



# Exam II Review

---

IOE 373



# Exam Info

---

- Online Exam on Canvas

- Timed 85 minute exam once you open it on Canvas. You'll be able to take the exam at any time within 48 hr after publication. Exam will open on **Thursday Nov 30 12PM (noon) EST**. Exam will close on **Saturday 12/02 12PM (noon) EST**.
- The College of Engineering Honor Code applies and should be followed (please complete the honor pledge when you finish your test).
- **Exam material should not be posted/discussed publicly or shared in any way (including Piazza Board)**

- Exam recommendations:

- Prepare a note sheet.
- Have a notebook/blank paper for annotations and in case you need to submit them for re-grading.
- Have a simple scientific calculator



# Exam Format

---

- Section I, combination of true/false and multiple choice questions (30 points)
- Section II, 2 open ended questions – conceptual (20 points)
- Section III, 4-5 questions, you'll be asked to write code for queries based on the statements/methods covered in class (and HW) or you may be given code and a worksheet and may be asked to show the outcome (50 points)



# Study Guide (Lec 13 – ML Summary Linear Rgression)

---

- Python Basics
- NumPy
- Pandas
- EDA/Visualization
- Multiple Linear Regression



# Python

---

- Developed by Guido Van Rossum
- General purpose, open source and free
- Very popular with data scientists
- Many libraries (NumPy, SciPy, Matplotlib, Pandas, etc.)
- Dictionaries – enable fast database-like operations
  - Great for text analysis (e.g. social media)



# Interactive versus Script

---

- Interactive
  - - You type directly to Python one line at a time and it responds
- Script
  - - You enter a sequence of statements (lines) into a file using a text editor and tell Python to execute the statements in the file



# Type Conversions

---

- When you put an integer and floating point in an expression, the integer is implicitly converted to a float
- You can control this with the built-in functions `int()` and `float()`

```
>>> print(float(99) + 100)
199.0
>>> i = 42
>>> type(i)
<class'int'>
>>> f = float(i)
>>> print(f)
42.0
>>> type(f)
<class'float'>
>>>
```



# String Conversions

- You can also use `int()` and `float()` to convert between strings and integers
- You will get an error if the string does not contain numeric characters

```
>>> sval = '123'
>>> type(sval)
<class 'str'>
>>> print(sval + 1)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object
to str implicitly
>>> ival = int(sval)
>>> type(ival)
<class 'int'>
>>> print(ival + 1)
124
```

```
>>> nsv = 'hello bob'
>>> niv = int(nsv)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int()
with base 10: 'x'
```





# User Input

---

- We can instruct Python to pause and read data from the user using the `input()` function
- The `input()` function returns a string

```
nam = input('Who are you? ')\nprint('Welcome', nam)
```

```
Who are you? Luis\nWelcome Luis
```



# Comparison Operators

- Boolean expressions ask a question and produce a Yes or No result which we use to control program flow
- Boolean expressions using comparison operators evaluate to True / False or Yes / No
- Comparison operators look at variables but do not change the variables

Python	Meaning
<	Less than
<=	Less than or Equal to
==	Equal to
>=	Greater than or Equal to
>	Greater than
!=	Not equal



# Indentation

---

- Increase indent after an if statement or for statement (after : )
- Maintain indent to indicate the scope of the block (which lines are affected by the if/for)
- Reduce indent back to the level of the if statement or for statement to indicate the end of the block
- Blank lines are ignored - they do not affect indentation
- Comments on a line by themselves are ignored with regard to indentation



# Multi-way

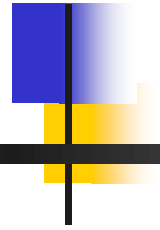
---

```
# No Else
x = 5
if x < 2 :
    print('Small')
elif x < 10 :
    print('Medium')

print('All done')
```

```
if x < 2 :
    print('Small')
elif x < 10 :
    print('Medium')
elif x < 20 :
    print('Big')
elif x < 40 :
    print('Large')
elif x < 100:
    print('Huge')
else :
    print('Ginormous')
```

# Sample try / except – Entry Validation



```
rawstr = input('Enter a number:')
try:
    ival = int(rawstr)
except:
    ival = -1

if ival > 0 :
    print('Nice work')
else:
    print('Not a number')
```

```
$ python3 trynum.py
Enter a number:42
Nice work
```

```
$ python3 trynum.py
Enter a number:forty-two
Not a number
$
```



# Functions

---

- Built-in Functions or Custom Functions
- For list of built-in Python functions:
  - <https://docs.python.org/2/library/functions.html>
- Some examples:
  - `max()`, `min()`, `str()`, `int()`, `float()`
  - `range()`, `len()`, `abs()`, `pow()`
  - `sum()`
  - `open()`

# Custom Functions

- A custom function to convert string data to float data

```
8 def check_data(data_in):  
9     try:  
10         data_in = float(data_in)  
11     except:  
12         print "Cannot convert to numerical data"  
13     return data_in
```

Function declaration

Passed parameter

Indentation

Return statement



# Loops

---

- for-loop Structure

- Known number of iterations through loop

```
for i in [0, 1, 2]:  
    print i
```

Performs the indented steps 3 times  
with, sequentially, i=0, i=1, i=2

- [0, 1, 2] is a list

- We'll discuss this in more detail later





# Lists

---

- Another version of range() function
  - range(i,j) creates a list with integer elements starting at i and ending with j-1



# Breaking Out of a Loop

---

- The `break` statement ends the current loop and jumps to the statement immediately following the loop
- It is like a loop test that can happen anywhere in the body of the loop

```
while True:
    line = input('> ')
    if line == 'done' :
        break
    print line
print 'Done!'
```

```
> hello there
hello there
> finished
finished
> done
Done!
```

# Finishing an Iteration with continue

- The continue statement ends the current iteration and jumps to the top of the loop and starts the next iteration

```
while True:
    line = raw_input('> ')
    if line[0] == '#' :
        continue
    if line == 'done' :
        break
    print line
print 'Done!'
```

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```



# Looping and Counting

---

- This is a simple loop that loops through each letter in a string and counts the number of times the loop encounters the 'a' character

```
word = 'banana'
count = 0
for letter in word :
    if letter == 'a' :
        count = count + 1
print count
```



# Slicing Strings

---

- We can also look at any continuous section of a string using a colon operator
- The second number is one beyond the end of the slice - “up to but not including”
- If the second number is beyond the end of the string, it stops at the end

M	o	n	t	y		P	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11

```
>>> s = 'Monty Python'
>>> print s[0:4]
Mont
>>> print s[6:7]
P
>>> print s[6:20]
Python
```

# Slicing Strings

- If we leave off the first number or the last number of the slice, it is assumed to be the beginning or end of the string respectively

M	o	n	t	y		P	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11

```
>>> s = 'Monty Python'
>>> print s[:2]
Mo
>>> print s[8:]
thon
>>> print s[:]
Monty Python
```



# Slicing Strings - Reversing

---

- We can reverse strings using slicing

M	o	n	t	y		P	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11

```
>>> s = 'Monty Python'  
>>> print s[::-1]  
nohtyp ytnoM
```



# String Concatenation

---

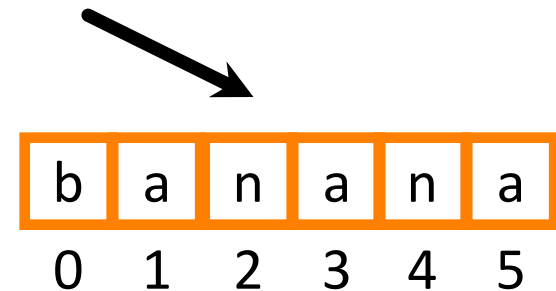
- When the `+` operator is applied to strings, it means “concatenation”

```
>>> a = 'Hello'
>>> b = a + 'There'
>>> print b
HelloThere
>>> c = a + ' ' + 'There'
>>> print c
Hello There
>>>
```



# Searching a String

- We use the `find()` function to search for a substring within another string
- `find()` finds the first occurrence of the substring
- If the substring is not found, `find()` returns -1
- Remember that string position starts at zero



```
>>> fruit = 'banana'
>>> pos = fruit.find('na')
>>> print pos
2
>>> aa = fruit.find('z')
>>> print aa
-1
```



# Stripping Whitespace

---

- Sometimes we want to take a string and remove whitespace at the beginning and/or end
- `lstrip()` and `rstrip()` remove whitespace at the left or right
- `strip()` removes both beginning and ending whitespace

```
>>> greet = '    Hello Bob    '  
>>> greet.lstrip()  
'Hello Bob    '  
>>> greet.rstrip()  
'    Hello Bob'  
>>> greet.strip()  
'Hello Bob'  
>>>
```



# Prefixes - startswith

---

```
>>> line = 'Please have a nice day'
>>> line.startswith('Please')
True
>>> line.startswith('p')
False
```



# Parsing and Extracting

21



31



From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print atpos
21
>>> sppos = data.find(' ', atpos)
>>> print sppos
31
>>> host = data[atpos+1 : sppos]
>>> print host
uct.ac.za
```



# Using open()

---

- `handle = open(filename, mode)`
  - returns a handle use to manipulate the file
  - filename is a string
  - mode is optional and should be 'r' if we are planning to read the file and 'w' if we are going to write to the file

```
fhand = open('mbox.txt', 'r')
```



# Counting Lines in a File

---

- Open a file read-only
- Use a for loop to read each line
- Count the lines and print out the number of lines

```
fhand = open('mbox.txt')
count = 0
for line in fhand:
    count = count + 1
print 'Line Count:', count
```

```
$ python open.py
Line Count: 132045
```



# Searching Through a File

---

- We can put an **if** statement in our for loop to only print lines that meet some criteria

```
fhand = open('mbox-short.txt')
for line in fhand:
    if line.startswith('From:') :
        print line
```



# Searching Through a File (fixed)

---

- We can strip the whitespace from the right-hand side of the string using **rstrip()** from the string library
- The newline is considered “white space” and is stripped

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.startswith('From:') :
        print line
```

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
....
```





# Skipping with continue

---

- We can conveniently skip a line by using the continue statement

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From:') :
        continue ←
    print line
```



# Delimiters

---

When you do not specify a delimiter, multiple spaces are treated like *one* delimiter -  
You can specify what delimiter character to use in the splitting

```
>>> line = 'A lot                of spaces'
>>> etc = line.split()
>>> print etc
['A', 'lot', 'of', 'spaces']
>>>
>>> line = 'first;second;third'
>>> thing = line.split()
>>> print thing
['first;second;third']
>>> print len(thing)
1
>>> thing = line.split(';')
>>> print thing
['first', 'second', 'third']
>>> print len(thing)
3
>>>
```



# Example

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From ') : continue
    words = line.split()
    print words[2]
```

Sat  
Fri  
Fri  
Fri  
...

```
>>> line = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> words = line.split()
>>> print words
['From', 'stephen.marquard@uct.ac.za', 'Sat', 'Jan', '5', '09:14:16', '2008']
>>>
```



# Dictionaries

---

- Dictionaries are Python's most powerful data collection
- Dictionaries allow us to do fast database-like operations in Python
- Dictionaries have different names in different languages
  - Associative Arrays - Perl / PHP
  - Properties or Map or HashMap - Java
  - Property Bag - C# / .Net

[http://en.wikipedia.org/wiki/Associative\\_array](http://en.wikipedia.org/wiki/Associative_array)



# When we see a new name

---

- When we encounter a new name, we need to add a new entry in the dictionary and if this is the second or later time we have seen the name, we simply add one to the count in the dictionary under that name

```
counts = dict()
names = ['csev', 'cwen', 'csev', 'zqian', 'cwen']
for name in names :
    if name not in counts:
        counts[name] = 1
    else :
        counts[name] = counts[name] + 1
print counts
```

```
{'csev': 2, 'zqian': 1, 'cwen': 2}
```




# Definite Loops and Dictionaries

---

- Even though dictionaries are not stored in order, we can write a **for** loop that goes through all the entries in a dictionary - actually it goes through all of the keys in the dictionary and looks up the values

```
>>> counts = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> for key in counts:
...     print key, counts[key]
...
jan 100
chuck 1
fred 42
>>>
```



```
name = raw_input('Enter file:')
handle = open(name)
text = handle.read()
words = text.split()

counts = dict()
for word in words:
    counts[word] = counts.get(word,0) + 1

bigcount = None
bigword = None
for word,count in counts.items():
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print bigword, bigcount
```

```
python words.py
Enter file: words.txt
to 16
```

```
python words.py
Enter file: clown.txt
the 7
```



# Regular Expression Quick Guide

---

<code>^</code>	Matches the beginning of a line
<code>\$</code>	Matches the end of the line
<code>.</code>	Matches any character
<code>\s</code>	Matches whitespace
<code>\S</code>	Matches any non-whitespace character
<code>*</code>	Repeats a character zero or more times
<code>*?</code>	Repeats a character zero or more times (non-greedy)
<code>+</code>	Repeats a character one or more times
<code>+?</code>	Repeats a character one or more times (non-greedy)
<code>[aeiou]</code>	Matches a single character in the listed set
<code>[^XYZ]</code>	Matches a single character not in the listed set
<code>[a-z0-9]</code>	The set of characters can include a range
<code>(</code>	Indicates where string extraction is to start
<code>)</code>	Indicates where string extraction is to end



# Matching and Extracting Data

`re.search()` returns a True/False depending on whether the string matches the regular expression

If we actually want the matching strings to be extracted, we use `re.findall()`

`[0-9]+`



One or more digits

```
>>> import re
>>> x = 'My 2 favorite numbers are 19 and 42'
>>> y = re.findall('[0-9]+',x)
>>> print(y)
['2', '19', '42']
```



# Tuples Are Like Lists

- Tuples are another kind of sequence that functions much like a list
  - They have elements which are indexed starting at 0
  - A tuple is a comma-separated list of values. (it is common to enclose tuples in parentheses to help us quickly identify tuples when we look at Python code)

```
>>> x = ('Glenn', 'Sally', 'Joseph')
```

```
>>> print(x[2])
```

```
Joseph
```

```
>>> y = 1, 9, 2
```

```
>>> print(y)
```

```
(1, 9, 2)
```

```
>>> print(max(y))
```

```
9
```

```
>>> for iter in y:  
...     print(iter)
```

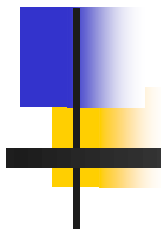
```
...
```

```
1
```

```
9
```

```
2
```

```
>>>
```

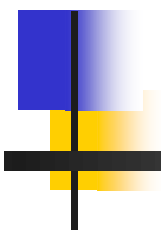


# Lists vs Tuples

---

```
>>> l = list()
>>> dir(l)
['append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

```
>>> t = tuple()
>>> dir(t)
['count', 'index']
```



# Tuples are More Efficient

---

- Since tuple structures are not modifiable, they are simpler and more efficient in terms of memory use and performance than lists
- If you need temporary variables, use tuples instead of lists, for more efficient and quicker scripts!



# Tuples and Assignment

---

We can also put a tuple on the left-hand side of an assignment statement

We can even omit the parentheses

```
>>> (x, y) = (4, 'fred')
>>> print(y)
fred
>>> (a, b) = (99, 98)
>>> print(a)
99
```



# Tuples and Dictionaries

---

The items() method  
in dictionaries  
returns a list of (key,  
value) tuples

```
>>> d = dict()
>>> d['csev'] = 2
>>> d['cwen'] = 4
>>> for (k,v) in d.items():
...     print(k, v)
...
csev 2
cwen 4
>>> tups = d.items()
>>> print(tups)
dict_items([('csev', 2), ('cwen', 4)])
```



# Tuples are Comparable

---

The comparison operators work with tuples and other sequences. If the first item is equal, Python goes on to the next element, and so on, until it finds elements that differ.

```
>>> (0, 1, 2) < (5, 1, 2)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
>>> ( 'Jones', 'Sally' ) < ( 'Jones', 'Sam' )
True
>>> ( 'Jones', 'Sally' ) > ( 'Adams', 'Sam' )
True
```



# Numpy

---

- Numpy is the core library for scientific computing in Python.
- It provides a high-performance multidimensional array object, and tools for working with these arrays.





# Arrays

---

- A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers.
- The number of dimensions is the rank of the array
- The shape of an array is a tuple of integers giving the size of the array along each dimension.



# Arrays

---

- We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

```
import numpy as np
```

```
a = np.array([1, 2, 3])    # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"
```

```
b = np.array([[1,2,3],[4,5,6]]) # Create a rank 2 array
print(b.shape)                 # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0]) # Prints "1 2 4"
```

# Array Indexing

- **Slicing:** Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```
import numpy as np
```

```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
```

Create the following rank 2 array  
with shape (3, 4)

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

```
b = a[:2, 1:3]
```

# Use slicing to pull out the subarray  
consisting of the first 2 rows  
# and columns 1 and 2; b is the  
following array of shape (2, 2):

```
# [[2 3]
#  [6 7]]
```

```
# A slice of an array is a view into the same data, so modifying it  
# will modify the original array.
```

```
print(a[0, 1])    # Prints "2"  
b[0, 0] = 77      # b[0, 0] is the same piece of data as a[0, 1]  
print(a[0, 1])    # Prints "77"
```

# Array Math

```
import numpy as np
```

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)
print(x * y)
print(np.multiply(x, y))
```

```
# Elementwise product; both produce the
array
# [[ 5.0 12.0]
#  [21.0 32.0]]
```

```
print(x / y)
print(np.divide(x, y))
```

```
# Elementwise division; both produce the
array
# [[ 0.2      0.33333333]
#  [ 0.42857143  0.5      ]]
```

```
print(x + y)
print(np.add(x, y))
```

```
# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
```

```
print(x - y)
print(np.subtract(x, y))
```

```
# Elementwise difference; both produce the
array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
```

```
print(np.sqrt(x))
```

```
# Elementwise square root; produces the
array
# [[ 1.      1.41421356]
#  [ 1.73205081  2.      ]]
```



# Matrix/Vector Operations

- Use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices.

```
import numpy as np
```

```
x = np.array([[1,2],[3,4]])  
y = np.array([[5,6],[7,8]])
```

```
v = np.array([9,10])  
w = np.array([11, 12])
```

```
# Inner product of vectors; both produce 219  
print(v.dot(w))  
print(np.dot(v, w))
```

```
# Matrix / vector product; both produce  
the rank 1 array [29 67]  
print(x.dot(v))  
print(np.dot(x, v))
```

```
# Matrix / matrix product; both produce  
the rank 2 array  
# [[19 22]  
# [43 50]]  
print(x.dot(y))  
print(np.dot(x, y))
```

# Computations/Statistics on Arrays

- Full list: <https://numpy.org/doc/stable/reference/routines.math.html>

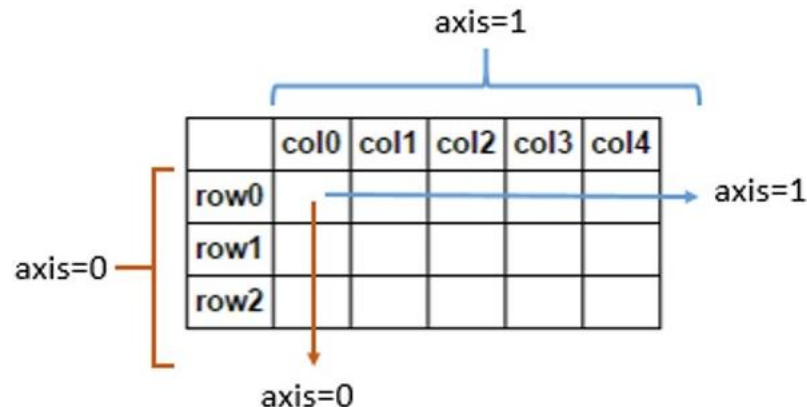
```
import numpy as np
```

```
x = np.array([[1,2],[3,4]])
```

```
print(np.sum(x)) # Compute sum of all elements; prints "10"
```

```
print(np.sum(x, axis=0)) # Compute sum of each column; prints "[4 6]"
```

```
print(np.sum(x, axis=1)) # Compute sum of each row; prints "[3 7]"
```





# Array Transpose

---

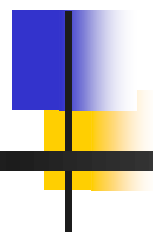
```
import numpy as np
```

```
x = np.array([[1,2], [3,4]])  
print(x)           # Prints "[[1 2]  
                   #       [3 4]]"
```

```
print(x.T)         # Prints "[[1 3]  
                   #       [2 4]]"
```

# Note that taking the transpose of a rank 1 array does nothing:

```
v = np.array([1,2,3])  
print(v)  # Prints "[1 2 3]"  
print(v.T) # Prints "[1 2 3]"
```



# What is “Pandas”?

---

- Pandas is an open source library built on NumPy
- Allows data cleaning, preparation and fast analysis
- It has built-in visualization features
- Can work with data from a variety of sources





# Series

---

- A Series is a one-dimensional array-like object containing an array of data (of any NumPy data type) and an associated array of data labels, called its index. The simplest Series is formed from only an array of data:

```
In [3]: import pandas as pd
```

```
In [4]: obj = pd.Series([4, 7, -5, 3])
```

```
In [5]: obj
```

```
Out[5]:
```

```
0    4
1    7
2   -5
3    3
dtype: int64
```



# Series

- Often it will be desirable to create a Series with an index identifying each data point:

```
In [6]: obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
```

```
In [7]: obj2
```

```
Out[7]:
```

```
d    4
b    7
a   -5
c    3
dtype: int64
```

```
In [8]: obj2.index
```

```
Out[8]: Index(['d', 'b', 'a', 'c'], dtype='object')
```



# Data Frames

---

- DataFrame represents a tabular, spreadsheet-like data structure containing an ordered collection of columns
  - Each can be a different value type (numeric, string, boolean, etc.).
- DataFrame has both a row and column index
- Numerous ways to construct a DataFrame,
  - Most common is from a dictionary of equal-length lists or NumPy arrays

```
In [18]: data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada',  
                          'Nevada'], 'year': [2000, 2001, 2002, 2001, 2002], 'pop': [1.5, 1.7,  
                                         3.6, 2.4, 2.9]}
```

```
In [19]: frame = pd.DataFrame(data)
```



# Data Frames

---

- The resulting DataFrame will have its index assigned automatically as with Series (starting at index 0), and the columns are placed in sorted order:

```
In [20]: frame
```

```
Out[20]:
```

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



# Data Frames

---

- You can specify the sequence of columns
  - The DataFrame's columns will be exactly what you pass:

```
In [22]: pd.DataFrame(data, columns=['year', 'state', 'pop'])
```

```
Out[22]:
```

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



# Data Frames

---

- A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:

```
In [26]: frame2.state
```

```
Out[26]:
```

```
one      Ohio
two      Ohio
three    Ohio
four     Nevada
five     Nevada
Name: state, dtype: object
```

```
In [27]: frame2.year
```

```
Out[27]:
```

```
one      2000
two      2001
three    2002
four     2001
five     2002
Name: year, dtype: int64
```



# Data Frames

---

- Rows can also be retrieved by position or name by a couple of methods, such as the `.loc` indexing field or `.iloc`(for integer index)

```
In [33]: frame2.loc['three']
```

```
Out[33]:
```

```
year      2002
```

```
state     Ohio
```

```
pop        3.6
```

```
debt       NaN
```

```
Name: three, dtype: object
```



# Data Frames

---

- Like Series, the values attribute returns the data contained in the DataFrame as a 2D ndarray:

```
In [54]: frame3.values
```

```
Out[54]:
```

```
array([[2.4, 1.7],  
       [2.9, 3.6],  
       [nan, 1.5]])
```



# Operations – Unique Values

- Finding unique values in a Data Frame

```
In [58]: df=pd.DataFrame({'col1':[1,2,3,4], 'col2':  
[444,555,666,444], 'col3':['abc', 'def', 'ghi', 'xyz']})
```

```
In [59]: df.head()
```

```
Out[59]:
```

	col1	col2	col3
0	1	444	abc
1	2	555	def
2	3	666	ghi
3	4	444	xyz

- Unique method **unique()** will return an array with the unique values

```
In [60]: df['col2'].unique()
```

```
Out[60]: array([444, 555, 666], dtype=int64)
```



# Operations

---

- What if I want the number of unique values?
  - Can find out the length of the array of unique values using the **len** function:

```
In [61]: len(df['col2'].unique())  
Out[61]: 3
```

- Or the built in method **nunique()**

```
In [62]: df['col2'].nunique()  
Out[62]: 3
```



# Operations

---

- Frequency table with **value\_counts()** function

```
In [63]: df['col2'].value_counts()
```

```
Out[63]:
```

```
444    2
```

```
555    1
```

```
666    1
```

```
Name: col2, dtype: int64
```

# Operations – Selecting Data

- Conditional Selection, specify the criterion for selection:

```
In [64]: df[df['col1']>2]
```

```
Out[64]:
```

	col1	col2	col3
2	3	666	ghi
3	4	444	xyz

- To combine conditions, enclose in parenthesis and use '&' for and or '|' for or (| is the vertical bar not the letter I)

```
In [65]: df[(df['col1']>2)&(df['col2']==444)]
```

```
Out[65]:
```

	col1	col2	col3
3	4	444	xyz



# Operations – Apply function

---

- Aside from standard functions such as `sum()`, you can apply custom functions to your data frames.

```
In [71]: df['col1'].sum()  
Out[71]: 10
```

```
In [69]: def times2(x):  
...:     return x*2  
...:
```

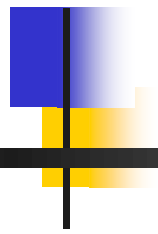
```
In [70]: df['col1'].apply(times2)  
Out[70]:  
0      2  
1      4  
2      6  
3      8  
Name: col1, dtype: int64
```

# Operations – Apply Function

- We can also apply built-in functions
  - For example, say we want to know the length of the strings in one of the columns, we could apply the **len** function:

```
In [72]: df['col3']
Out[72]:
0    abc
1    def
2    ghi
3    xyz
Name: col3, dtype: object
```

```
In [73]: df['col3'].apply(len)
Out[73]:
0     3
1     3
2     3
3     3
Name: col3, dtype: int64
```



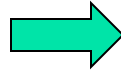
# Operations – isnull()

- Use isnull() to check for null values

```
In [81]: df
```

```
Out[81]:
```

	col1	col2	col3
0	1	444	abc
1	2	555	def
2	3	666	ghi
3	4	444	xyz



```
In [80]: df.isnull()
```

```
Out[80]:
```

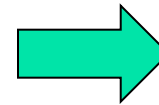
	col1	col2	col3
0	False	False	False
1	False	False	False
2	False	False	False
3	False	False	False

# Pivot\_Table Function

## ■ Pivot tables in data frames

```
In [84]: data={'A':['foo','foo','foo','bar','bar','bar'],'B':  
['one','one','two','two','one','one'],'C':  
['x','y','x','y','x','y'],'D':[1,3,2,5,4,1]}
```

```
In [85]: df=pd.DataFrame(data)
```



```
In [86]: df
```

```
Out[86]:
```

	A	B	C	D
0	foo	one	x	1
1	foo	one	y	3
2	foo	two	x	2
3	bar	two	y	5
4	bar	one	x	4
5	bar	one	y	1

```
In [87]: df.pivot_table(values='D',index=['A','B'],columns=['C'])
```

```
Out[87]:
```

		C	x	y
A	B			
bar	one		4.0	1.0
	two		NaN	5.0
foo	one		1.0	3.0
	two		2.0	NaN





# Filtering out

---

- You have a number of options for filtering out missing data:
  - **dropna** can be very helpful. On a Series, it returns the Series with only the non-null data and index values:

```
In [95]: from numpy import nan as NA
```

```
In [96]: data = pd.Series([1, NA, 3.5, NA, 7])
```

```
In [97]: data.dropna()
```

```
Out[97]:
```

```
0    1.0
```

```
2    3.5
```

```
4    7.0
```

```
dtype: float64
```



# Filtering out

---

- Analogous to Boolean indexing using **notnull()**:

```
In [98]: data[data.notnull()]
```

```
Out[98]:
```

```
0    1.0
```

```
2    3.5
```

```
4    7.0
```

```
dtype: float64
```

# Filtering in Data Frames

- With DataFrame objects, these are a bit more complex. You may want to drop rows or columns which are all NA or just those containing any NAs.
  - **dropna()** by default drops any row containing a missing value:

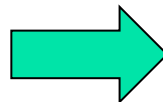
```
In [99]: data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],  
    ...:      ...:      [NA, NA, NA], [NA, 6.5, 3.]])
```

```
In [100]: cleaned = data.dropna()
```

```
In [101]: data
```

```
Out[101]:
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0



```
In [102]: cleaned
```

```
Out[102]:
```

	0	1	2
0	1.0	6.5	3.0

# Filtering

- `how='all'` will only drop rows that are all NA:

```
In [103]: data.dropna(how='all')
```

```
Out[103]:
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
3	NaN	6.5	3.0

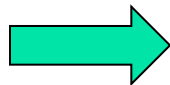
- Dropping columns in the same way is only a matter of passing `axis=1`:

```
In [104]: data[4] = NA
```

```
In [105]: data
```

```
Out[105]:
```

	0	1	2	4
0	1.0	6.5	3.0	NaN
1	1.0	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN
3	NaN	6.5	3.0	NaN



```
In [106]: data.dropna(axis=1, how='all')
```

```
Out[106]:
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

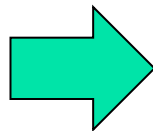
# Filling in Data

- What if you may want to fill in the “holes” in there data?
- **fillna** method is the workhorse function to use. Calling fillna with a constant replaces missing values with that value:

```
In [114]: df
```

```
Out[114]:
```

	0	1	2
0	0.026430	NaN	NaN
1	0.454207	NaN	NaN
2	0.253467	NaN	0.331486
3	-0.491507	NaN	-1.133550
4	-0.020115	0.140531	-2.560774
5	-0.585818	0.162767	0.351409
6	0.321682	0.842006	-0.284016



```
In [115]: df.fillna(0)
```

```
Out[115]:
```

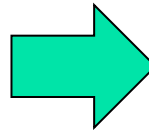
	0	1	2
0	0.026430	0.000000	0.000000
1	0.454207	0.000000	0.000000
2	0.253467	0.000000	0.331486
3	-0.491507	0.000000	-1.133550
4	-0.020115	0.140531	-2.560774
5	-0.585818	0.162767	0.351409
6	0.321682	0.842006	-0.284016

# Filling in Data

- Calling **fillna** with a dictionary you can use a different fill value for each column:

```
In [114]: df
Out[114]:
```

	0	1	2
0	0.026430	NaN	NaN
1	0.454207	NaN	NaN
2	0.253467	NaN	0.331486
3	-0.491507	NaN	-1.133550
4	-0.020115	0.140531	-2.560774
5	-0.585818	0.162767	0.351409
6	0.321682	0.842006	-0.284016



```
In [117]: df.fillna({1: 0.5, 2: -1})
Out[117]:
```

	0	1	2
0	0.026430	0.500000	-1.000000
1	0.454207	0.500000	-1.000000
2	0.253467	0.500000	0.331486
3	-0.491507	0.500000	-1.133550
4	-0.020115	0.140531	-2.560774
5	-0.585818	0.162767	0.351409
6	0.321682	0.842006	-0.284016



# Filling in Data

---

- With **fillna** you can also pass the mean or median value of a Series (common method for replacing missing values):

```
In [118]: data = pd.Series([1., NA, 3.5, NA, 7])
```

```
In [119]: data.fillna(data.mean())
```

```
Out[119]:
```

```
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

```
In [120]: data.fillna(data.median())
```

```
Out[120]:
```

```
0    1.0
1    3.5
2    3.5
3    3.5
4    7.0
dtype: float64
```

# GroupBy

- As we learned in SQL, we can create aggregations in Python as well
- Groupby allows you to group together rows based off of a column and perform an aggregate function on them

The diagram illustrates the concept of GroupBy aggregation. It shows a large table on the left with columns 'ID' and 'Value', partitioned into three groups based on the 'ID' column. Arrows indicate that the 'Value' for each 'ID' is summed across all rows in that partition to produce the aggregated values in the table on the right.

	ID	Value
Partition 1	1	50.30
	1	123.30
	1	132.90
Partition 2	2	50.30
	2	123.30
	2	132.90
	2	88.90
Partition 3	3	50.30
	3	123.30

ID	Value
1	306.50
2	395.40
3	173.60



# Groupby Method

- The groupby method allows you to group rows of data together and call aggregate functions

```
In [122]: data = {'Company':  
['GOOG', 'GOOG', 'MSFT', 'MSFT', 'FB', 'FB'],  
...:             'Person':  
['Sam', 'Charlie', 'Amy', 'Vanessa', 'Carl', 'Sarah'],  
...:             'Sales':[200,120,340,124,243,350]}
```

```
In [123]: df = pd.DataFrame(data)
```

```
In [124]: df
```

```
Out[124]:
```

	Company	Person	Sales
0	GOOG	Sam	200
1	GOOG	Charlie	120
2	MSFT	Amy	340
3	MSFT	Vanessa	124
4	FB	Carl	243
5	FB	Sarah	350



# Groupby Method

---

- Use **.groupby()** method to group rows together based off of a column name.
  - For example let's group based off of Company. This will create a DataFrameGroupBy object:

```
In [125]: df.groupby('Company')
```

```
Out[125]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001927E40B4F0>
```

- We can save this object as a new variable:

```
In [126]: by_comp = df.groupby("Company")
```



# Groupby Method

---

- With this variable we can call aggregate methods, such as `mean()`:

```
In [127]: by_comp.mean()
```

```
Out[127]:
```

	Sales
Company	
FB	296.5
GOOG	160.0
MSFT	232.0

```
In [128]: df.groupby('Company').mean()
```

```
Out[128]:
```

	Sales
Company	
FB	296.5
GOOG	160.0
MSFT	232.0

# Or Describe...

```
In [133]: by_comp.describe()
```

```
Out[133]:
```

	Sales							
	count	mean	std	min	25%	50%	75%	max
Company								
FB	2.0	296.5	75.660426	243.0	269.75	296.5	323.25	350.0
GOOG	2.0	160.0	56.568542	120.0	140.00	160.0	180.00	200.0
MSFT	2.0	232.0	152.735065	124.0	178.00	232.0	286.00	340.0

```
In [134]: by_comp.describe().transpose()
```

```
Out[134]:
```

Company	FB	GOOG	MSFT
Sales count	2.000000	2.000000	2.000000
mean	296.500000	160.000000	232.000000
std	75.660426	56.568542	152.735065
min	243.000000	120.000000	124.000000
25%	269.750000	140.000000	178.000000
50%	296.500000	160.000000	232.000000
75%	323.250000	180.000000	286.000000
max	350.000000	200.000000	340.000000

```
In [142]: by_comp.describe().transpose()['GOOG']
```

```
Out[142]:
```

Sales count	2.000000
mean	160.000000
std	56.568542
min	120.000000
25%	140.000000
50%	160.000000
75%	180.000000
max	200.000000

```
Name: GOOG, dtype: float64
```



# Intro to Matplotlib

---

- Matplotlib is the "grandfather" library of data visualization with Python.
- Created by John Hunter. He created it to try to replicate MatLab's plotting capabilities in Python.
- If you happen to be familiar with matlab, matplotlib will feel natural to you



# Basic Commands

---

- We can create a very simple line plot using the following `plt.plot`:

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

```
In [4]: x
```

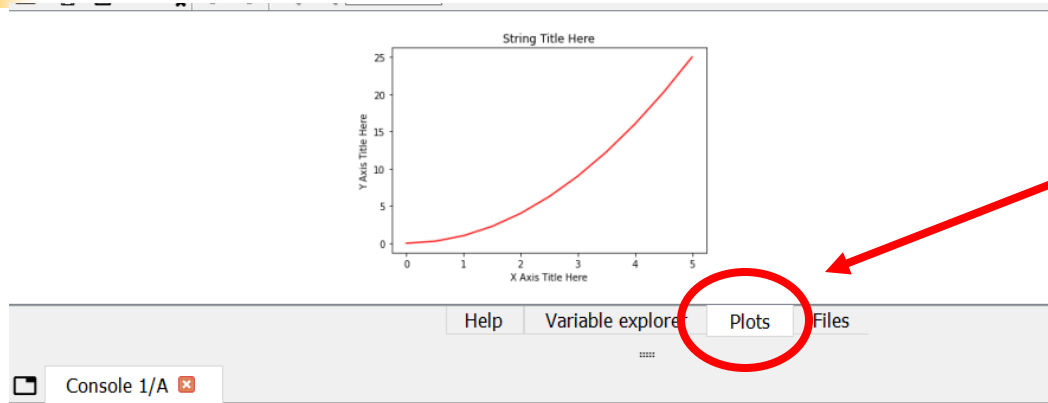
```
Out[4]: array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ])
```

```
In [5]: y
```

```
Out[5]:  
array([ 0. ,  0.25,  1. ,  2.25,  4. ,  6.25,  9. , 12.25, 16. ,  
       20.25, 25. ])
```

```
In [6]: plt.plot(x, y, 'r') # 'r' is the color red  
....: plt.xlabel('X Axis Title Here')  
....: plt.ylabel('Y Axis Title Here')  
....: plt.title('String Title Here')  
....: plt.show()
```

# Basic Commands



The plot will be shown in the Plots Pane on Spyder...

- you can show the plots in the console if you're using Jupyter Notebooks using the command:  
`%matplotlib inline`

```
In [4]: x
Out[4]: array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. ,

In [5]: y
Out[5]:
array([ 0. ,  0.25,  1. ,  2.25,  4. ,  6.25,  9. , 12.
        20.25, 25.  ])

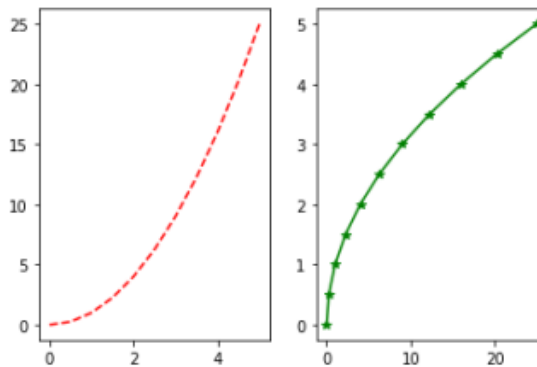
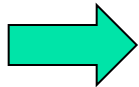
In [6]: plt.plot(x, y, 'r') # 'r' is the color red
....: plt.xlabel('X Axis Title Here')
....: plt.ylabel('Y Axis Title Here')
....: plt.title('String Title Here')
....: plt.show()
```



# Multiplots

- Creating Multiplots on Same Canvas:

```
In [9]: # plt.subplot(nrows, ncols, plot_number)
...: plt.subplot(1,2,1)
...: plt.plot(x, y, 'r--')
...: plt.subplot(1,2,2)
...: plt.plot(y, x, 'g*-');
```





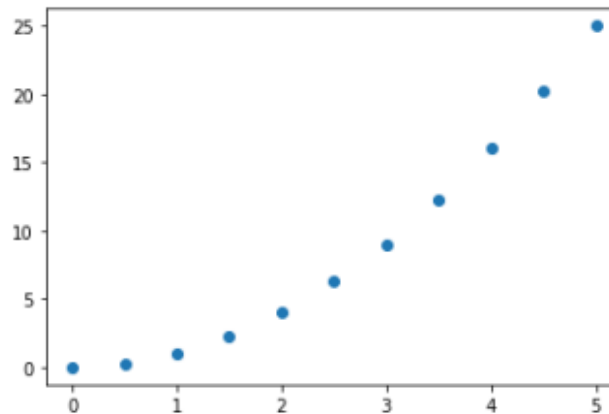


# Special Plot Types

- There are many specialized plots we can create, such as barplots, histograms, scatter plots:
  - Scatter Plot

```
In [29]: plt.scatter(x,y)
```

```
Out[29]: <matplotlib.collections.PathCollection at 0x25bf412ad90>
```

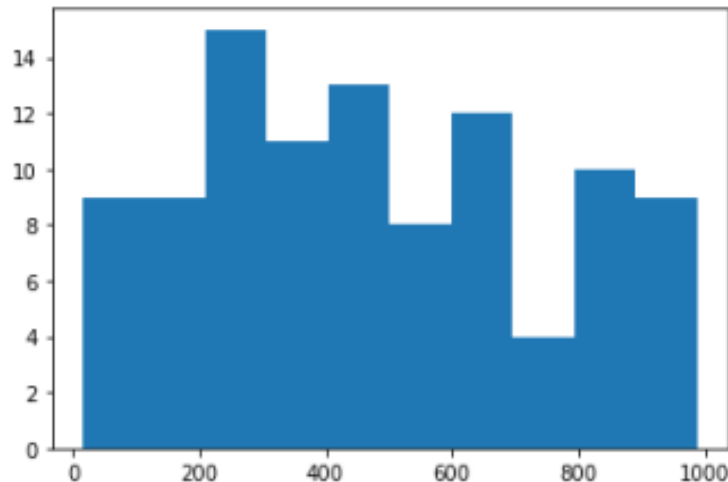


# Special Plot Types

## ■ Histogram:

```
In [30]: from random import sample
...: data = sample(range(1, 1000), 100)
...: plt.hist(data)
```

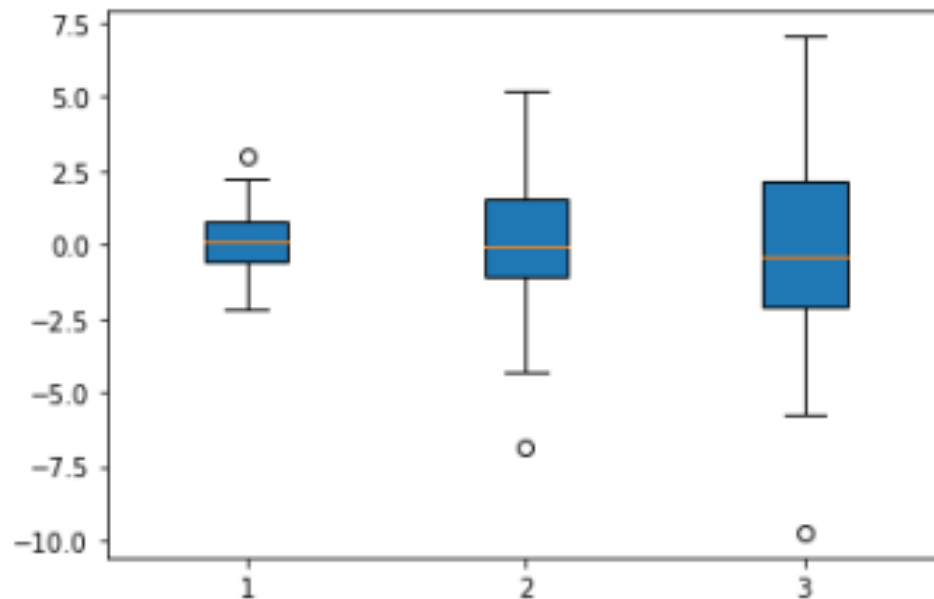
```
Out[30]:
(array([ 9.,  9., 15., 11., 13.,  8., 12.,  4., 10.,  9.]),
 array([ 16. , 113.1, 210.2, 307.3, 404.4, 501.5, 598.6, 695.7, 792.8,
        889.9, 987. ]),
 <BarContainer object of 10 artists>)
```



# Special Plot Types

## ■ Boxplot:

```
In [31]: data = [np.random.normal(0, std, 100) for std in range(1, 4)]  
...:  
...: # rectangular box plot  
...: plt.boxplot(data,vert=True,patch_artist=True);
```

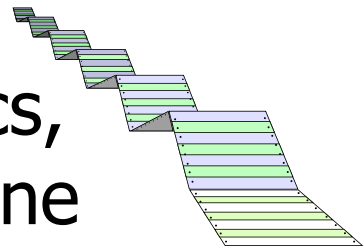
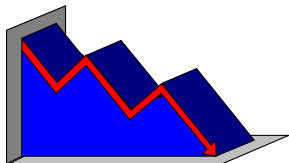




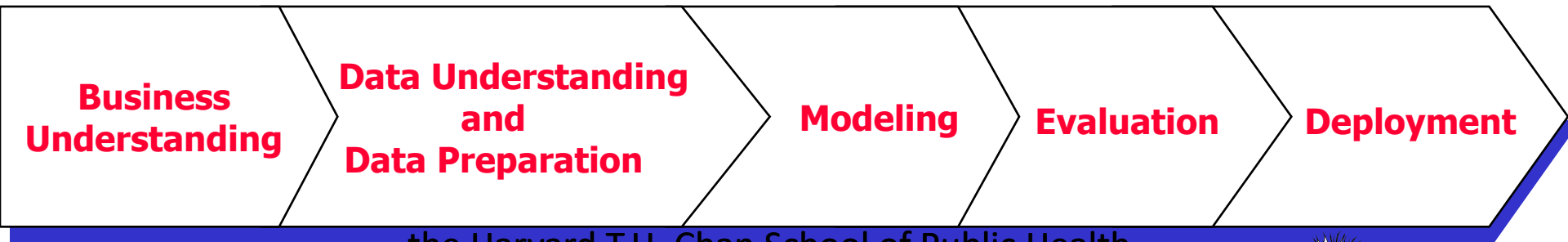
# What is Analytics?

---

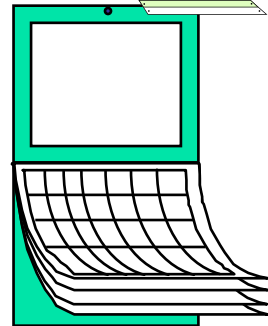
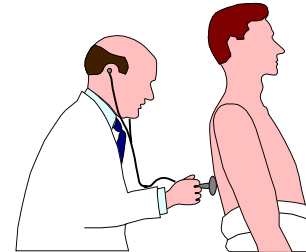
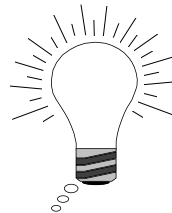
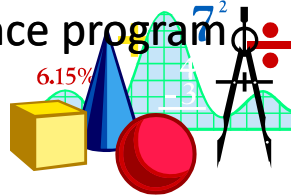
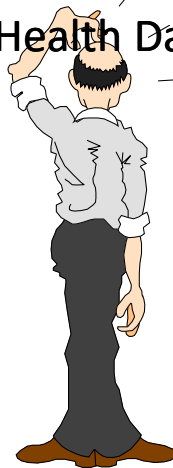
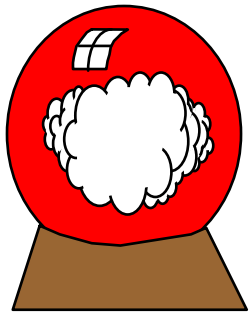
- Analytics uses data and math to answer business questions, discover relationships, predict unknown outcomes and automate decisions.
- This diverse field is used to find meaningful patterns in data and uncover new knowledge based on applied mathematics, statistics, predictive modeling and machine learning techniques



# Process



the Harvard T.H. Chan School of Public Health -  
Health Data Science program





# Supervised vs Unsupervised

---

- Within artificial intelligence (AI) and machine learning, there are two basic approaches: supervised learning and unsupervised learning. The main difference is one uses labeled data to help predict outcomes, while the other does not.
  - Supervised(inputs and output): Regression, Decision Trees, Random Forest
  - Unsupervised(inputs): Clustering, Neural Networks, Dimensionality Reduction

# Basic Modeling Methods

## (supervised learning)

---

- **Regression** - A linear equation of predictor variables (for continuous output or target)
- **Logistic Regression** – A variation of linear regression used to predict probabilities (for categorical output or target)
- **Decision Trees/CART/Random Forest** - A hierarchical structure of significant variables in order of importance (works for either continuous or categorical output or target)



# Multiple Linear Regression Model

---

- The coefficients  $\beta_0, \dots, \beta_p$  and the standard deviation of the noise ( $\sigma$ ) determine the relationship in the population of interest.
  - Estimate them from the data using a method called *ordinary least squares* (OLS).
  - Minimize the sum of squared deviations between the actual values ( $Y$ ) and their predicted values based on that model ( $\hat{y}$ ).
- To predict the value of the dependent value from known values of the predictors,  $x_1, x_2, \dots, x_p$  we use sample estimates for  $\beta_0, \dots, \beta_p$  in the linear regression model
  - The predicted value,  $\hat{y}$ , is computed from the equation
    - $\hat{Y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \dots + \hat{\beta}_p x_p$





# MLR Model

---

- Predictions based on this equation will be unbiased (equal to the true values on average) and will have the smallest average squared error compared to any unbiased estimates *if* we make the following assumptions:
  - The noise  $\varepsilon$  (or equivalently, the dependent variable) follows a normal distribution.
  - The linear relationship is correct.
  - The cases are independent of each other.
  - The variability in  $Y$  values for a given set of predictors is the same regardless of the values of the predictors (*homoskedasticity*).

# Multiple Linear Regression

- Predict a single **response** variable,  $Y$ , as a linear function related to  $k$  (explanatory variables) **inputs**

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_k X_k + \varepsilon$$

$$e_i(\text{residual}) = y_i - \hat{y}_i \quad (\hat{y} = \text{fitted value using equation})$$

- Regression coefficients represent:
  - **Expected** change in  $y$  when the related  $x$  is changed by one unit **and** all other inputs are held constant

# Multiple Linear Regression: Input Variables ( $X$ 's)



---

- $X_i$ 's may consist of different variable types
  - Continuous (preferred)
  - Binary (0/1 variable may be linearly modeled to any output)
  - Discrete (more than 2 levels with order)
    - Practically, we may include Ordinal – though one must be very careful about interpretation of results as true magnitude between levels is unknown
  - Note: nominal variables (e.g., Region A vs. B vs. C) require conversion to 'indicator variables'



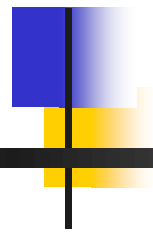
# Train Test Split

---

- Now let's split the data into a training set and a testing set. We will train out the model on the training set and then use the test set to evaluate the model.

```
In [11]: from sklearn.model_selection import train_test_split
```

```
In [12]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,  
random_state=101)
```



# Creating and Training the Model

---

```
In [13]: from sklearn.linear_model import LinearRegression
```

```
In [14]: lm = LinearRegression()
```

```
In [15]: lm.fit(X_train,y_train)
```

```
Out[15]: LinearRegression()
```

# Model Evaluation

- Let's evaluate the model by checking out its coefficients and how we can interpret them.

```
In [16]: # print the intercept
...: print(lm.intercept_)
-2640159.796851911
```

```
In [31]: ► coeff_df = pd.DataFrame(lm.coef_.T, index = X.columns, columns=['Coefficient'])
print(coeff_df)
```

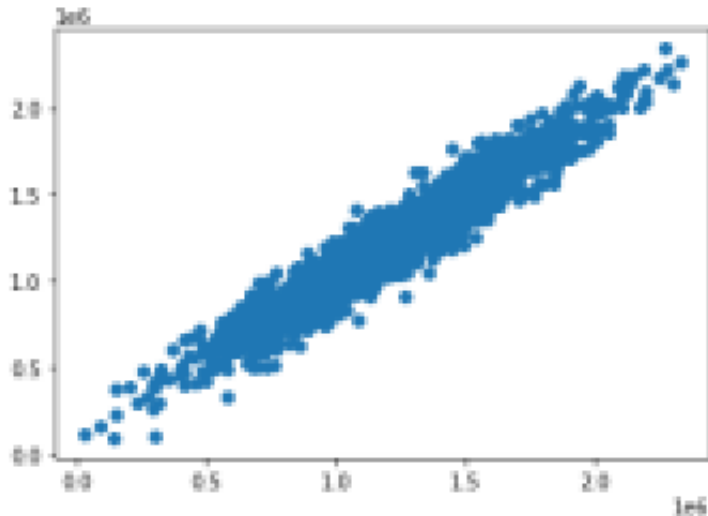
	Coefficient
Avg. Area Income	21.528276
Avg. Area House Age	164883.282027
Avg. Area Number of Rooms	122368.678027
Avg. Area Number of Bedrooms	2233.801864
Area Population	15.150420

# Predictions from our Model

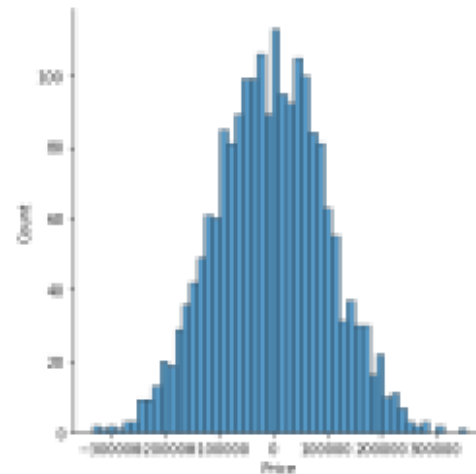
```
In [18]: predictions = lm.predict(X_test)
```

```
In [19]: plt.scatter(y_test, predictions)
```

```
Out[19]: <matplotlib.collections.PathCollection at 0x2b0545521c0>
```



```
In [21]: sns.displot((y_test-predictions),bins=50);
```





# Regression Evaluation Metrics

- Here are three common evaluation metrics for regression problems:
- **Mean Absolute Error** (MAE) is the mean of the absolute value of the errors:

$$\frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

- **Mean Squared Error** (MSE) is the mean of the squared errors:

$$\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

- **Root Mean Squared Error** (RMSE) is the square root of the mean of the squared errors:

$$\sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$





# R-squared

- Proportion of the variance for a dependent variable that's explained by an independent variable or variables in a regression model.

$$R^2 = 1 - \frac{RSS}{TSS}$$

$R^2$  = coefficient of determination

$RSS$  = sum of squares of residuals

$TSS$  = total sum of squares

$$RSS = \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

$$TSS = \sum_{i=1}^n (y_i - \bar{y}_i)^2$$



# Calculation of Evaluation Metrics

---

- **MAE** is the easiest to understand, because it's the average error.
- **MSE** is more popular than MAE, because MSE "punishes" larger errors, which tends to be useful in the real world.
- **RMSE** is even more popular than MSE, because RMSE is interpretable in the "y" units (MSE is in squared units)
- All of these are **loss functions**, because we want to minimize them.

```
In [22]: from sklearn import metrics
```

```
In [23]: print('MAE:', metrics.mean_absolute_error(y_test, predictions))
...: print('MSE:', metrics.mean_squared_error(y_test, predictions))
...: print('RMSE:', np.sqrt(metrics.mean_squared_error(y_test, predictions)))
```

```
MAE: 82288.22251914957
```

```
MSE: 10460958907.209501
```

```
RMSE: 102278.82922291153
```



# Full Report

In [34]: `import statsmodels.api as sm  
X_train_temp = sm.add_constant(X_train)  
results = sm.OLS(y_train, X_train_temp).fit()  
results.summary()`

Out[34]: OLS Regression Results

<b>Dep. Variable:</b>	Price	<b>R-squared:</b>	0.918
<b>Model:</b>	OLS	<b>Adj. R-squared:</b>	0.918
<b>Method:</b>	Least Squares	<b>F-statistic:</b>	6715.
<b>Date:</b>	Tue, 21 Nov 2023	<b>Prob (F-statistic):</b>	0.00
<b>Time:</b>	11:20:22	<b>Log-Likelihood:</b>	-38807.
<b>No. Observations:</b>	3000	<b>AIC:</b>	7.763e+04
<b>Df Residuals:</b>	2994	<b>BIC:</b>	7.766e+04
<b>Df Model:</b>	5		
<b>Covariance Type:</b>	nonrobust		

	coef	std err	t	P> t	[0.025	0.975]
<b>const</b>	-2.64e+06	2.22e+04	-119.047	0.000	-2.68e+06	-2.6e+06
<b>Avg. Area Income</b>	21.5283	0.174	124.039	0.000	21.188	21.869
<b>Avg. Area House Age</b>	1.649e+05	1883.872	87.524	0.000	1.61e+05	1.69e+05
<b>Avg. Area Number of Rooms</b>	1.224e+05	2082.358	58.764	0.000	1.18e+05	1.26e+05
<b>Avg. Area Number of Bedrooms</b>	2233.8019	1683.015	1.327	0.185	-1066.181	5533.785
<b>Area Population</b>	15.1504	0.184	82.391	0.000	14.790	15.511