scatter (x, y, marker=".")
>>> ax.plot (x, y, marker="0")
```

Linestyles

```
>>> plt.plot (x, y, linewidth=4.0)
>>> plt.plot (x, y, ls = 'solid')
>>> plt.plot (x, y, ls = '--')
>>> plt.plot (x, y, '--', x**2, y**2, '-.')
>>> plt.setp (lines, color = 'r',
    linewidth = 4.0)
```

Text & Annotations

```
>>> ax.text (1,
             -2.1,
             'Example Graph',
             style='italic'
```

Continued on Next Page

NOTES

MATPLOTLIB

Text & Annotations (Cont.)

```
>>> ax.annotate (“Sine”,
        xy=(8, 0),
        xycoords=’data’,
        xytext=(10.5, 0),
        textcoords=’data’,
        arrowprops=dict (arrowstyle =”->”,
                connectionstyle=”arc3”) , )
```

Mathtext

```
>>> plt.title (r’$sigma_i=15$’,
    fontsize=20)
```

Limits, Legends & Layouts

Limits & Autoscaling

```
>>> ax.margins (x=0. 0, y=0.1)
```

Add **padding** to a plot

```
>>> ax.axis (‘equal’)
```

Set the **aspect ratio** of the plot to **1**

```
>>> ax.set (xlim = [0, 10.5], ylim =
    [-1.5, 1.5])
```

Set **limits** for **x**-and **y**-axis

```
>>> ax.set_xlim (0, 10.5)
```

Set **limits** for **x**-axis

Legends

```
>>> ax.set (title= ‘An Example Axes’,
                ylabel= ‘Y-Axis’,
                xlabel= ‘X-Axis’)
```

Set a **title** and **x**-and **y**-axis labels

```
>>> ax.legend (loc= ‘best’,)
```

No **overlapping** plot elements

Ticks

```
>>> ax.xaxis.set (ticks=range (1.5),
        ticklables= [3, 100, -12, “foo”])
```

Manually **set x**-ticks

Continued on Next Page

NOTES

```
>>> ax. tick_params (axis='y',
                     direction='inout',
                     length=10)
```

Make **y-ticks** **longer** and go **in** and **out**

Subplot Spacing

```
>>> fig3.subplots_adjust
      (wspace=0.5,
       hspace=0.3,
       left=0.125,
       right=0.9,
       top=0.9,
       bottom=0.1)
```

Adjust the **spacing** between **subplots**

```
>>> fig.tight_layout()
```

Fit **subplot(s)** in to the **figure** area

Axis Spines

```
>>> ax1.spines ['top'] .set_visible
    (False)
```

Make the **top** axis line for a **plot** invisible

```
>>> ax1.spines ['bottom'] .set_position (
    ('outward', 10) )
```

Move the **bottom axis** line outward

❺ SAVE PLOT

Save Figures

```
>>> plt.savefig ('foo.png')
```

Save Transparent Figures

```
>>> plt.savefig ('foo.png',
    transparent=True)
```

❻ SHOW PLOT

```
>>> plt.show ()
```

Close & Clear

```
>>> plt.cla ()
```

Clear an **axis**

```
>>> plt.clf ()
```

Clear the **entire** figure

```
>>> plt.close ()
```

Close a **window**

NOTES

PANDAS
DATA ANALYSIS

NOTES

PANDAS Data Analysis Cheat Sheet

Created by Arianne Colton and Sean Chen

DATA STRUCTURES

SERIES (1D)

One-dimensional array-like object containing an **array** of data (of any NumPy data type) and an **associated** array of **data lables**, called its "index." If index of data is **not specified**, then a default one consisting of the integers **0** through **N-1** is created.

Create **Series**	series1 = pd.Series ([1, 2], index = ['a', 'b']) series1 = pd. Series (dict1)*
Get Series **Values**	series1.values
Get Values by **Index**	series1 ['a'] series1 [['b', 'a']]
Get Series **Index**	series1.index
Get Series **Attribute** **(None is Default)**	series1.name series1.index.name
Common **Index Values are Added	series1 + series2
Unique but **Unsorted**	series2 = series1. unique()

*Can think of **Series** as a fixed-length, ordered dict. **Series** can be substitued into many functions that expect a dict.

Auto-align differently-indexed data in **arithmetic operations.

Get **Columns** and **Row Names**	df1.columns df1. index
Get Name Attribute **(None is Default)**	df1.columns.name df1.index.name
Get **Values**	df1.values # returns the data as a **2D** ndarray; the **dtype** will be chosen to accomandate all of the columns
Get **Column as **Series**	df1 ['state'] or df1. state
Get **Row as Series	df1.ix ['row2'] or df1.ix [1]
Assign a column that doesn't exist will create a new column	df1 ['eastern'] = df1.state == 'Ohio'

Continued on Next Page

NOTES

DATA STRUCTURES

SERIES (1D) cont.

Delete a **column**	del df1 [‘eastern’]
Switch **columns** and **rows**	df1.T
Get Series **Index**	series1.index

*Dicts of **Series** are treated the same as Nested dict of dicts.

Data returned is a ‘view’ on the underlying data, **NOT a copy. Thus, any in-place **modifications** to the data will be reflected in df1.

DATAFRAME (2D)

Tabular data structure with **ordered collections** of columns, each of which can be different value type. **Data Frame** (DF) can be thought of as a dict of Series.

Create DF (from a dict of **equal-length** lists or NumPy arrays)	dict1 = {‘state’: [‘Ohio’, ‘CA’], ‘year’: [2000, 2010]} df1 = pd.Data-Frame (dict1)
	# columns are placed in **sorted order**

Create DF (cont.)	df1 = pd.Data-Frame (dict1, index = [‘row1’, ‘row2’]))
	# specifying **index**
	df1 = pd.Data-Frame (dict1, columns = [‘year’, ‘state’])
	# columns are placed in your **give order**
*Create **DF** (from nested dict of dicts) The **inner keys** as row indices	dict1 = {‘coll’: {‘row1’: 1, ‘row2 : 2}, ‘col2’ : {‘row1 “ 3, ‘row2’ : 4 } } df1 = pd. Data-Frame (dict1)

NOTES

DATA STRUCTURES

PANEL DATA (3D)

Create **Panel Data** : (Each item in the Panel is a DF)

```
import pandas_datareader.data as
web

panell = pd.Panel ({stk : web.
get_data_yahoo (stk, '1/1/2000',
'1/1/2010') for stk in ['AAPL', 'IBM'] } )
```

```
# panel1 Dimensions: 2 (item)
*861 (major) *6 (minor)
```

"Stacked" DF form: (Useful way to represent panel data)

```
panell = panell.swapaxes('item',
'minor')
panell.ix[:, '6/1/2003', :].to_frame() *
=> Stacked DF (with hierarchical indexing**) :
#                  Open High Low Close Volumn Adj-Close
# major            minor
# 2003-06-02  AAPL
#                  IBM
# 2003-06-02  AAPL
#                  IBM
```

*DF has a "to_panel()" method which is the inverse of "to_frame()".

**Hierarchical indexing makes N-dimensional arrays unnecessary in a lot of cases AKa prefer to use Staced DF, not Panel data.

INDEX OBJECTS

Immutable objects that hold the **axis labels** and other metadata (i.e. axis name)

> Index, MultiIndex, DatetimeIndex, PeriodIndex
> Any sequence of labels used when constructing Series or DF internally converted to an Index.
> Can function as fixed-size set in addition to being array-like.

HIERARCHICAL INDEXING

Multiple index levels on an **axis**: A way to work with higher **dimensional data** in a lower dimensional form.

```
series1 = Series (np.random.randn
(6), index = [ ['a', 'a', 'a', 'b' 'b', 'b'],
[1, 2, 3, 1, 2, 3] ] )
series1.index.names = ['key1', 'key2']
```

Continued on Next Page

NOTES

HIERARCHICAL INDEXING (Cont.)

Series **Partial** Indexing	series1 [‘b’] #Outer Level series1 [:, 2] #Inner Level
DF **Partial** Indexing	df1 [‘outerCo13’, ‘InnerCol2’] Or df1 [‘outerCo13’] [‘InnerCo12’]

Swaping and Sorting Levels

Swap **Level** (level interchanged)*	swapSeries1 = series1. swaplevel (‘key1’, ‘key2’)
Sort **Level**	series1.sortlevel (1) #sorts according to first inner level
Common Ops: **Swap** and **Sort****	series1.swaplevel (0, 1) .sortlevel (0) #the order of **rows** also change

*The order of the **rows** do not change. Only the **two levels** got swapped.

**Data selection performance is much better if the index is sorted starting with the outermost level, as a result of calling sortlevel (0) or sort_index().

DataFrame’s Columns as Indexes

New DF using columns as index	df2 = df1.set_ index([‘co13, ‘co14]) * ‡ # co13 becomes the **outermost index**, co14 becomes **inner index**. Values of **co13**, **co14** become the index values.

* “resent_index” does the opposite of “set_index”, the hierarchical index are moved into columns.

‡By default, ‘col3’ and ‘co14’ will be removed from the DF, though you can leave them by option: ‘drop = False’,

Continued on Next Page

NOTES

MISSING DATA

HIERARCHICAL INDEXING (Cont.)

Python	NaN - np.nan (not a number)
Pandas*	NaN or Python built-in None mean missing/ NA values

*Use pd.isnull (), pd.notnull () **or** series1/df1.isnull () to detect missing data.

FILTERING OUT MISSING DATA

dropna () returns with **ONLY** non-null data, source data **NOT** modified.

```
df1.dropna ()
# drop any row containing missing value
df1.dropna (axis = 1)
# drop any column containing missing values
df1.dropna (how = 'all')
# drop row that are all missing
df1.dropna (thresh = 3) # drop any row containing < 3 number of observations
```

FILLING IN MISSING DATA

```
df2 = df1.fillna (0) # fill all missing data with 0
df1.fillna ('inplace = True')
# modify in-place
```

Use a different fill value for each column:

```
df1.fi11na ({'co11' : 0, 'co12' : -1})
```

Only forward fill the **2 missing** values in front:

```
df1.fillna (method = 'ffill', limit = 2)
```

i.e. for **column1**, if row **3-6** are missing, so **3** and **4** get filled with the value from **2**, **NOT 5** and **6**

ESSENTIAL FUNCTIONALITY

INDEXING (SLICING/SUBSETTING)†

† Same as 'NdArray' : In **INDEXING** : **'view'** of the source array is returned.

† Endpoint is inclusinve in pandas slicing with labels : series1['a' : 'c'] where Python slicking is NOT. Note that pandas non-label (i.e. integer) slicking is still non-inclusive.

Index by Column(s)	df1 ['co11'] df1 ['co11, 'co13']]

Continued on Next Page

NOTES

ESSENTIAL FUNCTIONALITY

INDEXING (SLICING/SUBSETTING)†

Index by Row(s)	df1.ix ['row1'] df1.ix ['row1']
Index by **Both Columns(s)** and **Row(s)**	df1.ix [['row2', 'row1'], 'co13']
Boolean Indexing	df1 [[True, False]] df1 [df1['co12'] > 6] * #returns **df** that has **co12** value **> 6**

* Note that df1['co12'] > 6 returns a boolean Series, with each **True/False** value determine whether the **respective row** in the result. Avoid integer indexing since it might introduce subtle bugs (e.g. series1[-1]). If have to use **position-based** indexing, use "iget_value()" from Series and "irow/icol ()" from DF instead of integer indexing.

DROPPING ROWS/COLUMNS

Drop operation returns a new object (i.e. DF):

Remove Rows(s) (axis = 0 is default)	df1.drop ('row1) df1.drop (['row1', 'row3'])
Remove Column(s)	df1.drop ('co12', axis = 1)

REINDEXING

Create a new **object** with rearranging data conformed to a new index, introducing **missing values** if any index values were not already present.

Change df1 **Date Index** Values to the new Index **Values** (**ReIndex** default is **row index**)	date_index = pd.date_range ('01/23/2010', periods = 10, freq = 'D') df1.reindex (date_index)
Replace **Missing Values** (NaN) with **0**	df1.reindex (date_index, fill_value = 0)
ReIndex **Columns**	df1.reindex (columns = ['a', 'b'])
ReIndex **Both Rows and Columns**	df1.reindex(index = [. .], columns = [. .])
Succint ReIndex	df1.ix [[. .], [. .]]

ARITHMETIC AND DATA ALIGNMENT

> df1 + df2 : For indices that don't overlap, internal data alignment introduces NaN.

Continued on Next Page

NOTES

ESSENTIAL FUNCTIONALITY

ARITHMETIC AND DATA ALIGNMENT (Cont.)

1 Instead of NaN, replace with 0 for the indice that is not found in the df:

```
df1.add(df2, fill_value = 0)
```

2 Useful Operations:

```
df1 - df1.ix[0] # subtract ever row in df1 by first row
```

SORTING AND RANKING

Sort Index/Column †

> sort_index () returns a new, **sorted** object. Default is "ascending = True".

> Row index are **sorted** by default, "axis = 1" is used for sorting column.

† Sorting Index/Column means sort the row/column labels, not sorting the data.

Sort Data

Missing values (np.nan) are sorted to the end of the Series by default

Series **Sorting**	sortedS1 = series1.order () series1.sort () # In-place **sort**
DF Sorting	df1.sort_index (by = ['co12', 'co11]) # sort by **co12** first then **co1**

Ranking

Break **rank** ties by assigning each **tie-group** the mean rank. (e.g. **3**, **3** are tie as the 5th place; thus, the result is **5.5** for each)

Output **Rank** of **Each Element** (Rank start from 1)	series1.rank () df1.rank (axis = 1) # rank each **row's** value

FUNCTION APPLICATIONS

NumPy works fine with **pandas objects** : np.abs (df1)

Applying a **Function** to Each Column or **Row** (Default is to **apply** to each column: axis = 0)	f = lambda x: x.max () - x.min () # return a **scalar** value def f (x) : return Series ([x.max (), x.min ()]) # return multiple values df1.apply (f)

Continued on Next Page

NOTES

FUNCTION APPLICATIONS (Cont.)

Applying a **Function** Element-Wise	f = lambda x: '%.2f' %x df1.applymap (f) # **format** each entry to **2-decimals**

UNIQUE, COUNTS

> It's NOT mandatory for index labels to be unique although many functions require it. Check via: series1/df1.index.is_unique

> pd.value_counts () returns value frequency.

DATA AGGREGATION AND GROUP OPERATIONS

Categorizing a dataset and applying a function to each group, whether an aggregation or transformation.

Note Aggergation of "Time Series" data - please see Time Series sectionl Special use case of "groupby" is used - called "resampling".

GROUPBY (SPLIT-APPLY-COMBINE)

Similar to **SQL GroupBy**

Compute **Group** Mean	df1.groupby ('co12'). mean ()
GroupBy More than **One Key**	df1.groupby ([df1 ['co12'], df1 ['co13']]).mean () # gets the mean of each group formed by 'co12'
"GroupBy" Object: **(ONLY computed intermediate data about the group key** - df1 ['co12']	grouped = df1['co11']. groupby (df1 ['co12']) grouped.mean() # gets the mean of each **group** formed by **'col2'**

Note: Any **missing values** in the group are excluded from the result.

Iterating Over GroupBy Object

"GroupBy" object supports iteration: generating a sequence of **2-tuples** containing the group name along with the chunk of **data**.

```
for name, groupdata in df1.groupby ('co12'):

# name is single value, groupdata is filtered DF contains data only match that single value.
```

Continued on Next Page

NOTES

DATA AGGREGATION AND GROUP OPERATIONS

Iterating Over GroupBy Object (cont.)

```
for (k1, k2), groupdata in df1.
groupby ( [ 'co12', 'co13'] ):

# If groupby multiple keys: first element in the tuple is
a tuple of key values.
```

Convert **Groups** to Dict	dict (list (df1.groupby ('co12'))) #co12 unique values will be keys of dict
Group **columns** by "**dtype**"	grouped = df1.group-by ([df1.dtypes, axis = 1) dict (list (grouped)) # separates data into different types

Grouping with Functions

Any **function** passed as a **group key** will be called once per (default is row index) **value**, with the return values being used as the **group names**. (This assumes row index are named)

```
df1.groupby (len) .sum()

# Returns a DF with row index that are length of
   the names.  Thus, names of same length will
   sum their values.  Column names retain.
```

DATA AGGREGATION

Data aggregation means any data transformation that produces **scalar** values from arrays, such as "mean," "max," etc.

Use **Self-Defined** Function	def func1 (array) : ... grouped.agg (func1)
Get DF with **Column** Names as **Function** Names	grouped.agg ([mean, std])
Get DF with **Self-Defined** Column Names	grouped.agg ([('co11', mean), ('co12', std)])

Continued on Next Page

NOTES

DATA AGGREGATION AND GROUP OPERATIONS

DATA AGGREGATION (Cont.)

Use Different **Function** Depending on the Column	grouped.agg ({ 'co11' : [min, max], 'co13' : sum})

DATA WRANGLING: MERGE, RESHAPE, CLEAN, TRANSFORM

COMBINING AND MERGING DATA

1. pd.merge () **aka database "join"** : connects rows in DF based on one or more keys

> Merge via **Column** (Common)

```
df3 = pd.merge (df1, df2, on = 'co12') *
# INNER join is default Or use option: how = 'outer/left/right'
# the indexes of df1 and df2 are discarded in df3
```

* Use **ALL** overlapping column names as the keys to merge. Good practice is to specify the keys : on = ['co12', 'co13'].

* If different key name in df1 and df2, use option: left on ='1key', right on ='rkey'

> Merge via **Row** (Uncommon)

```
df3 = pd.merge (df1, df2, left_index = True, right_index = True)
# Use indexes as merge key : aka rows with same index value are joined together.
```

2. pd.concat () : **glues or stacks** objects along an **axis** (default is along "rows : axis = 0")

```
df3 = pd.concat ( [df1, df2], ignore_index = True)
# ignore_index = True : discard indexes in df3
# If df1 has 2 rows, df2 has 3 rows, then df3 has 5 rows
```

3. combine_first () : combine **data** with **overlap**, patching **missing value**.

```
df3 + df1.combine_first (df2)
# df1 and df2 indexes overlap in full or part. If a row NOT exist in df1 but in df2, it will be in df3. If row1 of df1 and row3 of df2 have the same index value, but row1's co13 value is NA, df3 get this row with the co13 data from df2
```

NOTES

DATA WRANGLING:
MERGE, RESHAPE, CLEAN, TRANSFORM

PIVOTING

1. Pivoting

> Common **format** of storing multiple **"time series"** in databases and CSV is:

Long/Stacked Format:	"date, stock_name, price"

> However, a **DF** with these **3 columns** data like above will be difficult to work with. Thus, **"wide"** format is preferred: **'date'** as row index, 'stock_name' as columns, 'price' as DF data values.

```
pivotedDf2 = df1.pivot ('date', 'stock_name', 'price' )

# Example pivotedDf2 :
#                AAPL  IBM   JD
# 2003-06-01     120.2 100.1 20.8
```

COMMON OPERATIONS

1. Removing Duplicate Rows

```
series1 = df1.duplicated ()

# Boolean series1 indicating whether each row is
  a duplicate or not.

df2 = df1.drop_duplicates ()

# Duplicates has been dropped in df2.
```

2. Add New Column Based on Value of Column(s)

```
df1 [ 'newCo1'] = df1 [ 'co12' ] .map (dict1)

# Maps co12 value as dict1's key, gets
  dicts1's value
```

3. Replacing Values

```
# Replace is NOT In-Place
df2 = df1.replace (np.nan, 100)

# Replace Multiple Values at Once
df2 = df1.replace ( [-1, np.nan], 100 )
df2 = df2.replace ( [-1, np.nan], [1, 2] )

# Argument Can Be a Dict as Well
df2 = df1.replace ( {-1 : 1, nm.nan : 2} )
```

Continued on Next Page

NOTES

DATA WRANGLING:
MERGE, RESHAPE, CLEAN, TRANSFORM

COMMON OPERATIONS (Cont.)

4. Renaming Axis Indexes

Convert Index to **Upper** Case	df1.index = df1.index.map (str.upper)
Rename '**row1**' to '**newRow1**'	df2 = df1.rename (index = {'row1' : 'newRow1' }, columns = str.upper) # Optionally inplace = True

5. Discretization and Binning

> Continuous data is often **discretized** into "bins" for analysis.

```
# Divide Data into 2 Bins of Number (18 - 26], (26 - 35] # ']' means inclusive, ')' is NOT inclusive

bins = [18, 26, 35]

cat = pd.cut (array1, bins, labels=[. .])
# cat is "Categorial" object.
```

```
pd.value_counts (cat)

cat = pd.cut (array1, numofBins)
# Compute equal-length bins based on min and max values in array1

cat = pd.qcut (array1, numofBins)
# Bins the data based on sample quantiles - roughly equal-size bins
```

6. Detecting and Filtering Outliers

> any () test **along** an axis if any element is "True". Default is test along **column** (axis = 0).

```
df1 [ (np.abs (df1) > 3). any (axis = 1) ]
# Select all rows having a value > 3 or < -3.
# Another useful function : np.sign () returns 1 or -1.
```

7. Permutation and Random Sampling

```
randomOrder = np.random.permutation (df1.shape [0] )

df2 = df1.take (randomOrder)
```

Continued on Next Page

NOTES

DATA WRANGLING:
MERGE, RESHAPE, CLEAN, TRANSFORM

COMMON OPERATIONS (Cont.)

8. Computing Indicator/Dummy Vaiables

> If a column in **DF** has "**K**" distinct values, derive a "indicator" **DF** containing K columns of 0s and 1s. 1 means exist, 0 means **NOT** exist.

```
dummyDf = pd.get_dummies (df1 ['co12], prefix = 'col-')

# Add prefix to the K column names
```

GETTING DATA

TEXT FORMAT (CSV)

```
df1 = pd.read_csv (file/URL/file-like-object, sep = ', ', header = None)

# Type-Inference:  do NOT have to specify which columns are numeric, integer, boolean or string.

# In Pandas, missing data in the source data are usually empty string, NA, -1, #IND or NULL.  You can specify missing values via option i.e. :
 na_values = [ 'NULL'] .
```

```
# Default delimiter is comma.

# Default is first row is the column header.
```

```
df1 = pd.read_csv ( . . , names = [ . . ] )

# Explicitly specify column header, also
  imply first row is data.
```

```
df1.to_csv (filepath/sys.stdout,
sep = ' , ')

# Missing values appear as empty strings in the out.
Thus, It is better to add option i.e.,: na_rep =
'NULL'
```

JSON (JAVASCRIPT OBJECT NOTATION) DATA

One of the **standard** formats for sending data by HTTP request between **web browsers** and other applications. It is a much more flexible data format than tabular text from like **CSV**.

Convert **JSON** string to Python form	data = json.load (jsonObj)

Continued on Next Page

NOTES

DATA WRANGLING:
MERGE, RESHAPE, CLEAN, TRANSFORM

JSON (JAVASCRIPT OBJECT NOTATION) DATA

Convert **Python** object to JSON	asJson = json.dumps (data)
Create DF from **JSON**	df1 = pd.DataFrame (data['name'], columns = ['field1'])

XML AND HTML DATA

HTML:

```
doc = 1xml.html.
parse(urlopen ('http: / / . .') ) .getroot()
tables = doc.finda11 (' . / /table')
rows = tables [1].findall (' . / /tr')
```

XML:

```
lxml.objectify.parse (open (filepath) ) .
getroot ()
```

NOTES

NUMPY
PYTHON PACKAGE

Numpy Cheat Sheet

Python Package

Created by Arianne Colton and Sean Chen

NUMPY (NUMERICAL PYTHON)

What is NumPy?

Foundation package for scientific computing in **Python**

Why NumPy?

> Numpy '**ndarray**' is a much efficient way of storing and manipulating "numerical data" than the built-in **Python** data structures.

> Libraries written in **lower-level languages**, such as **C**, can operate on data stored in Numpy '**ndarray**' without copying any data.

N-DIMENSIONAL ARRAY (NDARRAY)

Why is NdArray?

Fast and space-efficient multidimensional array (container for homogeneous data) providing **vectorized** arithmetic operations.

Create **NdArray**	np.array (seq1) # seq1-is any sequence like object, i.e. [1,2,3]
Create **Special NdArray**	1, np.zeros (10) # **one**-dimensional ndarray with 10 elements of value **0** 2, np.ones (2, 3) # **two**-dimensional **ndarray** with **6** elements of value **1** 3, np.empty (3, 4, 5) * # **three**-dimensional **ndarray** of **uninitialized** values 4, np.eye (N) or np.identity (N) # creates **N** by **N** identity matrix
NdArray version of Python's range	np.arange (1, 10)
Get # of **Dimension**	ndarray1.ndim
Get **Dimension** Size	dim1size, dim2size, .. = ndarray1.shape

Continued on Next Page

NOTES

NUMPY (NUMERICAL PYTHON)

N-DIMENSIONAL ARRAY (NDARRAY)

Get **Data Type****	ndarray1.dtype
Explicit **Casting**	ndarray2 = ndarray1.astype (np.int32) ***

*Cannot **assume** empty () will return all zeros. It could be **garbage** values.

**Default data type is 'npfloat64'. This is equivalent to Python's float type which is 8 bytes (64 bits); thus the name 'float64'.

***If casting were to fail for some reason,'TypeError' will be raised.

SLICING (INDEXING/SUBSETTING)

> Slicing (i.e., ndarray1 [2:6]) is a **'view'** on the orginial array. **Data are NOT copied.** Any modifications (i.e., ndarray1 [2:6] = 8) to the "view" will be reflected in the original array.

> Instead of a **"view"**, explicit copy of slicing via:

```
ndarray1 [2:6] .copy ()
```

> **Multidimensional** array indexing notation:

```
ndarray1 [0] [2] or ndarray1 [0, 2]
```

***Boolean Indexing**

```
ndarray1 [ (names == 'Bob') | (names == 'Will'), 2 : ]

# '2:' means select from 3rd column on
```

*Selecting data by boolean indexing **ALWAYS** creates a copy of the data.

*The 'and' and 'or' keywords do NOT work with boolean arrays. Use & and |.

***Fancy Indexing (aka "indexing using integer arrays")**

Select a subset of rows in a particular order:

```
ndarray1 [  [3, 8, 4]  ]
ndarray1 [  [-1, 6]  ]
# negative indices select rows from the end
```

*Fancy Indexing **ALWAYS** creates a copy of the data.

Setting Data with Assignment

```
ndarray1 [ ndarray1 ] <  0 ] = 0 *
```

*If ndarray1 is two-dimensions, ndarray1 < 0 creates a two-dimensional Boolean array.

COMMON OPERATIONS

1. Transposing

> A special form of reshaping which returns a **"view"** on the underlying data without copying anything.

Continued on Next Page

NOTES

COMMON OPERATIONS (Cont.)

```
ndarray1.transpose () or
ndarray1.T or
ndarray1.swapaxes (0, 1)
```

2. Vectorized Wrappers (for functions that take scalar values)

> Slicing (i.e., ndarray1 [2:6]) is a **"view"** on the orginial array. **Data are NOT copied.** Any modifications (i.e., ndarray1 [2:6] = 8) to the "view" will be reflected in the original array.

```
ndarray1 [2:6] .copy ()
```

> math. np.sqrt () **works on only a** scalar

```
np.sqrt (seq1)
# any sequence (list, ndarray, etc.) to return a ndarray
```

3. Vectorized Expressions

> np.where (cond, x, y) is a **vectorized** version of the **expression "x if condition else y"**

```
np.where ( [True, False],  [1, 2], [2, 3])
= > ndarray (1, 3)
```

> Common Usages:


```
ndarray1.transpose () or
ndarray1.T or
ndarray1.swapaxes (0, 1)
```

2. Vectorized Wrappers (for functions that take scalar values)

> Slicing (i.e., ndarray1 [2:6]) is a **"view"** on the orginial array. **Data are NOT copied.** Any modifications (i.e. ndarray1 [2:6] = 8) to the "view" will be reflected in the original array.

```
ndarray1 [2:6] .copy ()
```

> math. np.sqrt () **works on only a** scalar

```
np.sqrt (seq1)
# any sequence (list, ndarray, etc) to return a ndarray
```

3. Vectorized Expressions

> np.where (cond, x, y) is a **vectorized** version of the **expression "x if condition else y"**

```
np.where ( [True, False],  [1, 2], [2, 3])
= > ndarray (1, 3)
```

> Common Usages:


```
np.where (matrixArray > 0, 1, -1)
= > a new array (same shape of or -1 values
```

Continued on Next Page

NOTES

NUMPY (NUMERICAL PYTHON)

COMMON OPERATIONS (Cont.)

np.where (cond, 1, 0) .argmax () *
= > Find the first **True** element

* Argmax () can be used to find the index of the maximum element. Example usage is to find the first element that has a "price > number" in an array of price data.

4. Aggregations/Reductions Methods (i.e. mean, sum, std)

Computer **mean**	ndarray1.mean () or np.mean (ndarray1)
Computer **Statistics Over** Axis*	ndarray1.mean (axis = 1) ndarray1.sum (axis = 0)

* axis = 0 means column axis, 1 is row axis.

5. Boolean Arrays Methods

Count # of '**Trues**' in **Boolean** Array	(ndarray1 > 0) .sum ()
If at Least **One** Value is '**True**'	ndarray1.any ()
If all values are '**True**'	ndarray1.all ()

Note: These methods also work with non-boolean **Arrays**, where **non-zero** elements evaluate to True.

6. Sorting

Inplace **Sorting**	ndarray1.sort ()
Return a **Sorted** Copy Instead of **Inplace**	sorted1 = np.sort (ndarray1)

7. Set Methods

Return Sorted Unique Values	np.unique (ndarray1)
Test Membership Of ndarray1 Values in [2, 3, 6]	resultBooleanArray = np.in1d (ndarray1, [2, 3, 6])

> Other set methods: intersect1d(), union1d(), setdiff1d, setxor1d()

8. Random number generation (np.random)

> Supplements the built-in **Python** random * with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions.

samples = np.random.normal
(size =(3, 3))

*Python built-in random **ONLY** samples One value at a time.

REFERENCE

Reference

Dateline strptime() String Formats		
https:\\docs.python.org/2/library/datetime.html#strftime-and-strptime-dehavor		
Directive	**Meaning**	**Example**
%a	Weekday as **locale's** abbreviated name	**Sun**, **Mon**, ..., **Sat** (en_US); **So**, **Mo**,..., **Sa** (de_DE)
%A	Weekday as locale's **full** name	**Sunday**, **Monday**, ..., **Saturday** (en_US); **Sonntag**, **Montag**,..., **Samstag** (de_DE)
%w	Weekday as a **decimal** number, where **0** is **Sunday** and **0**, **1**, ..., **6** is **Saturday**	**0**, **1**, ..., **6**
%d	Day of the month as a **zero-padded** decimal number	**01**, **02**, ..., **31**
%b	Month as locale's **abbreviated** name	**Jan**, **Feb**, ..., **Dec** (en_US); **So**, **Mo**,..., **Sa** (de_DE)

Directive	Meaning	Example
%B	Month as locale's **full** name	**January**, **February**, ..., **December** (en_US); **January**, **February**, ..., **December** (de_DE)
%m	Month as a **zero-padded** decimal number	**0**, **1**, ..., **6**
%d	Day of the month as a **zero-padded** decimal number	**01**, **02**, ..., **12**
%y	Year without **century** as a **zero-padded** decimal number	**00**, **01**, ..., **99**
%Y	Year with **century** as a **decimal** number	**1970**, **1988**, **2001**, **2013**
%H	Hour **(24-hour clock)** as a zero-padded decimal number	**00**, **01**, ..., **23**
%I	Hour **(12-hour clock)** as a zero-padded decimal number	**01**, **02**, ..., **12**
%p	Locale's equivalent of either **AM** or **PM**	**AM**, **PM** (en_US); **am**, **pm** (de_DE)
%M	Minute as a zero-padded **decimal** number	**00**, **01**, ..., **59**

Directive	Meaning	Example
%S	Second as a zero-padded decimal number	00, 01, ..., 59
%f	Microsecond as a decimal number, zero-padded on the left	000000, 000001, ..., 999999
%z	UTC offset in the form +HHMM or –HHMM (empty string if the object is naïve)	(empty), +0000, -0400, +1030
%Z	Time zone name (empty string if the object is naïve)	(empty), UTC, EST, CST
%j	Day of the year as a zero-padded decimal number	001, 002, ..., 366
%U	Week number of the year (Sunday as the first day of the week) as a zero-padded decimal number. All days in a new year preceding the first Sunday are considered to be in week 0.	00, 01, ..., 53
%W	Week number of the year (Monday as the first day of the week) as a zero-padded decimal number. All days in a new year preceding the first Monday are considered to be in week 0.	00, 01, ..., 53

Directive	Meaning	Example
%c	Locale's appropriate date and **time** representation	**Tue Aug 16** 21:30:00 **1988** (en_US); Di **16 Aug 21**:30:00 **1988** (de_DE)
%x	Locale's **appropriate** date representation	**08/16/88** (None); **08/16/1988** (en_US); **16.08.1988** (de_DE)
%X	Locale's appropriate time representation	**21:30:00** (en_US); **21:30:00** (de_DE)
%%	A literal `%' character	%

RESOURCES
ON THE COE

Resources
On the COE

- A link for downloading **Anaconda's Jupyter** Notebook (a great way to document and share code on the COE):
  http://repoman.vm.cloud.coe.ic.gov/pub/repo.continuum.io/archive/winzip/
- The **Python** resource **ADV** can be found here:
  https://appdev.proj.coe.ic.gov/adv-portal/homepage.php
- Joint enterprise **Modeling** & **Analytic** Intelligence community (JEMA-IC):
  Download & install the standalone version
  https://jema.nro.ic.gov/
- The **REGEX** (Regular Expressions) testing page:
  https://util.appdev.proj.coe.ic.gov/adv-regexp/

- Python **References** & **Resources**:

Safari Books Online - Think Python, 2nd Edition
Also is available for free as a pdf at Green Tea Press

Safari Books Online - Python in a Nutshell, 3rd Edition

Cheetsheet - Python

Cheetsheet - Regular Expressions

NGA's **Python Party** on r-Space

Python Brochure

- [Python.org] (https://www.python.org/]

  [**Python** for **Non-Programmers**]
  (https://wiki.python.org/moin/BeginnersGuide/NonProgrammers)

- **[PEP8** Style Guide for **Python]**
  (https://www.python.org/dev/peps/pep-0008/)

- [**O'Reilly's** Think **Python**: How to Think Like a **Computer Scientist**, 2nd Edition]
  (http://greenteapress.com/thinkpython2/thinkpython2pdf) - allen Downey had made this great introduction to Python 2.7 available for free at Green Tea Press. Should you decide to do the practical exercises in the book, Mr. downey has made the solutions available as Jupyter Notebooks **[here]** (http:/greenteapress.com/thinkpython/code/).

- [**Automate** the **Boring Stuff** with **Python** - Practical **Programming** for Total **Beginners**
  (https://automatetheboringstuff.com/) - "Written for office workers, students, administrators, and anyone who uses a computer to learn how to code small, practical programs to automate tasks on their computer."

- **[StackOverflow]**
  (https://stackoverflow.com/quesytions/tagged/python) - A great **resource** to find solutions for any coding problem you might have in this course and in your jobs. If you find yourself stuck with a problem during the **Practical Exercises**, think about what it is that your are trying to accomplish and ask Google. You will find that your are not to have a similar problem and there are plently of people out in the **StackOverflow** community who can help.

- **[Jupyter Notebook** (software)]
  (https://jupyter.org/) - the classroom exercises will be conducted using Jupyter Notebooks unlike the Udacity course which uses its own web interface. Jupyter Notebooks, or iPython Notebooks as they were called until recently, are a great way to create code, add documentation, and share the results with others.

- [What is **Jupyter** Notebook on **YouTube**]
  (https://video.search.yahoo.com/search/video;_ylt=A0LEVxT2upRZU94AX1tXNyoA;_ylu=X3oDMTEyNGJkYFqBGNvbG8DYmYXBHBvcwMxBHZ0aWQDQjQ0ODJfMQRzAWMDc2M?p=jupyter+notebook+tutorial&fr=dss_ysetchr#ed=8&vid=74d3ef639eeabb387d6267364dab002b&action=view)

- [Jupyter Notebook Users Manual]
  (https://athena.brynmawr.edu/jupyter/hub/dblandk/public/Jupyter%20Notebook%20Users%20Manual.ipynb)

Other Resources

Python Datetime
https://docs.python.org/2/library/datetime.html

Python Re
https://docs.python.org/2.7/library/re.html

Regular Expressions
http://regexr.com/

GeoJSON
http://geojson.org/

Cesium.js
https://cesiumjs.org/

NOTES

HDN

National Geospatial – Intelligence College