Python for Data Analysis and Visualization

Python

- Very popular general-purpose programming language
- Used from introductory programming courses to production systems

Python

- Open source general-purpose language.
- Object Oriented, Procedural, Functional
- Easy to interface with C/ObjC/Java/Fortran
- Easy-ish to interface with C++ (via SWIG)
- Great interactive environment

Python Features

- Dynamically typed

(rather than statically typed like Java or C/C++)

- Interpreted

(rather than compiled like Java or C/C++)

Python programs are comparatively…

+ Quicker to write
+ Shorter
– More error-prone
– Slower to run

A Code Sample

```
x = 34 -23  # A comment
y = "Hello"  # Another one
z = 3.45
if z == 3.45 or y == "Hello":
    x = x+1
    y = y+"world"  # String concat.
print(x)
print(y)
```

Enough to Understand the Code

- Assignment uses **=** and comparison uses **==**.
- For numbers **+ - * / %** are as expected.
  - Special use of **+** for string concatenation.
  - Special use of **%** for string formatting (as with printf in C)
- Logical operators are words (**and, or, not**) not symbols
- The basic printing command is **print**.
- The first assignment to a variable creates it.
  - Variable types don't need to be declared.
  - Python figures out the variable types on its own.

Basic Datatypes

- Integers (default for numbers)

  z = 5 / 2 # Answer is 2, integer division.

- Floats

  x = 3.456

- Strings
  - Can use “” or ‘’ to specify.

    “abc” ‘abc’ (Same thing.)
  - Unmatched can occur within the string.

    “matt’s”
  - Use triple double-quotes for multi-line strings or strings than contain both ‘ and “ inside of them:

    “””a‘b“c”””

Whitespace

Whitespace is meaningful in Python: especially indentation and placement of newlines.

- Use a newline to end a line of code.
  - Use **** when must go to next line prematurely.
- No braces **{ }** to mark blocks of code in Python… Use consistent indentation instead.
  - The first line with less indentation is outside of the block.
  - The first line with more indentation starts a nested block
- Often a colon appears at the start of a new block. (E.g. for function and class definitions.)

Assignment

- Binding a variable in Python means setting a name to hold a reference to some object.
  - Assignment creates references, not copies
- Names in Python do not have an intrinsic type. Objects have types.
  - Python determines the type of the reference automatically based on the data object assigned to it.
- You create a name the first time it appears on the left side of an assignment expression:

  x = 3

- A reference is deleted via garbage collection after any names bound to it have passed out of scope.

Accessing Non-Existent Names

- If you try to access a name before it's been properly created (by placing it on the left side of an assignment), you'll get an error.

```
>>> y

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    y
NameError: name 'y' is not defined
>>> y = 3
>>> y
3
```

Multiple Assignment

- You can also assign to multiple names at the same time.

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```

Naming Rules

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

  bob Bob _bob _2_bob_ bob_2 BoB

- There are some reserved words:

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

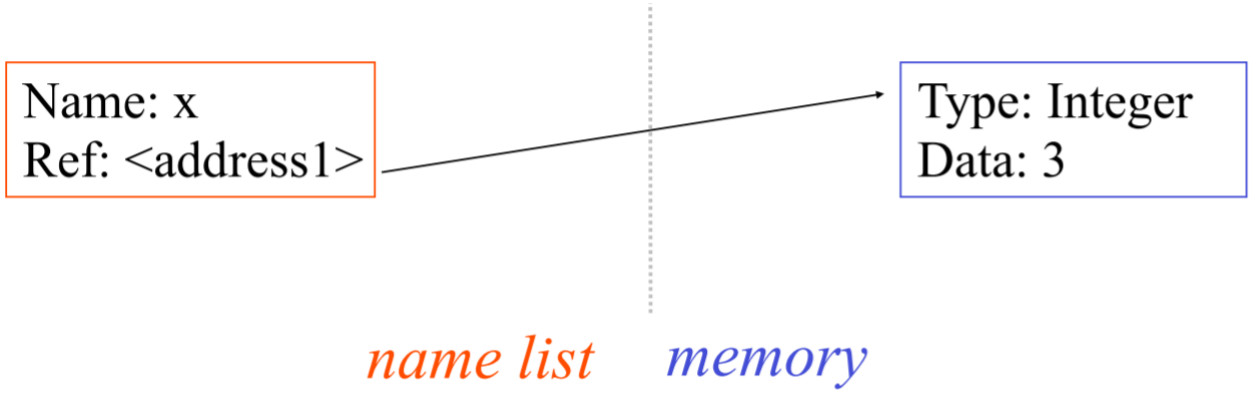
Understanding Reference Semantics

- Assignment manipulates references
  - x = y does not make a copy of the object y references
  - x = y makes x reference the object y references
- Very useful; but beware!
- Example:

```
>>> a = [1, 2, 3] # a now references the list [1, 2, 3]
>>> b = a # b now references what a references
 >>> a.append(4) # this changes the list a references
>>> print b # if we print what b references,
[1, 2, 3, 4] # SURPRISE! It has changed…
 Why??
```

Understanding Reference Semantics

- There is a lot going on when we type: x = 3
- First, an integer 3 is created and stored in memory
- A name x is created
- An reference to the memory location storing the 3 is then assigned to the name x
- So: When we say that the value of x is 3
- we mean that x now refers to the integer 3



Understanding Reference Semantics

- The data 3 we created is of type integer. In Python, the datatypes integer, float, and string (and tuple) are “immutable.”
- This doesn’t mean we can’t change the value of x, i.e. change what x refers to …
- For example, we could increment x:

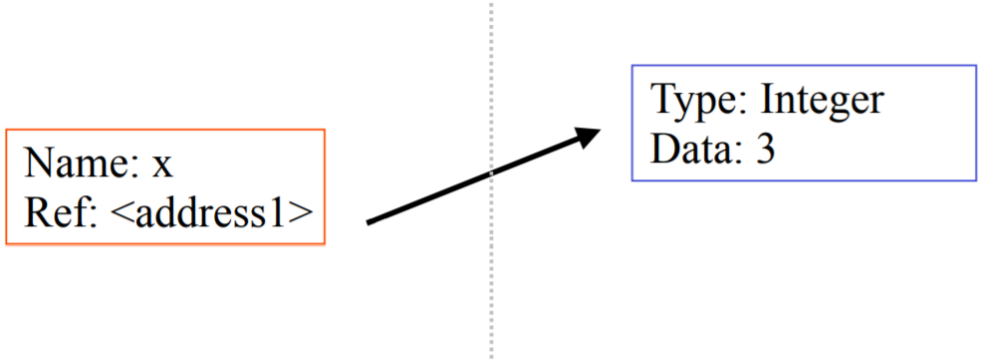
```
>>> x = 3
>>> x = x + 1
>>> print x
4
```

Understanding Reference Semantics

- If we increment x, then what's really happening is:

1. The reference of name x is looked up.
2. The value at that reference is retrieved.

```
>>> x = x + 1
```



Understanding Reference Semantics

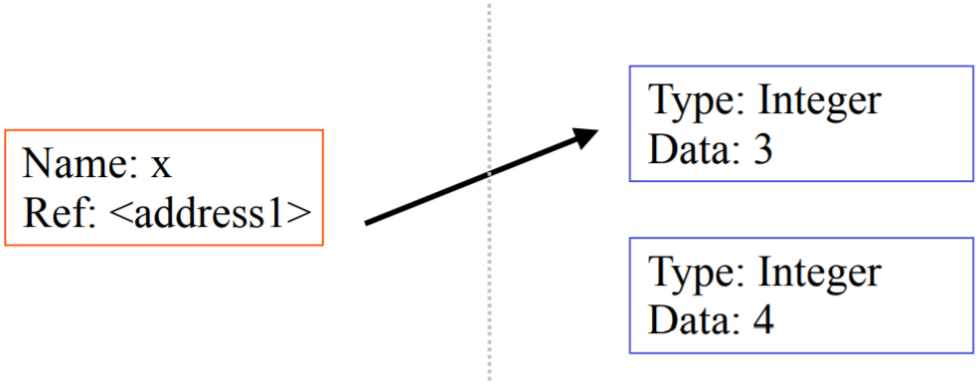
- If we increment x, then what's really happening is:

1. The reference of name x is looked up.

```
>>> x = x + 1
```

2. The value at that reference is retrieved.

3. The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.



Understanding Reference Semantics

- If we increment x, then what's really happening is:

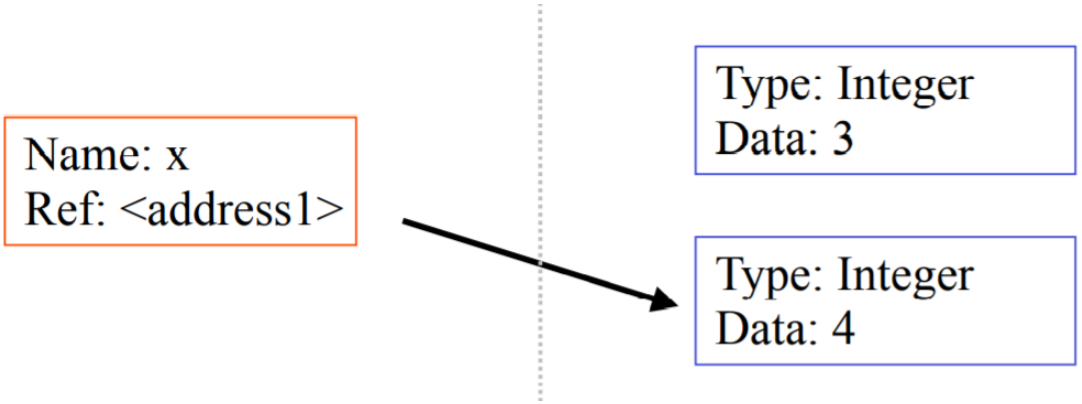
1. The reference of name x is looked up.

```
>>> x = x + 1
```

2. The value at that reference is retrieved.

3. The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.

4. The name x is changed to point to this new reference.



Understanding Reference Semantics

- If we increment x, then what's really happening is:

1. The reference of name x is looked up.

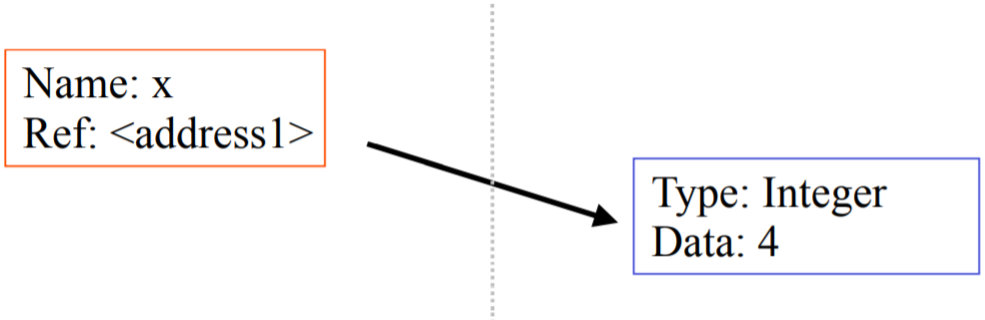
```
>>> x = x + 1
```

2. The value at that reference is retrieved.

3. The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.

4. The name x is changed to point to this new reference.

5. The old data 3 is garbage collected if no name still refers to it.



Sequence Types 1

- Tuples are defined using parentheses (and commas).

```
>>> tu = (23, ‘abc’, 4.56, (2,3), ‘def’)
```

- Lists are defined using square brackets (and commas).

```
>>> li = [“abc”, 34, 4.34, 23]
```

- Strings are defined using quotes (“, ‘, or “““).

```
>>> st = “Hello World”
>>> st = ‘Hello World’
>>> st = “““This is a multi-line string that uses triple quotes.”””
```

Sequence Types 2

- We can access individual members of a tuple, list, or string using square bracket “array” notation.
- Note that all are 0 based…

```
>>> tu = (23, ‘abc’, 4.56, (2,3), ‘def’)
>>> tu[1] # Second item in the tuple.
‘abc’
>>> li = [“abc”, 34, 4.34, 23]
>>> li[1] # Second item in the list.
34
>>> st = “Hello World”
>>> st[1] # Second character in string.
‘e’
```

Positive and negative indices

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0.

```
>>> t[1]
'abc'
```

Negative lookup: count from right, starting with –1.

```
>>> t[-3]
4.56
```

Slicing: Return Copy of a Subset 1

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying before the second index.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

You can also use negative indices when slicing.

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

Slicing: Return Copy of a Subset 2

```
>>> t = (23, ‘abc’, 4.56, (2,3), ‘def’)
```

Omit the first index to make a copy starting from the beginning of the container.

```
>>> t[:2] (23, ‘abc’)
```

Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> t[2:]
(4.56, (2,3), ‘def’)
```

The ‘in’ Operator

- Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- For strings, tests for substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- Be careful: the in keyword is also used in the syntax of for loops and list comprehensions

The + Operator

- The + operator produces a new tuple, list, or string whose value is the concatenation of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> “Hello” + “ ” + “World”
‘Hello World’
```

The * Operator

- The * operator produces a new tuple, list, or string that “repeats” the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> “Hello” * 3
‘HelloHelloHello’
```

Comments

- Start comments with # – the rest of line is ignored.
- Can include a “documentation string” as the first line of any new function or class that you define.
- The development environment, debugger, and other tools use it: it’s good style to include one.

```
def my_function(x, y):
“””This is the docstring. This function does blah blah blah.”””
 # The code would go here...
```

Python for Data

- Fairly easy to read/write/process data using standard features
- Plus special packages for…
  - Numerical and statistical manipulations - numpy
  - Visualization (“plotting”) - matplotlib
  - Relational database like capabilities – pandas
  - Machine learning - scikit-learn
  - Network analysis - networkx
  - Unstructured data – re, nltk, PIL

Python Libraries for Data Science

Many popular Python toolboxes/libraries:

- NumPy
- SciPy
- Pandas
- SciKit-Learn

Visualization libraries

- matplotlib
- Seaborn

Python Libraries for Data Science

NumPy:

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

Python Libraries for Data Science

Pandas:

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

Python Libraries for Data Science

SciPy:

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

Python Libraries for Data Science

SciKit-Learn:

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

Python Libraries for Data Science

matplotlib:

- python 2D plotting library which produces publication quality figures in a variety of hardcopy formats
- a set of functionalities similar to those of MATLAB

- line plots, scatter plots, barcharts, histograms, pie charts etc.
- relatively low-level; some effort needed to create advanced visualization

Python Libraries for Data Science

Seaborn:

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

Loading Python Libraries

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

Press Shift+Enter to execute the jupyter cell

Reading data using pandas

```
In [ ]: #Read csv file
        df = pd.read_csv("http://rcs.bu.edu/examples/python/data_analysis/Salaries.csv")
```

Note: The above command has many optional arguments to fine-tune the data import process.

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

```
pd.read_excel('myfile.xlsx',sheet_name='Sheet1', index_col=None, na_values=['NA'])

pd.read_stata('myfile.dta')

pd.read_sas('myfile.sas7bdat')

pd.read_hdf('myfile.h5','df')
```

Exploring data frames

In [3]:
```
#List first 5 records
df.head()
```

Out[3]:

	rank	discipline	phd	service	sex	salary
0	Prof	B	56	49	Male	186960
1	Prof	A	12	6	Male	93000
2	Prof	A	23	20	Male	110515
3	Prof	A	40	31	Male	131205
4	Prof	B	20	18	Male	104800

Data Frame data types

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

Data Frame data types

```
In [4]: #Check a particular column type
        df['salary'].dtype
```

```
Out[4]: dtype('int64')
```

```
In [5]: #Check types for all the columns
        df.dtypes
```

```
Out[4]: rank          object
        discipline    object
        phd           int64
        service       int64
        sex           object
        salary        int64
        dtype: object
```

Data Frames attributes

Python objects have attributes and methods

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

Data Frames methods

Unlike attributes, python methods have parenthesis.

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

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

Selecting a column in a Data Frame

Method 1: Subset the data frame using column name: df['sex']

Method 2: Use the column name as an attribute: df.sex

Note: there is an attribute rank for pandas data frames, so to select a column with a name "rank" we should use method 1.

Data Frames groupby method

Using "group by" method we can:

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

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

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

rank	phd	service	salary
AssocProf	15.076923	11.307692	91786.230769
AsstProf	5.052632	2.210526	81362.789474
Prof	27.065217	21.413043	123624.804348

Data Frames groupby method

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

```
In [ ]: #Calculate mean salary for each professor rank:
        df.groupby('rank')[['salary']].mean()
```

	salary
rank	
AssocProf	91786.230769
AsstProf	81362.789474
Prof	123624.804348

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

Data Frames groupby method

groupby performance notes:

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

```
In [ ]: #Calculate mean salary for each professor rank:
        df.groupby(['rank'], sort=False)[['salary']].mean()
```

Data Frame: filtering

To subset the data we can apply Boolean indexing. This indexing is commonly known as a filter. For example if we want to subset the rows in which the salary value is greater than $120K:

```
In [ ]: #Calculate mean salary for each professor rank:
        df_sub = df[ df['salary'] > 120000 ]
```

Any Boolean operator can be used to subset the data:

> greater; >= greater or equal;

< less; <= less or equal;

== equal; != not equal;

```
In [ ]: #Select only those rows that contain female professors:
        df_f = df[ df['sex'] == 'Female' ]
```

Data Frames: Slicing

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

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

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

```
In [ ]: #Select column salary:
        df[['rank','salary']]
```

Data Frames: Selecting rows

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

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

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

Data Frames: method loc

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

```
In [ ]: #Select rows by their labels:
        df_sub.loc[10:20,['rank','sex','salary']]
```

Out[ ]:

	rank	sex	salary
10	Prof	Male	128250
11	Prof	Male	134778
13	Prof	Male	162200
14	Prof	Male	153750
15	Prof	Male	150480
19	Prof	Male	150500

Data Frames: method iloc

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

In [ ]:
```
#Select rows by their labels:
df_sub.iloc[10:20,[0, 3, 4, 5]]
```

Out[ ]:

	rank	service	sex	salary
26	Prof	19	Male	148750
27	Prof	43	Male	155865
29	Prof	20	Male	123683
31	Prof	21	Male	155750
35	Prof	23	Male	126933
36	Prof	45	Male	146856
39	Prof	18	Female	129000
40	Prof	36	Female	137000
44	Prof	19	Female	151768
45	Prof	25	Female	140096

Data Frames: method iloc (summary)

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

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

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

Data Frames: Sorting

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

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

Out[ ]:

	rank	discipline	phd	service	sex	salary
55	AsstProf	A	2	0	Female	72500
23	AsstProf	A	2	0	Male	85000
43	AsstProf	B	5	0	Female	77000
17	AsstProf	B	4	0	Male	92000
12	AsstProf	B	1	0	Male	88000

Data Frames: Sorting

We can sort the data using 2 or more columns:

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

Out[ ]:

	rank	discipline	phd	service	sex	salary
52	Prof	A	12	0	Female	105000
17	AsstProf	B	4	0	Male	92000
12	AsstProf	B	1	0	Male	88000
23	AsstProf	A	2	0	Male	85000
43	AsstProf	B	5	0	Female	77000
55	AsstProf	A	2	0	Female	72500
57	AsstProf	A	3	1	Female	72500
28	AsstProf	B	7	2	Male	91300
42	AsstProf	B	4	2	Female	80225
68	AsstProf	A	4	2	Female	77500

Missing Values

Missing values are marked as NaN

In [ ]:
```
# Read a dataset with missing values
flights = pd.read_csv("http://rcs.bu.edu/examples/python/data_analysis/flights.csv")
```

In [ ]:
```
# Select the rows that have at least one missing value
flights[flights.isnull().any(axis=1)].head()
```

Out[ ]:

	year	month	day	dep_time	dep_delay	arr_time	arr_delay	carrier	tailnum	flight	origin	dest	air_time	distance	hour	minute
330	2013	1	1	1807.0	29.0	2251.0	NaN	UA	N31412	1228	EWR	SAN	NaN	2425	18.0	7.0
403	2013	1	1	NaN	NaN	NaN	NaN	AA	N3EHAA	791	LGA	DFW	NaN	1389	NaN	NaN
404	2013	1	1	NaN	NaN	NaN	NaN	AA	N3EVAA	1925	LGA	MIA	NaN	1096	NaN	NaN
855	2013	1	2	2145.0	16.0	NaN	NaN	UA	N12221	1299	EWR	RSW	NaN	1068	21.0	45.0
858	2013	1	2	NaN	NaN	NaN	NaN	AA	NaN	133	JFK	LAX	NaN	2475	NaN	NaN

Missing Values

There are a number of methods to deal with missing values in the data frame:

df.method()	description
dropna()	Drop missing observations
dropna(how='all')	Drop observations where all cells is NA
dropna(axis=1, how='all')	Drop column if all the values are missing
dropna(thresh = 5)	Drop rows that contain less than 5 non-missing values
fillna(0)	Replace missing values with zeros
isnull()	returns True if the value is missing
notnull()	Returns True for non-missing values

Missing Values

- When summing the data, missing values will be treated as zero
- If all values are missing, the sum will be equal to NaN
- cumsum() and cumprod() methods ignore missing values but preserve them in the resulting arrays
- Missing values in GroupBy method are excluded (just like in R)
- Many descriptive statistics methods have skipna option to control if missing data should be excluded . This value is set to True by default (unlike R)

Aggregation Functions in Pandas

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

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

Common aggregation functions:

- min, max
- count, sum, prod
- mean, median, mode, mad
- std, var

Aggregation Functions in Pandas

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

```
In [ ]: flights[['dep_delay','arr_delay']].agg(['min','mean','max'])
```

Out[ ]:

	dep_delay	arr_delay
min	-16.000000	-62.000000
mean	9.384302	2.298675
max	351.000000	389.000000

Basic Descriptive Statistics

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

Graphics to explore the data

- Seaborn package is built on matplotlib but provides high level interface for drawing attractive statistical graphics, similar to ggplot2 library in R. It specifically targets statistical data visualization
- To show graphs within Python notebook include inline directive:

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

Graphics

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

Data Frames: Slicing

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

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

Rows and columns can be selected by their position or label

What We'll Cover

1. Python basics
2. Data manipulation
3. Plotting
4. Pandas

(more in later topics)

For help while working with Python:

- Tutorials and help pages (website)
- Web search